Table of Contents

An OS9 Tutorial

Bob Montowski

Table of Contents

1. Lesson 1

This will be my first tutorial on using OS-9 and it will be for the beginners who bought OS-9 and are now ripping their hair out trying to figure out how to use it now that they have it... OS-9 is *not* a programming language. It is totally different from BASIC and if you wish to program in Basic then I suggest you buy Basic09 after you are a bit familiar with OS-9. For people who have Disk Basic 1.0 you will need to load the OS-9 BOOT disk and RUN"*". This will then tell you to put the OS-9 Master Disk in Drive 0 and push any key to continue. If you have Disk Basic 1.1 then all you need to do is put the OS-9 Master Disk in Drive 0 and type DOS... Now that OS-9 has started up and given you your Logo and license information it will ask you for the DATE and TIME. This information is *very* important and should be given correctly each time you start up OS-9. Do *not* just hit enter, give a date and time. This information is added to each file as it is saved to disk and will be used by the OS-9 in the future to keep track of current files. The same information is also available to you to help you keep tabs on the dates and times of the files that you saved to disk. OS-9 runs on a 24 hour clock so when giving the time you must remember that times after 12 noon convert to the following:

1 pm-1300 hours
2 pm-1400 hours
3 pm-1500 hours
.
.
.
10 pm-2200 hours
11 pm-2300 hours
midnite-0000 hours

To enter Dec 25, 1985...3:30 pm you would type

```
YY/MM/DD HH:MM:SS
85/12/25 15:30:00
```

After a date and time have been given to OS-9 you may check this time anytime you want from OS-9 by typing DATE T at OS9: prompt. If you just say DATE that is all you will get. You must say DATE T to get the date and the time... OS-9 has only a few commands already in memory. All the rest of the commands that you can use from OS-9 are on your Master Disk. Each time you give a command at the OS9: prompt the computer will check to see if the command is in memory and then it will go to the disk in drive 0 and check the /D0/CMDS directory to see if the command is in there. You must remember to type the command in correctly (SPELLING) or it won't be found when the computer goes to the /D0/CMDS directory looking for it. OS-9 can be a bit slow as it has to go to the /D0/CMDS directory each time you type a command at the OS-9 prompt but you can speed this up a bit by loading some of the commands that you will use the most in OS-9. So you could type:

```
OS9:load dir list del attr copy
```

You will now have the commands **dir**, **list**, **del**, **attr**, **copy** all in memory and they are ready for quick access. The drawback is that they are taking up memory that you might need later. The only way around this right now is to either set your drives to run at a new faster step rate (another tutorial) or to get a Hard Disk Drive for use with your OS-9. Radio Shack had OS-9 coded to run the disk drives at 30 MS. track to track and to format the disk as 35 tracks. Both of these can be changed with a little knowledge of OS-9 or by buying some commercial software that will make the changes in OS-9 for you. Another way to speed up OS-9 is to add a 256K Ram Disk to your CoCo. With the 256K Ram board installed and the right software added to OS-9 the extra memory will act like a *very* fast 40 track disk drive.

OS-9 always has 2 directories that it keeps track of. One is the DATA and the other is the EXECUTION directory. When you type a command OS-9 will check the current EXECUTION directory which is /D0/CMDS at startup for the command you just typed in. When you go to do a list, dir, del, rename, etc... OS-9 is going to do to the current DATA directory and look for your file there. The current DATA directory at startup is /D0. So if you just type **`DIR`**, OS-9 will go and assume you meant DIR /D0. If you wish to get a directory of say the DEFS directory you must give OS-9 the whole pathlist (NAME) to the directory. In this case you would type: **`DIR /D0/DEFS`** and OS-9 will know which directory you are talking about. So how do you know what is a command? Or what is a data file? Or what is a directory? You can get this information by typing: **`DIR E /D0`** and OS-9 will give you a directory of everything that is in the /D0 directory with exact information on each entry in that directory. You will get the date and time the entry was put on the disk and the user number (0 which means you), the entry's name, the attributes of the entry and the size of the entry in hexadecimal. It is the attributes of an entry that we will want to check. They list across like this:

```
DSPPPERW
  EWR
--------
```

That is 8 slots that can have a letter in it. If the **dir e** command shows this on a line

```
D--RW-RW
```

It would mean that it is a directory and that you and any timesharing users you had on your system could read and write to that directory... If the entry gives this back:

```
--E--ERW
```

It would mean that it is a command that can be used by you and your timesharing users and that you have the right to say copy that file, rename that file or delete that file. The timesharing user would only be able to execute the file.

If you don't want to do a **dir e** on a whole disk than you can get the information you need on a single entry by typing:

```
ATTR /D0/startup
```

this will printout the attributes in the same manner as the **dir e** command did, but you now have the option of changing the attributes of a file on the disk. We'll use the /D0/startup file for an example. say the **`ATTR /D0/startup`** prints this

```
------rw
```

This means that the file can be read and written to. But say you don't want to accidently delete or rename the file in the future? You can type:

```
ATTR /D0/startup -w
```

and the write ability to that file will be taken away. If you tried to delete that file now you would get an error message. You can use this **attr** command to change the

attributes on all your important files so that they will not be deleted by accident in the future. This is kind of like having a write protect tab on your disk like in Disk Basic. But you can protect single files on the disk. Or even lock out a DATA directory from having files written or deleted from it.

When I told you that OS-9 will check to see if a command is in memory and then check for it in the EXECUTION directory I left out a final thing that it does. It will go to the DATA directory and check to see if there is a DATA file there with the same name as what you typed in at the OS-9 prompt. You can check this out yourself. LIST the file startup like this:

```
list /D0/startup
```

You will see this:

```
setime <term
```

it looks like a command right? Well it is what OS-9 calls a procedure file. OS-9 will take the command you type in and first check to see if it is in memory, if that fails it will go to the EXECUTION directory and see if the command is there, if that fails it will go to the DATA directory and see if there is a procedure file there with the name you typed in. If there is it will read one line at a time from that file and treat it like you were typing in the lines from the keyboard. If you want to try this, just type startup at any OS-9 prompt and the system will ask you again for the DATE and TIME to use on the system. You can build a procedure file of your own that does a little more than the startup file does. *Do this* at the OS-9 prompt:

```
OS9:build /d0/myfile
```

you will then see a (?) at each (?) type these lines

```
? dir /d0
? dir /d0/cmds
? mfree
? free
? (enter)
```

You will now have a data file on /D0 called myfile. If you were to type myfile at an OS-9 prompt you will then see a DIR of /D0 and then a DIR of /D0/CMDS and then you will get a mfree (memory free), and finally you will get a free (free disk space) all listed to your screen one at a time. OS-9 did all the commands in the data file as if you just typed them in at the keyboard. Not bad huh?

Now the next important thing to worry about with OS-9 is how does it keep tabs on free space in memory and on the disks? Memory in the computer is split up in blocks of 256 bytes. If you do a **mfree** you will get back about 159 to 162 blocks of memory. If you know that 4 blocks of 256 bytes makes one K (kilobyte) then you know you have about 40K free in memory for your programs and commands. This same idea is carried over to the disk drive. All writes to the disk are done in blocks of 256 bytes or 1 sector. A newly formatted disk will have about 630 sectors on it. But 10 of these sectors are taken away for use as directory pointers. As OS-9 only writes out to the disk in blocks of 256 bytes you will be able to get more information on an OS-9 disk than a Radio Shack DOS disk which stores data to the disk in blocks of 9 sectors (9*256=2304 bytes). Write 1 character to an OS-9 disk and you lose 1 sector. Write 1 character to a RS DOS disk and you lose 9 sectors!!!

Now do a **`dir /D0/CMDS`** and you will see quite a long list of commands that are available to you. Don't worry about all those titles because as you learn OS-9 you will become familiar with all of them and probably not use all of them. The nice thing about OS-9 that was so different from RS Disk Basic was that it is so easy to add *more* commands to OS-9 than it was to add commands to the RS DOS. If you know 6809 machine language you might even write some commands that you will find useful and might want to sell or trade with other OS-9 users. If you aren't all

that familiar with machine language then you can buy some new commands for OS-9 from companies like Frank Hogg or from Computerware or D.P. Johnson. These are commands that are so easy to install on your OS-9 disk! All you need to do is copy them to your EXECUTION directory which is usually the /D0/CMDS directory. They are then available for your use. No worry on your part as to will they work with your OS-9! Some of these programs are actual commands that you call from OS-9 and other programs are what are called *filters* that you pipe data through under OS-9.

And now one final thing to cover on OS-9 before I end this lesson. Is there a difference between upper and lower case when you type in commands? The answer is no... no... no... If you type in **`DIR /D0`** or **`dir /d0`** they will both act correctly... if you type **`LIST /D0/STARTUP`** or **`list /d0/startup`** they will both work correctly. OS-9 doesn't care about the case of the commands you type in. But here is a standard that you might wish to keep to so that what is on your disk are a bit easier to understand. It is felt that if you keep all directory names in capital letters and all data/command files in lower case you will have a better idea of what is on your disk when you use the **dir** command. I find this a useful tip and try to follow it strictly when I work with OS-9.

The next lesson in the tutorial series will be on nested directories and on pipe and filters and how they are most useful under OS-9.

2. Lesson 2

OK, glad to see that you are back for lesson #2. I'll cover the way the OS-9 has multiple directories and how each directory can have directories within that. What is this good for and how can you use this on your OS-9 system? Well, first it makes it very easy to put your files on your disks in a manner that will make it easy for you to find those files again in the future... Let's take a blank disk and format it. If you have one drive do this:

```
OS9:load format dir makdir build free
```

take your Master Disk out of drive 0 and put in a blank disk. Now at the OS9 prompt you type format /d0. You will be asked if you really want to format the disk in drive 0? Type Y for yes. When the format is done you will be asked for a name to put on the disk. Each disk you format under OS-9 will have a NAME on the disk. For the time being we'll call this disk JUNK DISK. OS-9 will check the disk to be sure all the sectors are good and if not OS-9 will lock out the bad sectors from the directory. This means that you could use a scratched disk that you were not able to format under RS DOS... But you should beware of doing this if you intend to put anything *really* important on this disk. Now that the format is done at the OS-9 prompt. Type:

```
OS9:free /d0.
```

This will check the disk you just did the format on in drive 0 and it will tell you the name of the disk... how many total sectors there are on the disk and how many of those sectors are available for you to store data in. OS-9 is set up for a 35 track system with 18 sectors per track. This gives you 630 sectors total on the disk and OS-9 will take 10 of those sectors for its Directory information. If you do not see 620 sectors free for use than the disk had some bad sectors on it and you might not want to put anything important on this disk. But for now we will just experiment with the disk. at the OS9 prompt type:

```
OS9:makdir /d0/LETTERS
OS9:makdir /d0/BILLS
OS9:makdir /d0/LETTERS/FROM.JOE
OS9:makdir /d0/LETTERS/FROM.SUE
OS9:makdir /d0/LETTERS/FROM.TOM
OS9:makdir /d0/BILLS/PHONE
OS9:makdir /d0/BILLS/GAS
OS9:makdir /d0/BILLS/FOOD
```

```
OS9:chd /d0
OS9:dir /d0
```

....You will see that the **dir** returns LETTERS BILLS as what is on the disk in drive 0 But you made 6 directories... so where are the other ones? Try this:

```
OS9:dir /d0/BILLS
PHONE     GAS     FOOD
```

is what you will get. See how you can cluster important stuff in directories so that it has a logical flow and you can work your way down through the levels of the directories to get the information you want???? Try this:

```
OS9:dir /d0/LETTERS
FROM.JOE   FROM.SUE   FROM.TOM
```

is what you get... in a real life situation... say the business world you could then do this:

```
OS9:dir   /d0/LETTERS/FROM.TOM
```

and you would see the letters you stored from someone called TOM... A very neat, logical way to store and retrieve data from your disk. When you get into owning double sided disks for storage or even a Hard Disk drive for storage you will see how this makes it easier to get to your information. Imagine having a Hard Disk under RS DOS? A Dir of that drive might return a directory listing some 100-500 lines long... It would be a real pain to read all those titles and try to find the file you wanted to del or copy or rename...

If you have a two drive OS-9 system then you can go through this exercize too by just putting the disk you wish to format in drive /d1 and changing all the makedir and dir statements I gave so they say /d1 instead of /d0.

Now in the prior example I showed you the command chd... what is this? Well OS-9 has two commands built into it and you can call them to tell OS-9 that you are changing your DATA directory or your EXECUTION directory. Now this is very important to remember!!! If you take the Master Disk out of drive /d0 and put in a new Master Disk that say has more commands in its /D0/CMDS directory you *must* tell OS-9 that you did this... you do this by:

```
OS9:chx /d0/cmds
OS9:chd /d0
```

OS-9 will then check this disk so it will know where the DATA directory is on the disk and where the EXECUTION directory is on the disk. It will *not* always be in the same spot on each disk. You might be used to RS DOS where the directory was *always* on track 17, but this is not true under OS-9. OS-9 must always know where these two directories are before it will do a read/write for that disk. Another benefit of the chd and chx command are to save you some typing. So if you are not a quick or accurate typist these commands are a real boon to you. Take the example above where we had directories within directories. If you wished to copy files or delete files or build files in the /d0/letters/from.joe directory you would think you would have to type that long line each time. You could for your own piece of mind but there is a shortcut to all that typing... do this:

```
OS9:chd /d0/letters/from.joe
```

If you do a dir now you will see that there are *no* files in the directory you are in... You could build a file in this new DATA directory by typing

```
OS9:build /d0/letters/from.joe/june.1st
```

or you could just say:

```
OS9:build june.1st
```

as you used the chd command earlier OS-9 knows to add that whole string of characters in front of june.1st to make the whole pathlist to where you wish to build a file. You see that there is less chance of a typing error in this shorthand method rather than typing out that long string of characters each time....

When you go to makdir or build something on the disk you have to keep in mind that OS-9 expects titles of directories and files to obey certain rules. The names of these files/directories *must* begin with a letter(upper/lower case) and may have no spaces in the title. If you wanted to build a file called: a letter from my buddy You would need to type it in as: a.letter.from.my.buddy for OS-9 to accept it... you could have even typed it in as: aletterfrommybuddy But this is a bit harder to read... Another character you can use to separate words for easier reading is the left arrow sign... this can be made by typing the clear key/minus sign together... depending on the type screen you are reading OS-9 on you will see a left arrow or an underline. They are both the same ascii character. But the character. rom on the CoCo was setup for the left arrow sign. This clear key/minus key is a bit hard to remember and harder yet to type so I use the period(.) to separate my words in my titles and directories. File names and directories can be up to 29 characters long... You can have numbers mixed into this but the first characters of each file/directory *must* be a letter! so these names are perfectly legal:

```
number11111111    jan281985
q1234567890       a2gggg8888cccc9999
```

NO SPECIAL CHARACTERS MAY BE USED IN A FILE NAME OR A DIRECTORY NAME!!! This means no !@#$%^&*+="'- are allowed in any title. Some of these characters are used by OS-9 to perform other useful functions that will be covered in a future lesson.

OS-9 has the ability to take information and pass it through a pipe into a filter to change the information in some manner before showing it on your screen or your printer. The command for a pipe is the exclamation point(!). A filter can be thought of as a program that will take data in and do something with it before passing some data out. The ONLY filter that you have with your original OS-9 is the filter called TEE. If you were to do this:

```
OS9:list startup ! tee /d0/f1 /d0/f2
```

it would list the data in the file called startup through the pipe (!) into the filter TEE... this program would then send the data out to two files that are called /d0/f1 /d0/f2 and you would have two perfect copies of the file startup called f1 and f2... You could have done this same thing by typing:

```
OS9:copy /d0/startup /d0/f1
OS9:copy /d0/startup /d0/f2
```

TEE will take any data that is piped into it and send it to the list of devices or files that are printed after the tee command and separated by spaces. so a line like this:

```
OS9:dir /d0 ! TEE /d0/stuff /p /d0/s2
```

will send a directory of /d0 to your screen, a file called /d0/stuff, to the printer, and to a file called /d0/s2. This is a way to get some data to a lot of different places all at about the same time. There are other filters you can buy that will do the following:

```
OS9:list startup ! upper
```

this will take any data in a file called startup and send it through the filter called upper... upper will take ALL lower case letters and change them to upper case before passing that information on...

```
OS9:list startup ! wc
```

This command will list the file startup through the pipe into the filter wc which will count the # of lines, characters, and words in the file which **wc** will then print out to your screen. Imagine doing that by hand? There are a *lot of filters* that you can buy. Check the Official OS-9 Tour Guide out for a list of the filters you can buy and who sells them.

A piece of advice now. If you think you are going to get into OS-9 you should consider getting a 2nd disk drive if you now only have 1 drive. OS-9 can be run on a single drive CoCo but it is a real PAIN IN THE BUTT!!! On a 2 drive system you can keep all your commands on drive /d0 and all your data on drive /d1 and should speed along just fine and not worry about space being tight on your drives when you go to build files in the future.

Until you become familiar with OS-9 and the way it gives error #'s instead of letter codes for the errors you make then do this:

```
OS9:printerr
```

this will then print the error # of any error you have and give you a short english(???) sentence of what was wrong. As you use OS-9 more you will find that you will likely get 4-6 error codes that tend to repeat a lot. Most errors on OS-9 are caused by typing errors when entering directory names or file names.

I hope I don't appear to be jumping about two much with these lessons but I am trying to tackle the problems of OS-9 in the same manner that I ran into them and am sure other new users are finding them. So I give this final bit of advice... OS-9 comes with 3 books. a red(purple) one, a blue one and a yellow(orange)one. Don't even look at the blue book yet. It has machine language information in it that you *may never* use unless you get into machine language programming under OS-9. The red book will tell you all the commands available under OS-9 and a bit on how they work. READ THIS BOOK OVER AND OVER AT LEAST 4 TIMES!!! The yellow book has information on the text editor that comes with OS-9. This is covered in the 1st half of the book... READ THIS VERRRRRY CLOSELY... IT GETS COMPLICATED but there are a lot of examples... The 2nd half of the yellow manual also has some machine language information in it for doing assembly of machine language programs... If you are not going to get into machine language ever then you don't need to read this info. Even though you have these 3 fine manuals, run to the nearest RS store and buy the Official OS-9 Tour Guide. It has more information than these 3 manuals and it is written in a lot friendlier manner than these 3 manuals. It also gives better examples on how to use the commands available to you under OS-9 and give a history of OS-9 and why it is such a fine Operating System to run on your CoCo.

Next lesson? I will tell you the commands that you might never use and how to delete them to make more space on your Master Disk.

3. Lesson 3

Below is a list of the commands that you may seldom use and therefore may delete from your EXECUTION directory which is /D0/CMDS...

> **Note:** NONE OF THESE COMMANDS SHOULD BE DELETED FROM YOUR ORIGINAL DISK... NEVER DELETE FILES/DIRECTORIES/COMMANDS FROM YOUR ORIGINAL OS-9 MASTER DISK. ALWAYS MAKE THESE DELETIONS/CHANGES TO A BACKUP OF YOUR MASTER DISK

To make a backup of your OS-9 Master disk do this: For a single drive system you must at the OS-9 prompt type this:

```
OS9:load format free
```

Take your OS-9 Master disk out of your drive /d0... put a blank disk in drive /d0... now at the OS-9 prompt type:

```
OS9:format /d0
```

When OS-9 asks if your really want to format the disk in drive /d0 respond Y for yes or R for ready... When the format is done OS-9 will ask you for a name to put on the disk. You may give any name up to 32 characters. The name at this point does not matter as when we backup the original Master disk to this disk... it will retain the name of the original Master disk. So you can just call the disk NEW. When the format is done and the verify pass is done the OS-9 prompt will return At this point you want to type:

```
OS9:free /d0
```

If the free does not say...630 sectors total on the disk... and 620 available for use... *Do not use this disk to do a backup onto... it will not work!!!* Take the newly formatted disk out of drive /d0 and put your original Master disk back in drive /d0. At the OS-9 prompt you will type:

```
OS9:unlink format free
```

This will take the two commands out of memory and give you more memory to work with when you go to do your backup. At the OS-9 prompt you will type:

```
OS9:load backup
```

Take your original Master disk out of drive /d0 and put your newly formatted disk in drive /d0... at the OS-9 prompt you type:

```
OS9:backup s /d0 #32k
```

OS-9 will ask you if you are ready to backup from /d0 to /d0... You type Y for yes. OS-9 will now say ready the destination disk... you already have it in the drive. Hit any key to continue ... OS-9 will list the name that was on the disk and ask if it ok to write over this disk... type Y for yes... OS-9 will then say ready SOURCE disk hit a key... Put your OS-9 Master disk back in drive /d0 and hit any key... When OS-9 says to ready the DESTINATION disk... Take your Master disk out of drive /d0 and put the newly formatted disk in drive /d0... hit any key to continue... OS-9 will repeat this prompting till the whole original disk is copied exactly over to the new disk...

Warning

If the disk that you did the format on did not give you the whole 630 sectors on the disk and 620 free for use when you did the free /d0 on it... you may not backup to that disk... backup is a mirror copy of the original disk to the new disk... if there was a bad sector on he new disk the backup will not work... if there was a bad sector on the original disk the backup will not work... BOTH DISK MUST BE FREE OF ERRORS AND HAVE THE SAME NUMBER OF SECTORS ON THE DISK... A DISK WITH 80 TRACKS CAN'T BE BACKED UP TO A DISK WITH 40 TRACKS... A DOUBLE SIDED DISK CAN'T BE BACKED UP TO A SINGLE SIDED DISK... A HARD DISK CAN'T BE BACKED UP TO A FLOPPY DISK... OS-9 WILL CHECK TO SEE WHAT SIZE THE DISK IS THAT YOU ARE BACKING UP FROM AND TO... IT WILL ABORT THE BACKUP IF THEY ARE NOT THE SAME TYPE/SIZE...

WHEN THE BACKUP IS DONE DO THIS: Put your original Master disk away. You can leave the new Master disk in drive /d0... but you must let OS-9 know that there is a new disk in the drive... so at the OS-9 prompt you will type:

```
OS9:chd /d0
```

```
OS9:chx /d0/cmds
```

OS-9 is now ready to continue... and we can now delete some seldom used commands and gain some disk space back for our own files... To delete these commands you will type:

```
OS9:del /d0/cmds/command.name
```

Where command name is the names of each command you are deleting...

binex
exbin

These two commands are for turning a binary file into a text file and vice/versa. I have never used these two commands to date. While they might be useful, I am not sure who they are useful to?

cmp

This is for comparing two text files together and listing where (with an offset) the differences are. I find it easier/quicker/more reliable to just list the two files to my screen and look for the differences...

cobbler

This is only used for making a new os9boot file on your master disk. You won't need to use this command till later lessons to make a new boot disk. So you can delete it from the /D0/CMDS directory for now.

dcheck

This command does a total search of the disk it is called to check and will report if any files on the disk have been destroyed in some manner. You will only need to use this command on a disk that is used a lot and is almost full. If you ever try to use a command or file and get an error, try the ATTR command on the file/command to see if you have permission to use the file or command. If the ATTR says you do and you still can't get to the file/command then I would use the dcheck on the disk to see if the disk was damaged in some manner. Dcheck is very powerful but it will usually be the case that when you need to use it is when the sh*t has already hit the fan and some of the files on your disk have been damaged in some manner. For this reason you may want to feel safe and keep this command in your /D0/CMDS directory.

display

If you don't have a printer hooked up to your OS-9 system than you probably won't need the display command. It is meant for sending some series of hex codes to a device... if you do this:

```
OS9:display 0c
```

the screen will clear... this is sending a control-l to the screen... if you did this:

```
OS9:display 0c >/p
```

it will send a control-L to your printer which in most cases will do a form feed on your printer. If you have a decent printer you may use the display command with the (>) redirect sign to send hex codes to your printer for setting it for double strike, emphasized, underline, and any other special features your printer might support. As far as being able to send all the codes from 0 to 255 to your screen? I haven't found much use for this. I have only used the display Oc to clear the screen so far.

kill

This command is only used to kill off some multi-process command that you may have started up to run in the background... if you did this:

```
OS9:dir e /d0/cmds >/p&
```

OS-9 will do a dir e of your EXECUTION directory and send it to the printer. The OS-9 prompt will return and the list will continue in the background. If you did a procs e command at this point you will see that there is a process 3 or 4 or 5 running in the background and it is called list. To stop the list from continuing..you would have to type:

```
OS9:kill 3
```

or

```
OS9:kill 4
```

or whatever the process # is that you wish to stop... You won't be doing a lot of multi-processing while you learn OS-9 so you should not need this command in your /D0/CMDS right now...

link

This one is hard to explain right now... let's just say that when you do a load command... the computer does a link for you... so this command should not be really necessary for now. If you do a mdir e and see that a commands link count is say 3 or 4... then you would have to unlink the command 3 or 4 times to get it *out* of memory!!!

login

This is only needed if you are going to hook up your OS-9 system so that outside people can link into your CoCo and use the computer at the same time you are using it... When you delete this command you may also delete the 2 files in the /d0/sys directory called /d0/sys/motd and /d0/sys/password...

merge

This is used for putting two data files into a single data file with a new name... this can also be done with the list command like this:

```
OS9:list data1 data2 >/data3
```

so the merge command is not needed right now...

os9gen

This command is used for making a new boot disk... the new boot disk might have more/less commands that it will load into memory when it starts up. For the time being you won't need this command until the next lesson where I will try to teach you how to make a new and better boot disk...

printerr

When you are first learning OS-9 you will find this command most useful... when you start to learn the error numbers by heart you will be able to delete this command. You can also delete the /d0/sys/errmsg file also.

procs

You will only need this command if you do multi-processing... it will show you what is running in the computer and who is running it and how much of a priority it has... for now you will not use this command often.

setpr

This is for resetting the priority of a multi-process that you have running in the background... you won't need this command until you get into multi-process running.

sleep

This is for making a process stop doing what it was doing for a set period of time... I have never used this command... if used wrong it will seem like you locked up your computer as it counts down its sleep time.

tsmon

This command is used to make OS-9 scan the built in rs232 port for a carrier ready signal... it is used to set the computer up for another caller usually calling over a phone line... it is the basics of setting your OS-9 system up as a bbs.

Read over the commands in your red OS-9 book and decide for yourself how often you might use the commands listed above. If you think you will not need the commands often then delete them on your backup master disk. This will give you more space to store your own files on that disk, and if you are working with as single drive system then this is *very* important... You may also delete the asm command from your /d0/cmds directory if you *never* intend to do any machine language work... If you do delete the /d0/cmds/asm command then you can also get rid of the data files in the defs directory that the asm command works with. Do this by typing:

```
OS9:deldir /d0/defs
```

OS-9 will than say deleting a directory and offer you a list, delete or quit option... if you choose to list... it will show you what is in the directory that you wish to delete... if you choose to delete the directory... you will not be prompted again unless there is another directory found in the directory... This command will take a bit of time to get rid of the directory /d0/defs... but it will clear up a LOTTTTTTT of space for your own use....

OK, I saved you a bit of disk space now let me save you a bit of a headache!!! Radio Shack now has two versions of the OS-9 operating system. These are the 1.00.00 and the 1.01.00 versions. The 1.01.00 has some new stuff added to it but it is basicly the same as 1.00.00 It is not exactly the same... close but not exact. For this reason if you see any articles in say Rainbow mag that say you can change your OS-9 to have 6 ms. step rates on your drives or 40 tracks on your disk... you should be *very* careful that the instructions refer to your version of OS-9. Some of the early articles in Rainbow refer to making changes to OS-9 1.00.00... The most recent articles in Rainbow will usually say that these patches are for 1.00.00 or 1.01.00... With Radio Shack getting ready to come out with OS-9 2.00.00 it is very important you know what OS-9 you have when you read any articles that say how to change your OS-9 to add some new features to it.

If you have the original OS-9 1.00.00 then you can get the OS-9 upgrade from Radio Shack to 1.01.00 for about $15. The upgrade to OS-9 2.00.00 will cost about $25. These upgrades are only available to original owners of OS-9 1.00.00 or 1.01.00.

Let's talk about the devices that are available for you to use under OS-9.

/p

this is for your serial printer

/t1

this is for the built in RS232 port

/t2

this is for the RS232 cartridge

/d0

this is drive 0

/d1

this is drive 1

/d2

this is drive 2

/d3

this is drive 3

/term

this is for your keyboard and video screen

On my system I have a /H0 and /R0 which tell the OS-9 that I have a hard disk drive and a ram disk drive. If you are good at machine language you can write your own code to add your own hardware onto your OS-9 system. I understand that OS-9 2.00.00 has a device called /ssp and it is for the Radio Shack Speech Sound Pak and will let you send a text file through it and it will speak the file out. So you could do this:

```
OS9:dir /d0 >/ssp
```

and you would hear your directory. This could be very useful for anyone with impaired sight. While I have been talking about how you can add devices to your OS-9 system I also need to say that you can delete devices from your system also. This will free up ram for other programs you might wish to run. In OS-9 2.00.00 you can usually get rid of /d2, /d3 and the /t2 drivers. Most people don't have the drive 2 and 3 and don't have the Radio Shack RS232 pak so why keep these modules in memory wasting space? So you can delete them and save some space that is badly need in the 64K CoCo. How do you do this? We'll cover that in our next lesson; Making a new boot disk.

4. Lesson 4

Today we are going to make a new OS-9 boot disk. we can do this several different ways. First type this:

```
OS9:mdir
```

this will show you all the programs and descriptors that are in memory... you should see something like this:

```
OS9      OS9P2     INIT
BOOT     CCDISK    D0
D1       D2        D3
CCIO     TERM      IOMAN
RBF      SCF       SYSGO
CLOCK    SHELL     RS232
T1       PRINTER   P
PIPEMAN  PIPER     PIPE
```

These are all modules that are loaded into memory from the OS9boot file and you can change the OS9boot file so that it will load in more or less of these modules at boot time. For now we will just make a new boot disk that will load in all of the above modules exactly the same way. We can do this one of two ways... First let's format a new disk and we'll put our new OS9boot on it. At the OS-9 prompt type:

```
OS9:load format free
```

take your OS-9 master disk out of drive /d0, and install a blank disk in /d0 now at the OS-9 prompt type:

```
OS9:format /d0
```

when OS-9 asks if you are ready... type R for ready or Y for yes. When OS-9 asks you for a name to put on the disk give it the name NEW BOOT DISK. When the format is done at the OS-9 prompt type:

```
OS9:free /d0
```

and you should get 630 sectors on the disk with 620 available for use. This lets you know that the format was good with no bad sectors that needed to be locked out. Take the new disk out of drive /d0 and put your OS-9 master disk back in. Now type:

```
OS9:load cobbler makdir save os9gen build echo
```

You should now have the commands **format**, **free**, **cobbler**, **makdir**, **build**, **os9gen**, **save** and **echo** in your memory... Take your master disk out of drive /d0 and put the new disk in. Now at the OS-9 prompt type:

```
OS9:cobbler /d0
```

This will put the file OS9boot on the disk in drive /d0 and it will contain all the modules that were loaded into memory from the *last* boot. To get all the other files/commands/directories over onto this new boot disk you need to be very patient because you will have to go through a lot of typing, copying and swapping. Essentially what you have to do is: Use the **makdir** command to make all the directories on this new disk that were on the old Master disk. So you will have to do this:

```
OS9:makdir /d0/sys
OS9:makdir /d0/cmds
OS9:makdir /d0/defs
```

Then you will need to use the **copy** command to move all the files/commands over from the old Master disk to this NEW Master disk. We are talking about a *lot* of disk swapping here and you just might not be up to going through all this work. Read on... there are 2 easier ways to do this and I will let you decide which you like better.

While **cobbler** will help us to make a new boot disk there is no flexibility to it and you are stuck with a mirror image of the modules from the last boot you did. To give us total control of what goes in the `OS9boot` file we will need to use the **OS9gen** command. Leave the new boot disk in drive /d0 and format it again... then do the free on it to be sure the total sectors and free sectors works out to 630 and 620. We will now type:

```
OS9:makdir /d0/modules
OS9:save /d0/modules/ccdisk ccdisk
OS9:save /d0/modules/d0 d0
OS9:save /d0/modules/d1 d1
```

Note: if you only have a 2 drive system you can leave out the next two lines

```
OS9:save /d0/modules/d2 /d2
OS9:save /d0/modules/d3 /d3

OS9:save /d0/modules/ccio ccio
OS9:save /d0/modules/term term
OS9:save /d0/modules/ioman ioman
```

```
OS9:save /d0/modules/rbf rbf
OS9:save /d0/modules/scf scf
OS9:save /d0/modules/sysgo sysgo
OS9:save /d0/modules/clock clock
OS9:save /d0/modules/shell shell
```

Note: if you never intend to let an outside user link with your CoCo by an outside phone line then leave out the next 2 lines

```
OS9:save /d0/modules/rs232 rs232
OS9:save /d0/modules/t1 t1
```

Note: if you don't have a printer you may leave out the next two lines

```
OS9:save /d0/modules/printer printer
OS9:save /d0/modules/p p

OS9:save /d0/modules/pipeman pipeman
OS9:save /d0/modules/piper piper
OS9:save /d0/modules/pipe pipe
```

Note: if you don't have the Radio Shack RS232 pak than you can leave out the next two lines

```
OS9:save /d0/modules/acia acia
OS9:save /d0/modules/t2 t2
```

We have moved an image of the modules that are in memory over to the disk in drive /d0. If you did not have a printer or did not have drives /d2 and /d3 then you did not save the listed modules over to the disk. If you did not intend to have an outside user then you did not save rs232, t1, acia, and t2 over to the disk.

We are now going to build a data file that is going to tell OS9gen what modules it is to put into the OS9boot file that we are going to put on this disk. It is very important that the spellings you used in the **save** command are the same as the modules are spelled in memory... The new OS9boot we are going to make might not work if you spell any of the saved modules names wrong! OK, now at the OS-9 prompt you type:

```
OS9:build /d0/bootlist
```

You will then see a (?) for the prompt. at each (?) prompt type in the following lines... *without the (?) marks!!!*

```
? ccdisk
? d0
? d1
```

Note: if you did not save *d2* and *d3* in the save operation leave the next two lines out!!!

```
? d2
? d3
? ccio
? term
? ioman
? rbf
```

```
? scf
? sysgo
? clock
? shell
```

Note: if you did not save *rs232* and *t1* in the save operation leave the next two lines out!!!

```
? rs232
? t1
```

Note: if you don't have a printer and left *printer* and *p* out of the save operation then leave the next two lines out!!!

```
? printer
? p
? pipeman
? piper
? pipe
```

Note: if you did not save *acia* and *t2* in the save operation leave the next two lines out!!!

```
? acia
? t2
? (enter)
```

We now have everything we need on the disk in drive /d0 to make **os9gen** put an `os9boot` file on that disk... At the OS-9 prompt type:

```
OS9:chd /d0/modules
OS9:os9gen /d0 </d0/bootlist
```

You will then have a working `os9boot` file on the disk in drive /d0. Now if you left out the *d2*, *d3*, *printer*, *p*, *rs232*, *t1*, *acia* and *t2* modules when you did your **save** and when you built your bootlist... those modules will not be in memory when you use this disk to do your next boot... *But don't boot with this disk yet. It has no command directory on it.* You will need to copy all the data files and commands off your master disk to this disk by the same method explained in the start of this tutorial where I talk about the **cobbler** command.

Now I also said earlier that there was 2 other ways to make this boot disk and you have to decide if they suit your taste. The first way is to make the new `OS9boot` file with either **cobbler** or **os9gen** in the same way that I have already explained. Now that the boot file is on that disk... Don't copy the whole Master disk over to this new disk! Instead only do this:

```
OS9:makdir /d0/cmds
```

Copy the commands **setime**, **dir**, **free** and **echo** over to the /d0/cmds directory on this new disk from /d0/cmds on the old disk. Use the **build** command to build a new startup file on this disk...

```
OS9:build /d0/startup
```

and enter these lines in the file:

```
? setime </term
? echo take the disk out of drive 0
```

```
? echo and put your working disk in
? echo drive 0... This is ONLY A BOOT
? echo DISK!!!
? (enter)
```

So now when you wish to BOOT OS9 you can use this disk. After OS-9 boots ok you will take this disk out of /d0 and put in your Master disk. Then type:

```
OS9:chd /d0
OS9:chx /d0/cmds
```

and you will be up and running. This will be your working disk and the other disk will be used each time you wish to BOOT OS-9.

Another way to make a new BOOT disk is to do this: Backup the old master disk to a newly formatted disk. Leave this new disk in /d0. Now type:

```
OS9:chd /d0
OS9:chx /d0/cmds
```

This will let OS-9 know that you have changed disks. Use the **makdir** command to make a new directory called /d0/modules... Follow all the **save** commands listed earlier in this tutorial. Use the **build** command to build a file called `/d0/bootlist`. Type in all the lines as listed earlier in this tutorial. When this is all done, you will use the **del** command to:

```
OS9:del /d0/os9boot
```

at this point you will:

```
OS9:chd /d0/modules
OS9:os9gen /d0 </d0/bootlist
```

and your new BOOT file will go on this disk with more/less modules as you told it to put in the BOOT file. *Both* of these methods will give you a new BOOT disk but they both have their drawbacks. The first method gives you a boot disk but with little else on it in the way of commands. The 2nd way will give you a boot disk that has all your commands on it and all your working files. But the 2nd way to make a new BOOT disk will not work 100% of the time. When you go to **cobbler** or **os9gen** a new os9boot file on a new disk it will write that file out to track 34 of that disk. It *must* have an unbroken number of sectors on track 34 to put this bootfile. If you go to **cobbler** or **os9gen** on a disk that is pretty full the **cobbler** or **os9gen** might fail. *if* you use **os9gen** to make a new os9boot file and it has *less* modules in it then before the 2nd method will work just fine. *But* if the new os9boot file will have *more* modules in it then this 2nd method will not work 100% of the time.

It is for this reason that I suggest you **os9gen** on a disk that only had the /d0/bootlist file on it and the /d0/modules directory on it. You can then copy over to this new disk the few commands you think you will need and after you boot with this disk take it out of /d0 and put in the disk you intend to work with. A disk that has *all* the commands you know you will need!!!

You might also want to add some commands to the os9boot file so that they will be in memory at bootup time. Some very useful commands to have in memory all the time are **dir**, **build**, **del**, **mfree** and **free**. The only disadvantage of having these modules in the os9boot file is that once you boot and these modules are in memory all the unlinking in the world will *not* get them out of memory. So you have to decide if you want them in memory that bad. It should not cause too much of a memory problem if you have left out the *d2*, *d3*, *rs232*, *t1*, *acia*, *t2*, *printer*, and *p* modules.

One of the advantages to making a tailored os9boot file is that it gets rid of modules that you were never going to use and cleans up some RAM for you to use also. Not a lot of RAM but enough to make all this worthwile. The most important thing to

remember though when using the **os9gen** command is that you must move the modules from memory out to a directory where you will put all the modules you wish in the new os9boot file. Then you must build a data file with the names of all the modules you saved; change your data directory to the directory that has all the modules in it; then invoke the **os9gen** command telling it where to put the os9boot file and where it is to get the list of the modules it is to put in the os9boot file.

You may have noticed way back in the beginning that there were some modules in memory that were called:

```
os9   os9p2  init  boot
```

and we did not save them out to the /d0/modules directory and we did not put them in our bootlist. You don't need to. OS-9 knows to put those 4 modules in each new os9boot file it makes. It is something you *should not try to do.*

Right about this time you may be saying to yourself that it sure is a pain to go about making a new boot disk? Well on a single disk drive system it *is*. There is no getting around this. If you had two disk drives you could have formatted the disk in drive /d1. Then used the **cobbler** or **os9gen** command to put os9boot on that disk. You could have then used the **dsave** command to move all the directories/files from /d0 over to /d1 and you would have saved a lot of time and typing. So I now repeat that OS-9 will run on a 1 drive system but it sure runs a lot better on a 2 drive system!

You probably read this whole tutorial and said to yourself that you are *never* going to use cobbler or os9gen to make a new boot disk. Sounds like too much work. Well after reading this all over I tend to agree with you. I had OS-9 for about 1 year before I got around to using os9gen to make a tailored os9boot file. Why did I use it? I saw some fine articles in Rainbow magazine telling how to make my disk drives run at 6ms. under OS-9 and how to make OS-9 use the full 40 tracks that my drives were capable of. It was then I decided how great a command **os9gen** was and learned how to use it.

The OS-9 Boot Process

Alan DeKok

aland@striker.ottawa.on.ca

Revision History

Revision 1 Tue Apr 29 21:53:46 EDT 1997

Revision 2 13-Apr-2003
DragonDOS boot added

Table of Contents

1. COCO Computer

1. Typing 'DOS' at the DECB 'OK' prompt loads in track 34 off of the disk.
2. Track 34 gets loaded into address $2600, and execution of code starts at $2602.
3. $2602 contains a BRA to the execution point of the REL module.
4. REL copies the boot track ($2600 to $3800) to address $ED00, and jumps to another routine inside of REL, at the new address.
5. REL then jumps to OS9p1, which sets up system variables, the system memory map, system call tables, IRQ & SWI setup, and calls BOOT.
6. BOOT reads sector $000000 off of a disk, and finds out where the OS9Boot file is.
7. BOOT requests system memory for the size of OS9Boot, seeks to where OS9Boot is, and loads it directly into RAM.
8. It then returns to OS9p1, after setting up pointers in low memory to the OS9Boot file.
9. OS9p1 links to OS9p2, and executes it.
10. OS9p2 sets up more system calls, links to the clock module, and calls it.
11. Clock sets up some more system calls, starts multitasking, and returns to OS9p2.
12. OS9p2 then does F$Chain of 'CC3Go'. This prints a start up banner, and runs your 'startup' file through a shell.

2. Dragon 64

DragonDOS BOOT (when you type BOOT) loads sectors (numbered from 0) 2-17 (4096 bytes) into RAM at location $2600. The first two bytes of sector 2 must be ASCII 'OS' for this to work. It then jumps to $2602 and begins execution. The boot code switches into RAM mode, and copies the entire section to $F000 and jumps to $F04F.

Arrgh!!!!! My disk crashed, now what do I do?

Dave Gantz

Fixing crashed disks or at least recovering data from crashed disks can be a time consumming ordeal which is why the best recommendation is to ALWAYS keep current backups or DON'T use the disk at all if it is your only copy. However, crashed disks do not have to mean lost data if you know how to go about recovering the data.

Thats what this article is all about and comes as a result of the three crashed disks I've had this past week as well as Chris Spry's current predicament. I have co-written one other article on this topic, but that pertained more specifically to boot disk problems. (See the OS9 Newletter, Volume III Issue 11B, November 30, 1992)

You'll be needing a few tools handy to follow along with this article and if your planning to just practice for the eventuallity of a crashed disk then for God sakes use a backup of something.

----------Tool List----------

OS9 Level II Manual, the big thick one that comes with the OS9 Level II disks.

DED -- Copyright 1987 by Doug DeMartinis -- preferably the edition below

```
Header for:  dEd                 This edition has a patch made to it
Module size: $17A2    #6050      so that it will recognize and
Module CRC:  $299A3F (Good)      identify the Bit Allocation Map or
Hdr parity:  $8F                 BAM for short.  The BAM is also
Exec. off:   $0665    #1637      sometimes called DAM or Disk
Data Size:   $0316    #790       Allocation Map....
Edition:     $05      #5
Ty/La At/Rv: $11 $82             Besides its the one I use.  You could
Prog mod, 6809 obj, re-en, R/O   also use Qtip, but displays will differ
```

A screen dump utility of some sort would also come in handy for this exercise in avoiding futility. See the end of this article for suggested files and source(s).

Now to dive in there and rescue Data <Grin>.....

The first thing will cover is the breakdown of Logical Sector Number zero on any OS9 disk, as well as the invocation of DED. All numbers with a preceding $ are in hexadecimal (base 16) and others will be in decimal (base 10).

To invoke DED for this excercise type the following from any OS9 prompt on any 80x24 or 80x25 text screen or graphics screen with stdfonts merged.

```
OS9: DED /Dx@          (where x = the drive number with the disk
                                  to be worked on in it.)
```

> **Note:** The @ in the above command allows us to open any disk just as if it were a file by itself, thus allowing us to work with any and all data the disk contains with the exception of high density disks in most cases.

Here is an example of my LSN $00. Offsets (Relative Addresses) are read as LSN+left most column+top row. See the ** in the last row? This would be read as LSN or $00+column or $70+top row or $04 or a total offset of $0074. If we were on sector 1 then it would be $0174. The definitions of the important bytes follow the excerpt. Also see page 5-2 in the technical reference section of the OS9 Level II manual.

```
------------------------------------------------------------------------
LSN=$00  00

      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F    0 2 4 6 8 A C E
00:  00 0B D0 12 01 7A 00 01 00 00 03 00 00 FF D4 E3    ..P..z.......  Tc
10:  07 00 12 00 00 00 00 00 00 00 5D 03 08 10 2D 52    ..........]...-R
20:  69 42 42 53 20 43 6F 6D 6D 61 6E 64 73 20 44 69    iBBS Commands Di
```

```
30:  73 EB 00 00 00 00 00 00 00 00 00 00 00 00 01    sk..............
40:  01 03 21 03 00 54 02 00 00 12 00 12 03 09 00 61    ..!..T.........a
50:  40 00 00 00 00 00 00 00 00 00 00 00 00 74 2D 00    @............t-.
60:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
70:  00 00 00 00 ** 00 00 00 00 00 00 00 00 00 00 00    ................
```

Name	Relative Address	Size (Bytes)	Use or Function	Mine
DD.TOT	$00	3	Number of sectors on disk.	$000BD0
DD.TKS	$03	1	Track size in sectors.	$12
DD.MAP	$04	2	Number of bytes in allocation map.	$017A
DD.BIT	$06	2	Number of sectors per cluster	$0001
DD.DIR	$08	3	Starting Sector of of Root Dir	$000003
DD.OWN	$0B	2	Owners ID number (usually 0)	$0000
DD.ATT	$0D	1	Disk attributes	$FF
DD.DSK	$0E	2	Disk identification (internal use)	$D4E3
DD.FMT	$10	1	Disk Format, bit mapped	$07
DD.SPT	$11	2	Number of sectors per track	$0012
DD.RES	$13	2	Reserved for future use	$00
DD.BT	$15	3	Starting sector of bootstrap file	$000000
DD.BSZ	$18	2	Size of bootstrap file	$0000
DD.DAT	$1A	5	Date of creation (Y:M:D:H:M)	$5D0308102D
DD.NAM	$1F	32	Disk name, last char has MSB set	see above
DD.OPT	$3F		Path descriptor options	

Probably the most important byte to us here are the bytes at offsets $08, $09, and $0A which tell us where the root directory begins. Speaking of which, that is our next stop in the CoCo Zone....

```
LSN=$03  03

      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F    0 2 4 6 8 A C E
00:  BF 00 00 5D 04 1A 0D 0A 02 00 00 01 20 00 00 00    ?..]........ ...
10:  00 00 04 00 07 00 00 00 00 00 00 00 00 00 00 00    ................
20:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
30:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
40:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
50:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
60:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
70:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
80:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
90:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
A0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
B0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
C0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
D0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
E0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
F0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

Well here we are exactly where LSN $00 said the root directory starts, at LSN $000003. But where are the filenames, you ask? Well they start on the next sector.

This sector is called a File Descriptor sector or FD for short. Every file or directory on an OS9 disk has one of these. This is why you can't store a true 360K worth of files and user data on a DSDD 40 track drive for example.

For our purpose I'm going to skip the explanation of the first 16 bytes and get on with what we need from this sector to start finding data.

Starting with offset $10 ($0310) is what is called a segment list. This segment list tells OS9 where a file or directory on disk is located and how many sectors that file or directory occupies. There are 48 of these segments avaiable each being 5 bytes wide. For you programmers, think of it as a two dimensional array such as: DIM segment(48,5). What this means is that your file or directory can occupy space in 48 different locations on disk if it is badly fragmented.

In this case mine only occupies one segment starting at LSN $0400 and is 7 sectors in size. So guess where our trip through the OS9 disk takes us next? If you said sector 4 or offset $0400 your right!

```
LSN=$04  04

      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F    0 2 4 6 8 A C E
00:  2E AE 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
10:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03    ................
20:  AE 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

```
30:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03    ................
40:  43 4D 44 D3 00 00 00 00 00 00 00 00 00 00 00 00    CMDS............
50:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0B    ................
60:  53 59 D3 00 00 00 00 00 00 00 00 00 00 00 00 00    SYS.............
70:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 07 0A    ................
80:  72 69 62 62 73 2E 63 66 E7 00 00 00 00 00 00 00    ribbs.cfg.......
90:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 07 2F    .............../
A0:  72 69 62 62 73 67 EF 00 00 00 00 00 00 00 00 00    ribbsgo.........
B0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 AC    ................,
C0:  4D 45 4E 55 D3 00 00 00 00 00 00 00 00 00 00 00    MENUS...........
D0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 07 42    ...............B
E0:  4C 4F 47 D3 00 00 00 00 00 00 00 00 00 00 00 00    LOGS............
F0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 09 A3    ...............#
```

Well here we are, finally. The root directory. Again for our purposes I'm going to skip the first 32 bytes of this sector.

Each entry for each file or directory is composed of 32 bytes. 29 of them represent the file or directory name while the last 3 tell where to find that individual files or directories FD is located on the disk. Looking at this perhaps you can see the importance of having your directory names in ALL UPPERCASE and your file names in all lowercase.

In this example I have 4 directories (CMDS, SYS, MENUS, and LOGS) and two files (ribbs.cfg and ribbsgo). Lets start with a file, ribbsgo in this case.

Its entry starts at offset $A0 ($04A0) and ends at $BF ($04BF). The first 29 bytes as I said are for the file name, the last character of which has its Most Significant Bit set to mark the end of the file name. The last 3 bytes tell us where to find the FD for ribbsgo which is $0002AC or $02AC since the Most Significant Byte is 0.

So this is where we are off to next, sector $02AC.

```
LSN=$2AC  684

      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F    0 2 4 6 8 A C E
00:  0B 00 00 5D 04 19 0C 11 01 00 00 04 A5 5D 04 19    ...].........%]..
10:  00 02 AD 00 05 00 00 00 00 00 00 00 00 00 00 00    ..-.............
20:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
30:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
40:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
50:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
60:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
70:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
80:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
90:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
A0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
B0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
C0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
D0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
E0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
F0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

Ok here we are. This FD is very similar to the one we examined on our way to the root directory. It contains all the same information and takes on exactly the same format as the FD for the root directory except that this time we are talking about a file and not a directory.

It tells us that our file, ribbsgo, begins at sector $0002AD or $02AD and occupies 5 sectors. So that is where we will go next. For the purposes I will only include the first and last sectors in this text as examples. I forgot to mention that we have proceeded through pages 5-3, 5-4, and most of 5-5 of the technical reference section at this point.

```
LSN=$2AD  685

      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F    0 2 4 6 8 A C E
00:  6F 6E 65 72 72 20 67 6F 74 6F 20 66 61 74 61 6C    onerr goto fatal
10:  65 72 72 6F 72 0D 63 64 20 2F 64 64 0D 63 78 20    error.cd /dd.cx
20:  2F 64 64 2F 63 6D 64 73 0D 76 61 72 2E 30 3D 22    /dd/cmds.var.0="
30:  22 0D 64 69 73 70 6C 61 79 20 30 43 20 30 32 20    ".display 0C 02
40:  33 34 20 32 32 20 31 42 20 33 32 20 30 33 20 30    34 22 1B 32 03 0
50:  35 20 32 30 0D 65 63 68 6F 20 50 6C 65 61 73 65    5 20.echo Please
60:  20 49 6E 73 65 72 74 20 79 6F 75 72 20 4F 53 39     Insert your OS9
70:  20 42 6F 6F 74 20 44 69 73 6B 20 69 6E 20 2F 44     Boot Disk in /D
80:  30 2E 0D 64 69 73 70 6C 61 79 20 30 32 20 34 45    0..display 02 4E
90:  20 32 45 20 31 42 20 32 32 20 30 31 20 31 41 20     2E 1B 22 01 1A
```

```
A0:  31 30 20 31 39 20 30 33 20 30 30 20 30 31 20 30    10 19 03 00 01 0
B0:  31 20 30 32 20 32 36 20 32 32 0D 65 63 68 6F 20    1 02 26 22.echo
C0:  50 72 65 73 73 20 41 6E 79 20 4B 65 79 0D 76 61    Press Any Key.va
D0:  72 2E 30 0D 2A 6E 6F 6B 65 79 70 72 65 73 73 0D    r.0.*nokeypress.
E0:  69 66 20 25 30 3D 22 22 0D 67 6F 74 6F 20 6E 6F    if %0="".goto no
F0:  6B 65 79 70 72 65 73 73 0D 65 6E 64 69 66 0D 64    keypress.endif.d
```

Well we made it! The actual file data. There are no special codes or anything of that nature here to explain. Just the ASCII codes for the contents of the ribbsgo script file. With program modules it would be the hexadecial representations of the commands and variables and such within the program.

As I said there are 5 consecutive sectors (or 1 segment) that this file occupies but I will only include this and the last sector, because everything in between is technically the same.

```
LSN=$2B1  689

      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F    0 2 4 6 8 A C E
00:  6F 63 6D 64 73 0D 67 6F 74 6F 20 2B 65 78 69 74    ocmds.goto +exit
10:  0D 2A 66 61 74 61 6C 65 72 72 6F 72 0D 65 63 68    .*fatalerror.ech
20:  6F 20 45 72 72 6F 72 20 25 2A 20 69 6E 20 52 69    o Error %* in Ri
30:  42 42 53 47 6F 2E 20 20 46 69 78 20 61 6E 64 20    BBSGo.  Fix and
40:  74 72 79 20 61 67 61 69 6E 0D 67 6F 74 6F 20 2B    try again.goto +
50:  65 78 69 74 0D 2A 66 69 6E 69 73 68 75 70 0D 64    exit.*finishup.d
60:  69 73 70 6C 61 79 20 31 62 20 32 33 0D 72 69 62    isplay 1b 23.rib
70:  62 73 6D 61 69 6E 20 23 31 36 4B 20 3C 3E 3E 3E    bsmain #16K <>>>
80:  2F 77 37 26 0D 2A 65 78 69 74 0D 64 69 73 70 6C    /w7&.*exit.displ
90:  61 79 20 31 62 20 33 32 20 30 30 20 30 35 20 32    ay 1b 32 00 05 2
A0:  31 20 30 43 0D 00 00 00 00 00 00 00 00 00 00 00    1 0C............
B0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
C0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
D0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
E0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
F0:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

Ok this is the last sector of the ribbsgo file. The important thing to mention here is that this file does not contain essential information throughout the entire sector. The file ends with a carriage return ($0D) at offset $A4 or $355 taking into account for the sector we are on. See all the $00's following that carriage return? We don't need them. I'll explain how to get rid of them later, for now its enough for you to know that there not needed. In some cases those extra bytes may contain $E5's which is the value that OS9 writes to each sector when you format a disk.

Now that you have a basic understanding of how an OS9 disk is put together to become an effective storage medium we can go on and discuss how we are gonna go about recovering data.

```
   Up/Down Arrows  Read & display Next/Previous sector
 <CR> Clean up the screen display
   *  Restart
   $  Fork a SHELL (Ctrl-BREAK to return)
*  A  Append displayed sector to output file
*  C  Close output file
   D  Diddle (adjust) file length
   E  Edit the displayed sector
   F  Find a byte or text string (BREAK aborts)
   H  Help screen (also use '?')
   L  Link to a module - List all modules
*  N  Next occurrence of byte(s) or string (Find)
*  O  Open a file for output (use with Append)
   P  Push current sector onto stack
   Q  Quit dEd - Exit to OS9
   R  Remove and display a sector from stack
*  S  Skip to given sector (sector # in hex)
   U  Unlink from module
   V  Verify all modules in file
   W  Write the sector back to the disk
   X  eXpert mode toggle on/off
   Z  Zap (fill in) the sector displayed
```

What you see above is the built in help from DED. The options we will be using most often are the starred options above.

* S Skip to given sector (sector # in hex)

This option will let us skip to the sector(s) that we have identified from the file desciptors (FD's) and will speed things up considerably.

* O Open a file for output (use with Append)

Once we have found the first sector of the data we wish to recover we can use this option to open a path to another disk (or RAM disk) on which we will store the recovered data. Since we will have to do some editing on the recovered file a RAM disk is recommended.

* A Append displayed sector to output file

Once we have opened the destination file for the data we are trying to recover this option will let us add the current sector to that new file. You use this until you either reach the end of that particular segment (Another FD will most likely be displayed at the end of a segment or file) or the end of the file.

```
   Up/Down Arrows  Read & display Next/Previous sector
 <CR> Clean up the screen display
   *  Restart
   $  Fork a SHELL (Ctrl-BREAK to return)
*  A  Append displayed sector to output file
*  C  Close output file
   D  Diddle (adjust) file length
   E  Edit the displayed sector
*  F  Find a byte or text string (BREAK aborts)
   H  Help screen (also use '?')
   L  Link to a module - List all modules
*  N  Next occurrence of byte(s) or string (Find)
*  O  Open a file for output (use with Append)
   P  Push current sector onto stack
   Q  Quit dEd - Exit to OS9
   R  Remove and display a sector from stack
*  S  Skip to given sector (sector # in hex)
   U  Unlink from module
   V  Verify all modules in file
   W  Write the sector back to the disk
   X  eXpert mode toggle on/off
   Z  Zap (fill in) the sector displayed
------------------------------------------------------------------------
```

* C Close output file

Now that we have recovered the data or file we must close the file before doing anything else with it.

* F Find a byte or text string (BREAK aborts)

* N Next occurrence of byte(s) or string (Find)

If you know specific words or byte sequences to look for within the data or file your trying to recover then these two are handy for locating those words or sequences.

Well we've recovered a file or data. There is, if you recall, quite likely some extra unwanted bytes. What do we do to get rid of them? Thats easy, again using DED (and ident for program modules) we diddle with the file length. Now you won't be dealing with real sector numbers, just the relative sector offset from the beginning of the file. In this case it will read LSN $00 thru $04 although we may not actually be on sectors 0-4.

At any rate you need to find the last relative sector of the file probably using the arrow keys to scroll through it. When you reach the last sector look and the LSN, left column number, and top row number and determine the offset for the last byte (the

carriage return) and add 1. In this example that last byte will be at $04A4 then add 1 giving us $04A5.

Hit D for Diddle with file length. It will tell you the old length and ask for the new length. Type it in ($04A5 for this example) and press enter. You will see the extra bytes disappear in front of you.

Now hit Q to quit and answer 'Y' and you have just recovered your first file. Give yourself a pat on the back, get a cup of coffee and dig in cause your gonna be dancin on the keyboard for several hours to completely recover one DSDD disk.

It took me roughly 24 hours to recover all data from 3-3 1/2" 756K floppies (I have mine formatted for 84 tracks double side rather than the usual 80 tracks double side <Evil Grin>).

For some disks your directories will get trashed and there is little one can do to recover the directories (that I know of) in which case you will have to sit there with the arrow keys in DED identifying FD's and locating the 'lost' files. This is what took me so long, my directories got trashed.

This is, as I said, a time consumming method but I know of no program that will do it for you. If I ever get some of my other programming projects finished I intend to write something, but for now this method will have to do.

Good luck recovering that lost data!

dEd

Disk Editor

Doug DeMartinis

dEd is a screen-oriented disk editor utility written in assembly language. It was originally conceived as a floppy disk editor, so the display is organized around individual sectors. It performs most of the functions of Patch, from Computerware, but is faster, more compact, and screen-oriented rather than line-oriented. Individual files or the disk itself (hard, floppy, ram) can be examined and changed, sectors can be written to an output file, and executable modules can be located, linked to and verified. With simple changes, it will run on any CoCo Level I OS-9 and possibly others (CoCo Level II OS-9).

To use, type:

```
dEd <pathlist>
```

where <pathlist> is of the form: filename or dirname or /path/filename or /D0@ (edits entire disk)

dEd will read in and display the first 256 bytes in the file (disk). This is Logical Sector Number (LSN) zero. You move through the file sector (LSN) by sector using the up and down arrow keys. The current LSN number is displayed in Hex and Decimal in the upper left corner of the screen. If the disk itself was accessed (by appending '@' to it's name when dEd was called), the LSN is the disk sector number. If an individual file is being editted, however, the LSN displayed refers to the file, not to the disk. All numbers requested by dEd must be in Hex format. All commands are accessed by simply pressing the desired key.

Table 1. Commands

Up/Down Arrows — Display Next/Previous Sector (LSN)

Each keypress moves the display to the next or previous 256 bytes. Auto-repeat allows skimming quickly through the file. To halt the key-repeat, type Control-W to pause the display, then hit any key. The LSN displayed represents the most-significant byte of the offset of the bytes from the start of the file (module), so byte number $1457 would be found in LSN $14 on row 50, column 7.

A Command — APPEND Current LSN to Output File

This command writes the sector currently displayed to the file opened with the O command. Append is inactive unless an Output file has been created. This mode is useful for recovering files with unreadable sectors in them, as all the sectors before and after the crashed sector may be accessed and saved to a new file. Append also increments the display to the next LSN automatically to speed this process. At the end of a file, generally fewer than 256 bytes will be displayed, as the length of most files is not an even multiple of 256. The Append command will only write out the bytes that are displayed, and the display will remain on the same LSN. Pressing 'A' twice on the last LSN of a file results in it being written to the output file twice.

C Command — CLOSE Output File

This command closes the file opened with the O command and removes the file name from the display, making Append inactive.

D Command — Diddle with the File Length

This command displays the current file length, in Hex, then allows you to change it. This is potentially very dangerous (e.g. if you use it on loadable modules). If you just press <ENTER> at the prompt, you will be returned to the command mode (this is useful for just checking the file length). If you enter a valid length (number of BYTES, not sectors, in Hex), the file will be expanded or contracted to that length. This is useful for stripping the Control-Z's ($1A) off the end of files downloaded with the XModem protocol. Remember, the LENGTH of a file is 1 greater than the number of the last byte in the file (remember, counting from 0!)

E Command — EDIT the displayed Sector

This is the heart of dEd. The cursor will appear over the first byte or character in the LSN. If it's over the BYTE, you can change the value by typing 2 new nibbles in Hex, e.g. 6c. The display will be updated and the cursor will move to the next BYTE. If the cursor is over the CHARACTER part of the display, you can change the value by typing a new ASCII character. Again, the display will be updated and the cursor will shift to the next character. You switch between the BYTE and CHAR modes at any time by hitting the <BREAK> key, as noted at the bottom of the display. You navigate through the sector to individual BYTEs or CHARs using the 4 Arrow keys. The Right and Left Arrows wrap the cursor around to the next or previous row on the display. The Up and Down Arrows wrap around from top to bottom and vice-versa. Once you are done Editting, pressing <ENTER> will exit the Edit mode, as noted at the bottom of the display. As with Zap, the sector is NOT written back to the disk unless the Write command is then used (unless in Expert mode).

F Command — FIND

This searches the file for a given character or byte string You will be prompted with 'Find byte string $'. Enter a series of up to 16 Hex bytes, without spaces (you MUST enter leading zeroes for the numbers 00 - $0F) then press <ENTER>. The search begins at the start of the LSN currently displayed. If a string is located that EXACTLY matches the string you input, the LSN in which it is located will be displayed, with the first byte/character in the string highlighted. By pressing the <BREAK> key at the prompt, you can toggle between the BYTE and CHARACTER search modes. In the character mode, the prompt is 'Find char string: '. Enter up to 16 ASCII characters, then press <ENTER>. In this mode, dEd will locate any string that matches the one you input, regardless of the Upper/Lower case status of either string. As well, characters with bit 7 set (e.g. file names in directories) are treated as if bit 7 was clear. If no matching string is found, you are returned to the command prompt. If you enter an invalid character or byte string, a beep sounds and the same prompt is re-issued. If you just press <ENTER> at either prompt, you will be returned to the command mode.
If you wish to abort a search in progress, just press the <BREAK> key. This will simulate a "string not found" and return you to the CMD: prompt.

H (or '?') Command — HELP

This displays a Help screen.

L Command — LINK to a Module/LIST Modules

If you are editting a file that consists of various executable modules merged together (e.g. OS9Boot), this command allows you to 'Link' to one of the modules. It will be treated as if it is an individual file, i.e. the start of the module will be displayed as LSN 0 and only THAT MODULE will be accessible for display and/or Editting. The LSN displayed is referenced to the MODULE, not to the main file or the disk. You will be prompted with 'Link to which Module? '. You may enter a valid name and press <ENTER>. If that module can be located AND if it's header information is correct, it will be 'Linked'. The top row of the display reflects this by displaying the module name and it's offset, in bytes, from the beginning of the main file. At the 'Link to which Module? ' prompt, if you just hit <ENTER>, all the valid modules and their offsets from the beginning of the main file as well as their lengths, in bytes Hex, will be Listed. Hitting any key after this redisplays the current LSN. The Link mode is useful for changing a given byte in a module using the offset one would use for Debug. For example, to permanently change the printer baud rate, you would call 'dEd /D0/OS9Boot'. Then hit 'L' for Link. At the prompt 'Link to which Module? ', Type P and hit <ENTER>. If P is in your OS9Boot file, the top row of the display will have MODULE: P and give the offset of it from the beginning of the OS9Boot file. Enter the Edit mode by hitting 'E' then use the arrows to move the cursor to byte number 27 (row 20, column 7). Type in the new value for the baud rate then hit <ENTER> to exit Edit mode. Hit 'W' to write the sector, then 'V' to verify the modules. If a module is linked, the 'Find' command only searches for strings in that module.

N Command — Find Next occurrence of String

This is used in conjunction with the 'Find' command. Once a Hex byte or ASCII character string has been located with 'F', the next occurrence may be located by pressing 'N'. This search starts where the 'Find' search left off, IF the LSN hasn't changed since the string was initially located. If the LSN has changed, the search starts at the beginning of the current LSN. If the search is successful, the string will be highlighted, as with 'Find', otherwise you are returned to the command prompt, with a beep.

O Command — OPEN an Output File

You will be prompted with 'OUTFILE: '. Type in a file name or path (that does not already exist) and press <ENTER>. A new file will be created and opened, with the name displayed on the second row of the screen. Individual sectors can be written to this file using the Append command.
To abort the O Command, just press <ENTER> or <BREAK>.

P Command — Push an LSN onto the Sector Stack

This pushes the current LSN onto a funtional Stack of sectors. The 'R' command then can remove (pull) them from the Stack in the reverse order (Last In, First Out) that they were pushed. Up to 16 LSN's can be saved on the Stack. This is very useful when trying to reconstruct a deleted file that was fragmented on the disk. By pushing each sector examined onto the Stack, you can retrace your steps backwards without having to remember the sector number of each sector along the way.

Q Command — QUIT dEd

This exits dEd immediately. You should be sure to Write any Editted or Zapped sectors back to the disk before Quitting.

R Command — Remove an LSN from Sector Stack and Display

This removes (pulls), from the Sector Stack, the last LSN that was pushed (with the 'P' command) and displays it.

S Command — SKIP to a given LSN

You will be prompted with 'LSN=$ '. Type a sector number (in Hex) and press <ENTER>. That LSN will then be displayed, if possible. If the LSN entered is greater than the last LSN in the file, nothing will happen.

U Command — Unlink a Module

This 'unlinks' a module that has previously been 'linked' using the 'L' command. The first sector in the file (LSN 0) is displayed. Using this command when no module has been linked has no effect.

V Command — VERIFY All Modules

This command calculates and writes a new CRC value for EACH module in a fale. It must be used after Editting executable modules or they will not be loadable. Verify is terminated if an error is located in the header of a module, but each module is verified individually and the CRC is rewritten to each before moving to the next module in a file, so modules in front of one with an error will be verified.

W Command — WRITE the Sector displayed to the Disk

WARNING!!! This command can be very dangerous to the well- being of your disk data. It writes the sector displayed back to the disk (at the same location from whence it came, but not necessarily with the same information, hence the danger). After Zapping or Editting a sector, you must use this command to make the changes on the disk (unless you're in the Expert mode). Because you can effectively maim your data with this command, you will be prompted with 'Are you sure (Y/N)? '. If you hit any key other than 'Y' (or 'y') the Write command will be aborted. Otherwise the sector will be written, with the display advising you that dEd is 'Writing Sector...'. On a hard disk or RAM disk, this is very quick.

X Command — EXPERT Mode

This command is potentially the most dangerous of all. It should be used only by those who are very brave (fools?) or those who never make mistakes. In this mode, any sectors changed by Edit or Zap will be automatically written to the disk. The Write command is not needed, and is inactivated in the Expert mode. Any errors made during Edit or Zap WILL be transferred to the disk, making this mode very good for crashing directories, etc. Having noted this, there is one route of escape from errors made in this mode. After Editting or Zapping a sector, the sector is not actually written back to the disk until after Edit or Zap is exitted and the next command is issued (i.e. the next Command Key is pressed). If you hit Shift-BREAK , before hitting any other Command Key, the automatic Write will not occur. You must type Shift-BREAK before any other command or the sector will be written to the disk. This is a one-time escape, so any further errors made will require hitting the Shift-BREAK key again at the appropriate time to avoid writing the bad sector to the disk. Unless you fully understand the OS-9 disk structure (or enjoy toying with the life of your data), this mode should probably be avoided. It is entered after responding with a 'Y' or 'y' to the 'Are you sure (Y/N)?' prompt, and the display advises you that you are in this mode. The 'X' command is a toggle switch, so to exit Expert mode, just hit 'X' again and the 'Expert Mode' message will be erased.

Z Command — ZAP the displayed Sector

You will be prompted with 'Zap Byte: ' or 'Zap Char: '. Enter either a Hex byte (e.g. 6c) or an ASCII character (e.g. $) and press <ENTER>. That byte or character will be written to the ENTIRE SECTOR. You can toggle between the BYTE/CHAR modes by hitting the <BREAK> key. In the CHAR mode, hitting ALT-Char then <ENTER> will Zap the sector with that char with bit 7 set. Unless you're in the Expert mode, only the buffer will be Zapped with the Zap command. To put this Zapped sector on the disk, you must use the Write command. If you decide not to Zap the sector, just hit <ENTER> without first entering a byte or character. You can redisplay a sector that has been Zapped, but not Written back to the disk, by going to the next LSN using the Up Arrow, then returning to the original LSN with the Down Arrow (again, provided you're not in the Expert mode).

$ Command — Fork a SHELL

A new Shell is forked to allow access to OS-9 without terminating dEd. To return to dEd, press Ctrl-BREAK (ESC).

<CR> Command — Clean up screen display

Pressing <ENTER> at the CMD: prompt will clear and re-write the screen which may have been trashed by an error. Current editing to the displayed sector is not lost or written. This command only resets the screen display.

Software for OS9

Jon Bird

Table of Contents

There has been a fair deal of material written on using OS9, applications and filters but very little that I have seen (in the Dragon world anyway) about actually writing software for OS9. I think most people only tend to use it for Stylograph and such like, because it is a completly different environent to the Dragon in it's native mode and can to be honest pretty horrendous to get to grips with on a normal 64K Dragon. Originally I intended to cover this as part of the stuff I've been writing on connecting PC's/Dragons but it's such a big subject I thought breaking it out separatly would be better. Therefore, I'll also cover device drivers and stuff like that in support of the PC thing, because OS9 gets a lot better and easier to use by the addition of say an 80 column screen and hard disk.

1. DragonDOS vs OS9

Trying to explain the difference between a Dragon running normal BASIC/DOS and one running OS9 is difficult if you have not used one or the other. However, for the programmer the Dragon is ideal, because you get the best of both worlds, in 'native' mode where you can program and do exactly what you want with the machine and under OS9 where things are a lot more tightly managed by the operating system which is a lot more how it is becoming with Windows and such like.

Essentially, OS9 is composed of a set of 'building blocks' and can easily be expanded by adding or changing another building block or module. A completly minmimal setup would therefore consist of the OS9 kernel (called OS9p1 & OS9p2 on a Dragon) which is just a block of core procedures required to manage the system. It has no idea of keyboards, disks or screens or any input/output come to that and does not even come with any sort of command interpretor (the shell utility performs this which is a seperate entity in its own right). In fact on its own it is pretty much useless on an ordinary home computer. However, OS9 has found its home (as many of the books on it will tell you) in lots of real time applications, where there is no need for direct human interaction. However, for most of use some sort of interaction is reqired so you will find a dedicated module for handling all I/O (called the IO Manager (or IOMAN for short)) and further modules for dealing with specific types of IO. As an example, standard OS9 is shipped with two file managers, the Sequential File Manager (SCFMan) which manages all sequential types of devices, such as the keyboard, screen and knows all about sequential type peripherals, such as delete keys, carriage return and other line editing functions. The other file manager is the Random Block File Manager (RBFMan) which looks after random block type devices, such as disks. This knows all about what an OS9 disks looks like, what makes up a file and how to find it. Finally, there are the drivers, which actually talk to the hardware, and on the Dragon are specifically designed for the Dragon's hardware. For example, DDISK the Dragon device driver, knows how to talk to the disk controller chip to read a bit of the disk. On the other hand, it does not know what a file is. Right at the bottom of

the change are the device descriptors, which are basicly blocks of memory to tell the system, this is disk drive 0, it uses DDISK to talk to the disk controller, and RBFMan is it's file manager. A common data flow would therefore look something like:

```
Your OS9 Program says:
  'read 100 bytes from path number n'

Which is passed to IOMAN as:
  'read 100 bytes from path number n'
which determines that RBFMan is responsible for path n:

Which is passed to RBFMan as:
 'read 100 bytes from path number n'
who knows all about what path n is, and where to get those
100 bytes from:

Which is then passed to DDISk as:
 'read sector m'
which actually gets the disk controller to read the data
```

This collection of modules forms, in equivalent Dragon terms the BASIC & DOS ROM starting at 32768 ($8000) onwards. On its own, they provide little more than procedures and subroutines for using your machine.

In order to utilise these routines, and actually get your machine doing some useful work, some sort of user friendly interface is needed to talk to the user, under OS9 a program called Shell is provided to do this, and the BASIC interpretor provides this facility under normal DragonDOS mode. They both work in a similar manner: each has a set of internal commands which it understands and can execute, and each has the ability to run external programs. For example, under Dragon BASIC the line:

```
PRINT "HELLO WORLD"
```

is effectivly running at internal program within the interpretor called PRINT. Likewise with the OS9 Shell:

```
chd /d0
```

is understood by the Shell to mean 'change directory to drive D0' and runs the internal code required to perform this operation.

In order to run a program external to the BASIC interpretor, you need to tell it explicitly, as in:

```
RUN "MYPROG"
```

Tells it that MYPROG.BAS is not part of the on board ROM, and that it has to go away and fetch it from somewhere else (in this case the disk). The Shell works slightly differently, in that any command you type which is not recognized by the Shell itself means that is will go away and attempt to find it externally. For example:

```
dir /d1
```

The DIR command is not recognized by the Shell, therefore it automatically attempts to load and run it from elsewhere. The Shell itself contains relativly few in-built com-

mands (unlike the BASIC interpretor) and so for the majority of things will be forcing the execution of an external program.

One of the utility programs shipped with Dragon OS9 is MDIR which simply provides a list of things in memory, or a very simplified OS9 memory map. When I run it on my system I get something like the following:

```
 Module Directory at 10:47:24
ADDR SIZE TY RV AT UC   NAME
---- ---- -- -- -- -- ---------
F05E  7E7 C1  1 r     OS9
F853  4FC C1  1 r   1 OS9p2
FD4F   2E C0  1 r   1 Init
FD7D  182 C1  1 r   1 BooT
BF00  122 C1  4 r   1 SYSGO
C022  101 C1  1 r   1 Clock
C123  6E7 C1  1 r   1 IOMan
C80A  CF5 D1  1 r   1 Rbf
D4FF  40E D1  1 r   2 Scf
D90D  1FF D1  1 r     PipeMan
DB0C  5E2 E1  7 r     Ddisk
E0EE  3AF E1  6 r   2 Kbvdio
E49D   8D E1  1 r     Printer
E52A   28 E1  1 r     Piper
E552   2E F1  1 r     D0
E580   2E F1  2 r     D1
E5AE   3C F1  1 r   2 Term
E5EA   3A F1  1 r     P
E624   26 F1  1 r     Pipe
E64A  4FA 11  1 r   2 Shell
EB44  121 E1  3 r   1 Rdisk
EC65   2E F1  1 r   1 R0
EC93  160 E1  1 r     PC_Disk
EDF3   30 F1  1 r     H0
5A00  1A6 11  1 r   1 Mdir
```

The majority of these modules are the drivers, managers and general routines that make up the 'core' of the OS9 system. In fact, the only programs as such are the Shell, and the MDIR program itself. One of the things to note, is that if you run MDIR on your OS9 system, it would almost certainly look different: not only would you probably have different drivers, they would almost certainly appear in different areas of memory. This is one of the key differences between OS9 and normal Dragon BASIC mode, that any program written for OS9 does not know where it will be in memory and therefore has to be written to be run from any location in memory. One of the side effects of this is that code modules in OS9 will probably tend to execute slower that a DragonDOS counterpart. Part of this is also due to the fact that within the Dragon BASIC/DOS ROM routines always lie at fixed addresses, and can be called quickly. OS9 programs have to perform more complex indexing operations to find routines. Therefore, OS9 always seems considerably more slower and therefore more painful to work with than DragonDOS.

To see this, and some of the pros and cons of writing programs for each system, here is a couple of 'hello world' routines for each system. Firstly, the Dragon assembler version - this assumes DASM to be the assembler:

```
10 EXEC &H6C00
20 ALL
30 @MSG FCC 0,"HELLO WORLD",0
40 @START LDX #@MSG
50 JSR $90E5
60 RTS
70 END @START
80 EXEC
```

It is a fairly simple piece of code, and also fairly easy to write, assemble and get running. Firstly, DASM is loaded from disk, you write the code much like a BASIC program using all the normal BASIC editor functions then just run it. The program itself makes use of a standard ROM routine, which prints a string pointed to by the X register. The 'END' statement at line 70 puts the starting address into the default EXEC location, which is then called by line 80.

Onto the OS9 version then:

```
 nam WORLD
 ttl Hello World program

 ifp1
  use /d0/defs/os9defs
 endc

*Data area
 org 0
stack rmb 200
datsiz equ .

*Module header
type equ PRGRM+OBJCT
revs equ REENT+1
 mod length,name,type,revs,start,datsiz
name fcs /WORLD/
msg  fcc /Hello World/
 fcb 13

*main code
start leax msg,pcr
 ldy #12
 lda #1
 os9 i$writln
 bcs error
 clrb
error os9 f$exit

 emod

length equ *
```

One of the first problems is actually getting an environment to write this in. Dragon OS9 comes with an edit command, which in all honestly is complete rubbish and I'd suggest you use Stylo or another editor. I use a program called 'ed' which is based on the Unix 'vi' editor. Once you've written it, saved it to disk, you can assemble it using the OS9 assembler:

```
asm world #20k
```

Assumes you've saved the file as world. Assuming all goes well, you will end up with a file called 'world' in your default execution directory (typically /d0/cmds). However, it is the norm for OS9 (I have found anyway) that things are not often that straightforward. Firstly, you may find you don't actually have a copy of the 'asm' program, and even if you do the program you have just written requires an additional file (called up by the 'use /d0/defs/os9defs' statement) which may or may not be on your disk. Either way, you will probably end up searching through a variety of disks in order to get the files required to the right place. Once you have, and a great deal of disk chugging later with any luck, simply typing 'world' at the OS9 shell prompt,

should once again after a bit of disk chugging display the message 'hello world' on your screen.

So how does it all work, and is it all worth it. Firstly, the NAM and TTL directives just tell the assembler the name and title of your program and are not strictly required. Next, there is an assembler directive to include the file /d0/defs/os9defs but only on pass one (the IFP1 condition) of the assembly. This file contains a list of definitions about OS9 and is almost always required for any OS9 module. Whilst you can get away without it, for general neatness and code readability it's a good idea to use it. For example, in our simple program, the words PRGRM, OBJCT, i$write and f$exit are all definitions from os9defs.

Next the program's data area is defined, which as a minimum should define at least 200 bytes for the stack (I'll cover data areas later). Following this, information for the programs module header is defined. In OS9, each program unit or module must conform to a specific format in order for OS9 to recognize it and successfully execute it. In particular, it contains information about the execution address, how much data size is required and also a checksum of the whole module. All this minimizes the risk of trying to run corrupted or invalid code. The 'mod' assembler directive tells the assembler to form this header using the information provided. This requires the overall length of the module then a pointer to the name of the module. Next are two equates defined as TYPE and REVS. The TYPE equate is a single byte defining the module type and language and is preset to a normal program module (PRGRM) and 6809-object code (OBJCT). The REVS equate is a single byte defining the module attributes and revision level. It has an attribute of sharable (REENT) and a revision level of 1. The final two parameters are the start address and overall module data size.

Following the module header is the module name. The FCS directive is much like the FCC directive except the top bit of the last character is the string is set. This is often used by OS9 to indicate the end of a string. Finally, the main code for the program occurs.

The code make use of the OS9 system call i$writln. In order to make a system call, the assembler directive 'os9' is used followed by the call name. All the valid OS9 call names are defined in the os9defs file. OS9 system calls are made through one of the software interrupts, in this case SWI2 followed by a unique code identifying the call number. Therefore, the assembler directive 'os9 i$writln' translates into 'SWI2 $8C' where $8C is the i$writln code.

This call is much like the ROM call at $90E5, where the X register points to the string to display. In addition, however the Y register should also contain the maximum number of characters to display and the A register should contain the path number. A path number of 1 refers to the standard output - the display. The call will then display the required number of characters, or up to the first carriage return - whichever occurs sooner. On completion of the call, the carry bit is set if an error has occured with the error number in the B register. Finally, the routine is terminated with a call to the F$EXIT routine not an RTS instruction.

Once the code is complete, the EMOD statement indicates that this is the end of the module, and generates the module's checksum and then the length label is calculated.

This program also gives an example writing position independent code ie. code that can be executed anywhere in memory. Note how the X register in the os9 example is set to point to the string compared to the DragonDOS one:

```
DDOS: LDX #@MSG       - absolute address of MSG
OS9: leax msg,pcr     - relative address of MSG
```

The OS9 variant sets X to point to the MSG label relative to the current program counter ie. whereever you are in memory. The DragonDOS one specifies the absolute

address in memory. This means that with the DragonDOS code you are forced to have your program at this location in order for it to work. Of course, you could replace your DragonDOS one with the same instruction as the OS9 one and make it position independent, but with OS9 you must use this format or it will not work.

2. OS9 Advantages

From the last couple of articles, we have created two comparable programs under Dragon BASIC and OS9 - the now famous 'Hello World' program. However, the steps to create the OS9 one are significantly more complex than the DragonDOS counterpart. We have already seen one of the OS9 program's advantages - that of position independent code, though with a bit of forethought the Dragon BASIC one can also be made position independent. Here are a few other advantages of the OS9 variant.

Suppose you wanted to print the words 'Hello World' on the printer. This can be accomplished fairly simply under both OS9 and Dragon BASIC:

```
Dragon BASIC:  POKE 111,254:EXEC
  OS9:  world >/p
```

The Dragon BASIC variant simply sets the default IO location to 254 (-2) the printer. The OS9 one utilises the shell re-direction operator (>) to send the output to the printer. So far so good. What if you wanted to send this message to a file for example. With standard DragonDOS, there is no easy way - later DOS variants can use the OPEN# format:

```
OPEN #1,"O","TEST.DAT"
POKE 111,1:EXEC
CLOSE #1
```

Once again the OS9 variant is pretty much straightforward:

```
world >test.dat
```

Suppose then you have added a custom device to your system, an extra parallel port for example for a second printer. Under OS9 all you need to do is create a new device driver/descriptor and call it say P1. The world program can automatically be used with your new device:

```
world >/p1
```

The OS9 world program can then theorhetically be run on any OS9 system and make use of any device which is implemented on that machine.

The other 'selling point' of OS9 is it's multi-tasking ability - ie. run more than one program at once. This is all handled by the OS9 kernel routines, and you do not need to worry about it when writing code, except to ensure your programs are position independent, and re-entrant. A re-entrant program is all to do with how any data your program utilises is managed and means that one copy of a program in memory can be running a number of times with a different set of data.

3. OS9 Data Storage

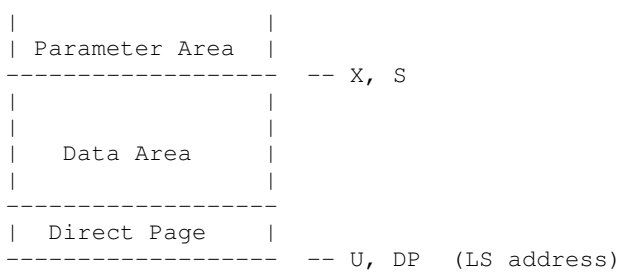
When you write your OS9 program, you should allocate all your data areas using rmb directives starting at 0 eg.

```
*Data area
 org 0                  specify starting at 0
length rmb 2            data items for
name   rmb 20           the program
next   rmb 2
stack  rmb 200
datsiz equ .
```

Your program's stack is also part of this data area, so you should allocate a sufficient area for this - 200 bytes is a recommended minimum. The datsiz label gives the total data size required for the program, and is used as part of the module header created by the MOD assembler directive:

```
 mod length,name,type,revs,start,datsiz
```

When OS9 starts your program, it sets up some of the CPU registers to provide information about your data area-



Firstly, the U and DP registers point to the start of your data area, with the stack pointer set to the top of your data area. Essentially then, you just need to reference the U register when accessing your data variables within your program:

```
 sty length,u           stores Y into 'length'
 ldx next,u             loads X from 'next'
```

However, any data items stored within the first 256 bytes of your data area can be accessed more quickly and efficiently by making use of the DP register. This is an 8 bit register which contains the upper 8 bits of an address. In order to specify DP addressing with the OS9 assembler simply supply the data label:

```
 sty length             use DP addressing to
 ldx next               access data items
```

Using the data declarations used earlier:

```
 length lies at offset 0 ($00).
 next lies at offset 22 ($16).
```

The assembler translates this into:

```
STY $00
LDX $16
```

instructing the processor to use the DP register to provide the upper part of the address. Therefore, if DP=$20 this would become:

```
STY $2000
LDX $2016
```

Not only does this form of addressing take up less storage space, it is also quicker to execute.

You will still need to use the U register to set up pointers to data buffers since you cannot use the DP register for this:

```
leax name,u       X=start of name item
```

There is an important difference between the way the OS9 assembler and DragonDOS assemblers (DASM & DREAM) use the DP register. By default, the OS9 assembler will use the DP register if you just enter something like:

```
ldx next  -> LDX $16
```

However, DragonDOS assemblers will automatically use extended addressing so assuming @NEXT lies at address $5020.

```
LDX #@NEXT -> LDX $5020
```

The assembler directive '>' is used to set extended addressing under OS9 ie.

```
ldx >$16   -> LDX $0016
```

and co-incidently is used to force DP addressing under DragonDOS ie.

```
ldx >$F0  -> LDX $F0
```

The DP register is nearly always set to 0 under DragonDOS so you can use it to quickly access memory locations 0-255.

So, just by the kernel setting the U and DP registers to different positions in memory the same piece of code need only be resident in memory once but can be running in different tasks with different data areas. The REENT bit set in the module header is used to tell OS9 that our program supports this - if you recall:

```
*Module header
type equ PRGRM+OBJCT
revs equ REENT+1
 mod length,name,type,revs,start,datsiz
name fcs /WORLD/
msg  fcc /Hello World/
 fcb 13
```

The parameter area (pointed to by the X register on program entry) is utilised to pass data to a program. If it has been started from the shell (command line) any text following the program name will be stored here terminated by a carriage return ie.

```
OS9:world this is a parameter
```

then X would point to a block of memory containing the ASCII string:

```
this is a parameter[CR]
```

4. Other Programming Tips

Generally, thats about all there is to writing OS9 programs. Remember, that an OS9 process should exit by calling os9 f$exit (with the B register containing any error code you wish to return, or 0 for none) instead of rts. The OS9 System Programmers Manual which you should have with your copy of OS9 contains all the system calls available. However, its worth mentioning a bit about standard IO paths.

We have already seen standard output used with the Hello World program:

```
 ldy #12
 lda #1          standard output
 os9 i$writln
 bcs error
```

On entry to any OS9 program 3 file paths are open:

```
0 = Standard Input (keyboard)
1 = Standard Output (screen)
2 = Standard Error (screen)
```

These may or may not correspond to screen or keyboard depending on whether the shell or other program has re-directed them ie.

```
world >/p
```

forces standard output to be the printer. This means that for the duration of your program, path number 1 is still valid but points to the printer instead. This is the premise of a lot of OS9 utilities called filters which typically perform some kind of modification to incoming data and write the modified data out. An example might be a program to add line feeds to a file in preparation to porting it to another environment which requires CR/LF terminated strings as opposed to CR which OS9 uses. The basis of the filter might be:

```
loop
 leax buffer,u      point to buffer
 ldy #1             1 char to read
 clra               from stdin
 os9 i$read         read byte
 bcs error          handle error
 inca               set to stdout
 os9 i$write        write byte
 bcs error
 ldb ,x             was it a CR
```

```
 cmpb #13
 bne loop           no, loop for next
 ldb #10            write an LF
 stb ,x             to stdout
 os9 i$write
 bcs error
 bra loop

error cmpb #E$EOF   check for EOF
 bne term           if not, exit with error
 clrb               else quit ok
term os9 f$exit
```

Here we make use of the i$read, i$write calls which read and write 'raw' data. A byte is read from stdin and then written out again to stdout. If it was a CR then an LF is also written to stdout. The program will terminate when an error condition occurs. If all the data is exhausted, this error will be EOF on the read which is therefore not treated as an error and causes the program to exit without error. This is the case for all filters. Assuming the program is assembled to the file 'addlf' an example usage could be with the stylograph mail merge utility (mm):

```
mm my_doc ! addlf >/d1/my_doc.out
```

Here the Stylograph file 'my_doc' is output by mailmerge and piped into our addlf utility, so stdin is redirected here from the pipe. The resulting output (stdout) is then sent to the file 'my_doc.out'.

This filter isn't very efficient, because it carries the overhead of calling i$read/i$write for every byte. If you recall, an OS9 system call translates into an SWI2 (software interrupt) call. This results in a lot of processing, and also the stacking/de-stacking of all the registers required for an SWI call. It would be far better to read the data in larger chunks and process it in a separate loop.

Note that any registers not updated explicity for output of a system call will remain unchanged from when the call was made eg.

```
i$read:  Entry:  X = Addr of data
   Y = Bytes to read
   A = Path number
  Exit:   Y = Bytes actually read
   ALL OTHER REGISTERS UNCHANGED
```

5. Device Drivers

One of the things which became apparant after the infamous 'world' routine was that this simple program could make use of any OS9 system peripherals provided the drivers were there for it to use. This is also true of most OS9 software - if you have the drivers any software can use it. For example, if you added a hard drive to your system and called it H0, you could quite easily store Stylograph on it, save your data on it, run the C compiler not to mention our 'Hello World' program. Unfortunatly, this isn't the case with DragonDOS which is more or less tied to the pre-set Dragon hardware. So if you are planning to add ram discs, hard discs etc. then OS9 is by far the easiest platform to write software for. You will also probably find that writing software becomes that much easier for OS9 once you have access to faster disk drives and maybe an 80 column screen. It's by no means impossible to write stuff for DragonDOS - just a lot harder thats all and I'll touch on this a bit later.

The key to this flexibility really comes down to device drivers and descriptors on an OS9 system. Generally, any device you add will fall under the category of disk - in which case an RBF device driver is required or sequential (serial port, printer etc.) in which case an SCF device driver is required.

Device drivers are just like any other OS9 program, they have a module header, data area and code to perform to necessary operations. Here is a rough outline of a device driver which is called DEVx:

```
 nam DEVx
 ttl DEVx device driver

 ifp1
 use /h0/defs/os9defs
 use /h0/defs/iodefs
 use /h0/defs/scfdefs
 endc

*data
 org V.SCF
PST equ .

type equ DRIVR+OBJCT
revs equ REENT+1
 mod PEND,PNAM,type,revs,PENT,PST

PNAM fcs /DEVx/

PENT lbra INIT
 lbra READ
 lbra WRITE
 lbra GETSTA
 lbra PUTSTA
 lbra TERM

INIT   .
READ   .
WRITE  .
GETSTA .
PUTSTA .
TERM   .

 emod
PEND equ *
```

It follows the standard module structure, first the name and title of the module NAM and TTL. Then the 'use' files are given. As well as os9defs we include iodefs which contains i/o specific definitions, and because this just happens to be an SCF device, scfdefs are included also. Next comes the data area, however because the file managers (RBF or SCF) also require some data, you should not start your data area off at 0 but the V.SCF equate (or DRVBEG for RBF devices). No stack space is required here as the stack used will be the one from the calling program. The PST label marks the datasize of the module, and then the module header is given. Notice that the PRGRM equate is replaced by the DRIVR equate indicating this is a device driver. The PENT label marks the start of the driver. All OS9 drivers have 5 entry points at the module start and LBRA instructions should always be used to call these routines which are as follows:

INIT - called once on module start. Include any initialisation code required to set your device up.

READ - called to read an 'item' of data from the device. Depending on the file manager an 'item' can be anything from 1 character (SCF devices) to a disk sector (RBF devices).

WRITE - called to write an 'item' of data to the device.

GETSTA - called to perform miscellaneous IO status routines. Since all IO devices have different capabilities this provides a method to perform operations which generally return some sort of status back to the caller - for example the End Of File function (EOF) is implemented via a GetStat call.

SETSTA - similar to GetSta, but generally used to set some status information on the device.

TERM - called once when the module is about to be removed from memory. Include any de-allocation of memory/clean up routines etc. here.

Prior to showing a couple of example drivers, there are a few data structures you need to be aware of when dealing with device drivers. Firstly, there is your static storage, which can be thought of in similar terms to the data area of a normal module. The static storage area for your device driver starts at the V.SCF (or DRVBEG) position given in the previous header file, and similarly to a normal OS9 program, the U register always points to this area. Note, that unlike a module's data area, you should not use DP addressing on data items. So, if you declare a variable as such:

```
 org V.SCF
flag rmb 1
```

it can be accessed in any of the 5 procedures as:

```
 lda flag,u
```

In addition to your own data held here, the system also stores some of it's own static data just prior to the V.SCF (or DRVBEG) label which you can also access. This consists of a 'common' area followed by a file manager specific area. The exact format of these are listed in the System Programmers Guide (and also in the iodefs,scf/rbfdefs files) but one of these items held in the common area which the example drivers will make use of is named V.Port and holds the address of the physical device in memory. How this gets set up, I will cover later but suffice to say it enables the driver to access a physical device in memory without having to know where it is exactly. As an example, if you had an SCF device driver to talk to MC6821 PIAs and you had 4 of these PIAs in your system you would only need 1 copy of the driver loaded - the system would ensure V.Port is setup correctly depending on which PIA is being accessed:

```
 lda #$FF      A=255
 ldx V.Port,u  port address
 sta 1,x       port address+1=255
```

The other data block you may need to be aware of is called the path descriptor. This is a unique set of data for each open file or device in the system. You cannot allocate your own data items here, but you can read & write to data items here. Generally, the path descriptor holds information relevent to the current open file or device (such as path numbers, file lengths etc.) rather than constant device characteristics (port address etc.). Again, there is a common part of the path descriptor, and then a file manager specific area. The format of path descriptors is also listed in the System Programmers Guide and can also be found in the iodefs,scf/rbfdefs files. Initially, you will not probably need to utilise anything in the path descriptor, but if you start writing more complex drivers you almost certainly will - generally the file manager will provide most of the path descriptor management.

The path descriptor is pointed to by the Y register given in the READ, WRITE, GETSTA & SETSTA procedures (remember, it only exists for an open file so will not be accessable by the INIT & TERM routines) and as such can be accessed as follows:

```
 lda PD.PD,y    A=path number from path descriptor
```

Last time I covered general device drivers. Next, are 2 example drivers, one an SCF device, the other an RBF device.

6. SCF Device Driver

This device driver utilises an MC6821 PIA to perform character driven IO.

The first section is the common module header format:

```
*****************************************
* PIA21
*  Device Driver for MC6821 PIAs
*
* By J.Bird (c) 1992
*
* Uses Side B of MC6821 PIA
* CA2 & CB2 Control Strobe lines
* Non-interrupt driven
*

 nam PIA21
 ttl MC6821 PIA Device Driver

 ifp1
 use /h0/defs/os9defs
 use /h0/defs/iodefs
 use /h0/defs/scfdefs
 endc

**********************
* Edition History
*
* #   date     Comments
* - -------- ------------------------------
* 1 05.05.92  Driver first written. JRB
* 2 25.03.95  Update for Parallel Dragon link. JRB.

Revision equ 2

PBCREG equ %00101101
PBOUTP equ %11111111
PBINP  equ %00000000

 org V.SCF
PST equ .

 mod PEND,PNAM,Drivr+Objct,Reent+Revision,PENT,PST
 fcb READ.+WRITE.
PNAM fcs /PIA21/

PENT lbra INIT
 lbra READ
 lbra WRITE
 lbra GETSTA
 lbra PUTSTA
 lbra TERM
```

This defines an SCF man device called PIA21. On a sideline note, you may have noticed the revision number as 2, and this is built into the module header. This will ensure that if this module is loaded into memory and revision 1 is also in memory (as part of the OS9 boot load for example) that revision 2 will be used. Subsequently, a revision 3 would take precedence over 2 etc. etc.

The following two routines will be used throughout the code to configure the port for either input or output. Note the use of the V.Port address from the static storage to find the IO port's address.

```
**********************
* SETOUT
*   Set port dir for o/p
*
SETOUT ldx V.PORT,u       load port address
 clr 1,x                  clear control reg
 lda #PBOUTP              set port for o/p
 sta ,x
 lda #PBCREG              reload ctrl reg
 sta 1,x
 clrb
 rts

**********************
* SETIN ldx V.PORT,u
*   Set port dir for i/p
*
SETIN ldx V.PORT,u
 clr 1,x
 lda #PBINP
 sta ,x
 lda #PBCREG
 sta 1,x
 clrb
 rts
```

The first of the device driver procedures INIT, just configures the port as input and returns. All routines should return with B=0 and the carry bit clear unless you wish to report an error, in which case the B register should contain the OS9 error code. Note, that rts is used to return from a device driver routine, not os9 f$exit as per normal OS9 programs.

```
**********************
* INIT
* Entry: U = Static storage
*   Setup the PIA
*
INIT bsr setin
 clrb
 rts
```

The READ routine is required to fetch a data item from the IO port. For SCF type devices, this is 1 character and should be returned in the A register.

```
**********************
* READ
* Entry: U = Static Storage
*        Y = Path Descriptor
* Exit:  A = Character read
*
```

```
*   Read a byte from Port B
*
READ ldx V.PORT,u      load port address
readlp tst 1,x         test for data ready
 bmi readbyte
 pshs x                sleep for a bit if not
 ldx #10
 os9 f$sleep
 puls x
 bra readlp
readbyte lda ,x        read byte
 sta ,x                issue strobe
 clrb
 rts
```

The READ routine makes use of another OS9 system call - sleep. On entry, X should contain the number of clock cycles to sleep for (clock cycle = 1/50 second on the Dragon). This is not strictly required, however when an IO request is performed OS9 prevents any other process from running until it completes or executes a sleep request. For example, if you had a process running as a background task using this device and the device stalled (ie. it never became ready) your machine would lockup. By sleeping for a short while between checks you ensure this cannot happen.

The WRITE routine acts in a similar way to READ, except the character passed in the A register should be written to the port:

```
**********************
* WRITE
* Entry: U = Static storage
*        Y = Path Descriptor
*        A = Char to write
*
*   Write a byte to Port B
*
WRITE ldx V.PORT,U   load port address
 sta ,x              write byte
writlp tst 1,x       wait for ack
 bmi wrt
 pshs x
 ldx #1
 os9 f$sleep
 puls x
 bra writlp
wrt lda ,x           clear control reg
ok clrb
 rts
```

The next two routines demonstrate device driver usage of the GetStat and SetStat calls and also the path descriptor.

There are a few predefined GetStat calls under OS9 which nearly all SCF device drivers will have to deal with. These are Test for Data Ready and Test for End Of File. The predefined GetStat codes are defined in the OS9Defs file but be warned: a lot are device dependent and will not be implemented. Generally, only the ones listed above are available on all SCF devices.

GetStat calls are made by the caller by issuing an OS9 i$getstt call with the A register containing the path number and the B register containing the GetStat code - the other register contents depend on the GetStat call being made. This example program loops until data is available from the keyboard:

```
chklp clra             pathnum of stdin
```

```
      ldb #SS.Ready    getstat code
      os9 i$getstt     issue call
      bcc okay         if c bit clear data is rdy
      cmpb #E$NRDY     otherwise check the code
      bne error        returned was not ready
      bra chklp
```

In order to write a routine to process a GetStat call, you need to access the contents of the registers when the call was made - in this case the B register to obtain the getstat code. Within the common part of the path descriptor is a pointer to a data area which contains all the register contents when the call was made. Labels set up in the os9defs file make accessing this information easy.

The following GetStat routine processes 3 codes. It first retrieves the register pointer from the path descriptor then loads what was in the B register into the A register using the R$B label to find the location. The 3 getstat codes are SS.Ready - test for data ready which is performed by using the PIA to determine if a data byte is waiting to be processed or not, SS.EOF - end of file which for this device has no meaning and therefore always returns with no error ie. not end of file. The final code is a user defined one I created specific for this device named SS.VSupv. When called with this code it forces the port to change direction to input using the 'setin' procedure defined earlier. All other codes return an error.

```
*********************
* GETSTA
*   U = Static Storage
*   Y = Path Descriptor
*
GETSTA ldx PD.Rgs,y     X=pointer to registers
 lda R$B,x              A=contents of B reg.
 cmpa #SS.Ready         check for ready code
 bne gsta1              if not, check next
 ldx V.Port,u           use the PIA to determine
 tst 1,x                if data is available
 bmi ok
 ldb #E$NotRdy
 coma                   sets the carry flag
 rts
gsta1 cmpa #SS.EOF      check for eof code
 beq ok                 which always returns ok
 cmpa #SS.VSupv         check for a special code
 beq setin              which changes port dir
 comb                   otherwise return
 ldb #E$UnkSVC          unknown code error
 rts
```

The SetSta call works in an identical way to GetStat, and generally SCF devices do not use SetStat calls so you can just return the E$UnkSVC error immediatly. For our example driver, SetStat supports the new SS.VSupv code which when received will force the port direction to be output using the 'setout' procedure:

```
*********************
* PUTSTA
*   Supv request sets port to o/p
*
PUTSTA ldx PD.Rgs,y     X=register stack
 ldb R$B,x              B=contents of reg B
 cmpb #SS.VSupv         check for recognized code
 beq setout             and process it
 comb                   otherwise return error.
 ldb #E$UnkSVC
```

```
 rts
```

Finally, the TERM routine. Since the driver has not allocated any extra memory or anything it can just return okay:

```
*********************
* TERM
*   Terminate Driver
*
TERM clrb
 rts

 emod
PEND equ *
```

Thats just about it.

Before proceeding to an RBF device driver there is one more thing required to complete the new SCF device - a device descriptor.

The device descriptor is just composed of a set of data items which define the device you are creating. In particular, it is the name you refer to when you want to access the device - so our example is called P1 so in order to access it you just refer to this device ie.

```
list startup >/p1
```

copies the startup file to device /p1. The descriptor also contains information to identify it's device driver, file manager and other unique information. It also contains the port address (which is copied to V.Port so your driver can refer to it) and in the case of SCF devices contains a whole list of attributes referring to the capabilities of the device.

Here is the PIA device descriptor to go with the PIA21 device driver:

```
************************************
* PIA Descriptor module
*
* Source by J.Bird (C) 1992
*
 ifp1
 use /h0/defs/os9defs
 endc

 nam P1
 ttl PIA Device Descriptor

 mod PEND,PNAM,DEVIC+OBJCT,REENT+1,PMGR,PDRV

 fcb READ.+WRITE.+SHARE.
 fcb $FF IOBlock (unused)
 fdb $FF32 hardware address
 fcb PNAM-*-1 option byte count
 fcb $0 SCF device
 fcb $0 Case (upper & lower)
 fcb $1 Erase on backspace
 fcb $0 delete (BSE over line)
 fcb $0 echo off
 fcb $1 lf on
 fcb $0 eol null count
 fcb $0 no pause
```

```
 fcb 24 lines per page
 fcb $8 backspace
 fcb $18 delete line char
 fcb $0D end of record
 fcb $1b eof
 fcb $04 reprint line char
 fcb $01 duplicate last line char
 fcb $17 pause char
 fcb $03 interrupt char
 fcb $05 quit char
 fcb $08 backspace echo char
 fcb $07 bell
 fcb $00 n/a
 fcb $00 n/a
 fdb pnam offset to name
 fdb $0000 offset to status routine
pnam fcs "P1"
pmgr fcs "SCF"
pdrv fcs "PIA21"
 emod
pend equ *
 end
```

Once again it's format is similiar to any OS9 module, with a program type of DEVIC indicating device descriptor. The module just consists of fcb data statements. Generally, this may well suffice for any SCF module you care to create. Some key features to note however are:

```
 fcb READ.+WRITE.+SHARE.
```

indicates the device can be read, written and also multiple processes can access it at any one time. Failing to specify some of these will cause the system to return 'device not capable of operation' type errors.

```
 fdb $FF32  hardware address
```

The physical hardware port address. This indicates our PIA sits at $FF32 in memory - copied to V.Port in the device driver.

```
 fcb $0     echo off
```

This is an important one. If set to non-zero when the SCF file manager reads a byte it will automatically send it back out using the write routine. In the case of our 6821 driver which only operates in one direction at a time this is not a good idea.

```
pnam fcs "P1"
pmgr fcs "SCF"
pdrv fcs "PIA21"
```

Finally, these give the device descriptor name (pnam), its associated file manager (pmgr) and device driver (pdrv) and is the only way the system has of working this out.

As an additional note, the device driver INIT also sets the Y register to point to the device descriptor data should you need to access it. Most of the parameters are also copied into the path descriptor option section where you can access them if required.

Remember, any modifications you make to the path descriptor locations will only be valid for the duration the device or file is 'open'.

Once you have your new driver and descriptor you just load them into memory and then any OS9 program can use them:

```
load pia21
load p1
list startup >/p
```

7. RBF Device Driver

RBF drivers tend to be a little more complex than SCF ones but not drastically so. In addition, practically everything covered in the SCF device also applies to RBF device covered here. They also tend to open up your system a lot more - essentially you are getting another disk drive out of it. The example I am using here is based around the RAM disc design[1] I wrote a couple of years ago in Up2Date.

The RAM disc design used once again an MC6821 PIA, which utilised one side to select a RAM function to be performed and the other side as the data bus. The design was specifically built to make it easy to interface into an OS9 system.

For RBF devices, the read and write functions are required to transfer a 256 byte sector from a given 24 bit logical sector number (LSN). LSNs give a means of addressing disks without having to worry about tracks/sectors per track. They are numbered from 0 upwards starting (on a 'real' disk anyway) as LSN 0=track 0, sector 1 etc. On a standard Dragon disk (40T/SS) LSN 0=Track 0, sector 1, LSN 1=Track 0, sector 2, LSN 18=Track 1, sector 1 etc.

The PIA design allowed for the following operations:

1. Write MSB of LSN. Write the top 16 bits of the LSN to the RAM drive (the top 8 bits are discarded as you would need a phenominal amount of RAM to need these bits).

2. Write LSN of LSN. Write the botton 8 bits of the LSN to the RAM drive.

3. Read a sector. Transfer the sector byte by byte from the RAM drive hardware to a buffer.

4. Write a sector. Transfer a sector byte by byte from a buffer to the RAM drive hardware.

So, there is really minimal processing involved to utilise the RAM drives. Most floppy, or hard disk drivers would normally require some LSN to track/sector conversion.

All RBF device drivers differ subtly from SCF drivers in that there are certain mandatory things you must perform within them in order for them to function correctly. I will attempt to explain these throughout the driver example.

An RBF driver starts off like all OS9 modules, with the standard header format. Here you see I am up to revision 6 of the PIA RAM disk driver:

```
*******************************************
* Rdisk
*  A driver for a Ram disk!
* By G.Twist (c) 1986
* modified by Bernd H. Neuner 1987
* Version for a totally different Ram disk!
*
* By J.B. & O.B. (C) 1991,1992
*

 nam Rdisk
```

```
 ttl A Device Driver for a RAM Disk

 ifp1
 use /d0/defs/os9defs
 use /d0/defs/iodefs
 use /d0/defs/rbfdefs
 endc

**********************
* Edition History

*  #   date    Comments
* -- -------- ------------------------------------------
*  1 86/12/17  Driver first developed GDT
*  2 86/12/18  work to fix minor access bugs GDT
*  3 30.11.87  bug in COPY routine fixed. BHN (no more error 214 now.)
*  4 31.12.91  Test version for main RAM JB/OB
*  5 08.01.92  Driver for 128K PIA RAM JB/OB
*  6 18.09.92  Up-issue to support up to 512K JB

Revision equ 6
NumDrvs  set 1 Number of drives

* pia control comands
pia.ddr equ %00111000
pia.off equ %00111100
pia.act equ %00101100
ext.msr equ %00000001
ext.lsr equ %00001000
ext.read equ %00000010
ext.writ equ %00000110
output equ %11111111
outb equ %00001111
pia.iora equ 0
pia.cnra equ 1
pia.iorb equ 2
pia.cnrb equ 3


 org Drvbeg
 rmb NumDrvs*DrvMem
LSNZERO  rmb 1
RAMSTA equ .

 mod RAMEND,RAMNAM,Drivr+Objct,Reent+Revision,RAMENT,RAMSTA
RAMNAM fcs /Rdisk/

RAMENT lbra INIT
 lbra READ
 lbra WRITE
 lbra GETSTA
 lbra PUTSTA
 lbra TERM
```

Along with the equates needed for the driver, the RBF driver static storage is being utilised (note the V.SCF equate replaced with DRVBEG). The first mandatory thing about an RBF driver is that you must allocate a drive table for EACH drive you plan the driver to deal with. The drive table will hold information particular to the drive (such as number of sectors/track etc.) and must be the first block of data in your static storage. You should reserve DRVMEM bytes (DRVMEM is defined in the rbfdefs file) multiplied by the number of drives your driver can handle (in this case NumDrvs is defined as 1). So the DDISK OS9 driver reserves 4 times this amount to handle the 4

possible floppy disks on a Dragon system. The only data item declared for our use is the LSNZERO byte.

The INIT routine of the driver also requires some mandatory code required by all RBF drivers:

1. Initialise the V.NDRV variable of the RBF static storage to the number of drives the driver will deal with.

2. Initialise the DD.TOT and V.Trak variables held within the drive tables of the static storage of EACH drive to a non-zero value.

In this driver, although it is only targetted for 1 drive, there is a loop construct present for dealing with multiple drives should this ever be a requirement. In addition, our INIT routine also sets up the PIA and initialises our own data item.

```
****************************
* INIT
*  Set up the ramdisk

INIT lda #NumDrvs Set no drives to 1
 sta V.NDRV,U
 clr LSNZERO,U    Initialise our data
 lda #$FF         non-zero value
 leax DrvBeg,U    X=Start of drive table
initdrv
 sta DD.Tot,X     write in non-zero values
 sta V.Trak,X
 leax DrvMem,x
 deca             loop through drives
 bne initdrv
* set up pia
 ldx V.PORT,U
 lda #pia.off
 sta pia.cnra,x   select a side off
 lda #pia.ddr
 sta pia.cnrb,x   select b side ddr
 lda #outb
 sta pia.iorb,x   set ls 4 bits to outputs
 lda #pia.off
 sta pia.cnrb,x   select b side io reg
 clr pia.iorb,x
 clrb
INITXIT rts
```

The READ routine is required to transfer an entire disk sector to a buffer. The 24 bit LSN number to read is held in the B and X registers as follows:

```
  bits#
  23 ... 15 ... 7 ... 0
  |   B  |      X     |
```

The VALID procedure actually performs the sector validation and loading of this data into the PIAs. The sector is then transfered into the sector buffer pointed to by the PD.BUF location in the path descriptor.

The READ routine (surprisingly enough) also has to perform some mandatory operations - in particular when LSN 0 is read. This contains disk ID information and whenever a request is made to read this sector, the driver is required to copy DD.SIZ bytes into the drive table for the required drive:

```
****************************
* READ
```

```
*  read a sector from disk
*  Entry: U = Static Storage
*         Y = Path Descriptor
*         B = MSB of LSN
*         X = LSB's of LSN
*  Exit: 256 byte sector in PD.BUF buffer
*
READ clr LSNZERO,U      init LSNZERO flag
 bsr VALID              validate the LSN supplied
 bcs READ99             raise error if invalid
 pshs Y                 preserve pointer to PD
 ldy PD.BUF,Y           Y=start of sector buffer
 ldx V.PORT,U           X=PIA port address
 lda #pia.ddr           program PIA for transfer
 sta pia.cnra,x
 clr pia.iora,x
 lda #pia.act
 sta pia.cnra,x
 lda #ext.read
 sta pia.iorb,x
 leax pia.iora,x
 clrb
READS10 lda ,x          transfer sector from PIA
 sta ,y+
 decb
 bne READS10
 leax V.Port,u          reset PIA
 clr pia.iorb,x
 lda #pia.off
 sta pia.cnra,x
 puls Y
*mandatory RBF code - copy LSN 0
 tst LSNZERO,U          VALID routine will set this
 beq READ90             if LSN 0 specified
 lda PD.DRV,Y           extract drive num
 ldb #DRVMEM            size of drive table
 mul
 leax DRVBEG,U          start of drive tables
 leax D,X               add on drive offset
 ldy PD.BUF,Y           copy DD.Siz bytes
 ldb #DD.SIZ-1          into drive table
COPLSN0 lda B,Y
 sta B,X
 decb
 bpl COPLSN0
READ90 clrb
READ99 rts
```

The sector write routine works in a similar manner: the LSN is supplied in the same format, and this time the sector held in PD.BUF must be transferred to the RAM disk PIAs. No special processing is required for LSN 0.

```
****************************
* WRITE
*  Write a sector to disk
* Entry: U = Static Storage
*        Y = Path Descriptor
*        B = MSB of LSN
*        X = LSB's of LSN
*        PD.Buf = Sector to write
*
WRITE bsr VALID          validate sector
 bcs WRIT99
WRITS pshs Y             save PD
```

```
 ldy PD.BUF,Y            Y=sector buffer
 ldx V.PORT,U            X=port address
 lda #pia.ddr            program PIA
 sta pia.cnra,x          for write txfr
 lda #output
 sta pia.iora,x
 lda #pia.act
 sta pia.cnra,x
 lda #ext.writ
 sta pia.iorb,x
 leax pia.iora,x
 clrb
WRITS10 lda ,y+          transfer sector
 sta ,x                  to PIA
 lda ,x
 decb
 bne WRITS10
 ldx V.PORT,U            restore PIA
 clr pia.iorb,x
 lda #pia.off
 sta pia.cnra,x
 puls Y
 clrb
WRIT99 rts
```

The VALID subroutine is shown below. This serves to program the PIA with the LSN supplied - note the top part of the LSN held in the B register is discarded for this driver, only the X part is used.

```
*****************************
* VALID
*  validate a sector
*  and set up external registers
*  to reqired page in ram
*
VALID
 cmpx #$0000     check for LSN 0
 bne NOTLSN0     set flag appropriatly
 inc LSNZERO,U
NOTLSN0 pshs y
 ldy V.PORT,U
 lda #pia.ddr    select direction reg
 sta pia.cnra,y
 lda #output     set bits to output
 sta pia.iora,y
 lda #pia.act    enable ca2 strobe
 sta pia.cnra,y
 lda #ext.msr    select ms sector reg
 sta pia.iorb,y
 tfr x,d
 sta pia.iora,y  write ms value
 lda pia.iora,y  do the read (strobe)
 lda #ext.lsr    select ls sector reg
 sta pia.iorb,y
 stb pia.iora,y  write ls value
 ldb pia.iora,y  do the read (strobe)
 clr pia.iorb,y  select nothing
 puls y          note a side still set for output
 clrb
 rts
```

On the GETSTA and SETSTA side of things, there are no 'mandatory' getsta codes you need to deal with and two SETSTA codes: SS.RST (restore head to track 0) and SS.WRT (write track). Restore head to track 0 serves no useful purpose on a RAM disc and can just return okay. The SS.WRT call is used during format operations to actually format the track - again for a RAM drive this will not be required and can just return okay. All others return with an error:

```
*************************
* GETSTA
*  get device status
*
GETSTA
Unknown comb
 ldb #E$UnkSVC
 rts

*************************
* PUTSTA
*  Set device Status
*
PUTSTA cmpb #SS.Reset
 beq PUTSTA90
 cmpb #SS.WTrk
 bne Unknown

PUTSTA90 clrb
 rts
```

Finally, the TERM routine simply exits cleanly:

```
*****************************
* TERM
*  terminate Driver
*
TERM clrb
 rts

 emod
RAMEND equ *
```

All that remains to cover is the RBF device descriptor, which is actually a lot simpler than the SCF one:

```
****************************************
* RamDisk Discriptor module
*
*
 ifp1
 use /d0/defs/os9defs
 endc

 nam R0
 ttl Drive Discriptor module

 mod DEnd,DNam,DEVIC+OBJCT,REENT+1,DMgr,DDrv

 fcb DIR.+SHARE.+PREAD.+PWRIT.+UPDAT.+EXEC.+PEXEC.
 fcb $FF IOBlock (unused)
 fdb $FF34 hardware address
 fcb DNam-*-1 option byte count
 fcb $1 Rbf device
 fcb 0 Drive number
```

```
 fcb 03 6ms Step rate
 fcb $80 Standard OS9 Winchester drive
 fcb 0 Single density
 fdb 4 number of tracks
 fcb 1 number of sides
 fcb 1 dont verify any writes
 fdb 256 sectors per track
 fdb 256 sectors on track 0, side 0
 fcb 1 sector interleave factor
 fcb 1 sector allocation size
DNam fcs "R0"
DMgr fcs "RBF"
DDrv fcs "RDisk"
 emod
DEnd equ *
 end
```

This is the device descriptor named R0 for our RAM drive. It shares similar properties to the SCF one, firstly the attributes byte indicating full access (READ.+WRITE. etc.) including directory access. There is also the familiar hardware address and at the end the device descriptor name, file manager name and driver name. Most of the descriptor is composed of the drive attributes: drive number, total tracks, sectors per track etc. Since this is a RAM drive all I have done is ensured the total sectors on the media is equal to the amount of RAM I have available - in this case 256K.

Once assembled, you can simply load and access the RAM drive like any other disk on your system:

```
load rdisk
load r0
chd /r0
copy /d0/startup /r0/startup #32k

etc.
```

That just about wraps it up. A couple of other points worth noting: it's normally worth either checking out or basing your driver on someone else's as a good starting point, there is also a fair deal of information around particularly from the CoCo side of things. And something I'd always recommend: as you have seen is nearly always easier to debug stuff under normal DragonDOS so I'd suggest at least prototyping your code under this environment to see if at least the logic works how you think it should. That way, when you port it to OS9 all you have to worry about is the quirks of OS9 rather than your own driver logic.

Notes

1. http://www.onastick.clara.net/rdisk.htm

OS-9 System Extension Modules

Boisy G Pitre

Table of Contents

The technical information, especially in the OS-9 Level Two manuals, is brimming with details and information that can unlock a wealth of understanding about how OS-9 works. Unfortunately, some of this information can be hard to digest without proper background and some help along the way. This series of articles is intended to take a close look at the internals of OS-9/6809, both Level One and Level Two. So along with this article, grab your OS-9 Technical Manual, sit down in a comfortable chair or recliner, grab a beverage, relax and let's delve into the deep waters!

1. Assemble Your Gear

For successful comprehension of the topics presented in this and future articles, I recommend that you have the following items handy:

- OS-9 Level Two Technical Reference Manual *or*
- OS-9 Level One Technical Information Manual (light blue book) and the OS-9 Addendum Upgrade to Version 02.00.00 Manual
- A printout of the `os9defs` file for your respective operating system. This file can be found in the DEFS directory of the OS-9 Level One Version 02.00.00 System Master (OS-9 Level One) or the DEFS directory of the OS-9 Development System (OS-9 Level Two).

In this article, we will look at a rarely explored, yet intriguing OS-9 topic: system extensions, a.k.a. P2 modules. When performing an `mdir` command, you have no doubt seen modules with names like `OS9p1` and `OS9p2` in OS-9 Level Two (or `OS9` and `OS9p2` in OS-9 Level One). These modules are essentially the OS-9 operating system itself; they contain the code for the system calls that are documented in the OS-9 Technical Reference documentation. In the case of OS-9 Level One, the modules `OS9` and `OS9p2` are located in the boot track of your boot disk (track 34). In OS-9 Level Two, `OS9p1` (equivalent to the `OS9` module in Level One) is found in the boot track while `OS9p2` is located in the bootfile. Both of the modules are of module type `Systm` and define the basic behavior and structure of OS-9. Even the module IOMan is a system extension, containing code for the I/O calls in the operating system.

While drivers and file managers have been the most common area to expand the capabilities of OS-9, they are pretty much limited to expanding the functionality of I/O. What system extensions allow you to do is even more powerful: they can add new system calls or even replace existing ones. Such functionality allows you to change the behavior of OS-9 in a very fundamental way. Of course, with such power, caution must be exercised. It is not wise to radically modify the behavior of an existing system call; such an action could break compatibility with existing applications.

What we aim to do in this article is not to replace an existing system call, but rather to add a new system call by looking at the example provided in Tandy's OS-9 Level Two documentation. Although the example is written for OS-9 Level Two, we will

look at how it can be changed to run under OS-9 Level One as well. But first, let's get a little background on system calls and how they are constructed in OS-9.

2. The System Call

As an operating system, OS-9 provides system level functions, or system calls to applications. These system calls give applications a base by which they can operate consistently and without fear of incompatibility from one OS-9 system to the next. The system call in OS-9/6809 evaluates to an SWI2 instruction on the 6809, which is a software interrupt. Suffice it to say that when this instruction is encountered by the CPU, control is routed to OS-9, which interprets and performs the system call on behalf of the calling process.

While system calls are generally hidden by wrapper functions or procedures in high-level languages such as Basic09 and C, we can see the system call in its native form by looking at 6809 assembly language. Consider the following assembly source fragment:

```
      lda   #1
      leax  mess,pcr
      ldy   #5
      os9   I$Write
      rts
mess  fcc   "Hello"
```

In the middle of what appears to be normal 6809 assembly language source code is a mnemonic called `os9`. This is a pseudo mnemonic, since Motorola did not place an `os9` instruction in the 6809 instruction set. The OS-9 assembler actually recognizes this pseudo mnemonic as a special case, along with the `I$Write` string, and translates the above piece of code into:

```
      lda   #1
      leax  mess,pcr
      ldy   #5
      swi2
      fcb   $8A
      rts
mess  fcc "Hello"
```

The $8A which follows the `swi2` instruction is the constant representation of the I/O system call `I$Write`. Since the `swi2` instruction calls into the OS-9 kernel, the code in the kernel looks for the byte following the `swi2` instruction in the module (the $8A) and interprets that as the system call code. Using that code, OS-9 jumps to the appropriate routine in order to execute the `I$Write`.

Since the system call code following the swi2 instruction is a byte, in theory this would allow OS-9 to have up to 256 different system calls that can be executed on behalf of an application. Under OS-9 Level Two, this is the case; however under OS-9 Level One there are restrictions placed on exactly which codes are available. The following tables show the range of system call codes.

Table 2. OS-9 Level One System Call Ranges

System call range	Function
$00-$27	User mode system call codes
$29-$34	Privileged system mode call codes
$80-$8F	I/O system call codes

Table 3. OS-9 Level Two System Call Ranges

System call range	Function
$00-$7F	User mode system call codes
$80-$8F	I/O system call codes
$90-$FF	Privileged mode system call codes

The idea behind *User mode* vs. *System mode* is to allow two different points of execution for the same system call, depending on whether the calling process is running in user state or system state. OS-9 controls this by maintaining two system call tables: one for user state and one for system state. When installing a system call, as we'll soon see, we can specify whether our system call should only be called from system state (hence only updating the system table) or from both user and system state (updating both the user and system tables).

An example of a system call that can be executed in both user and privileged modes is the `F$Load` function code (pp. 8-25 in the OS-9 Level Two Technical Reference manual; pp. 106 in the OS-9 Level One Technical Information manual). Since `F$Load` can be called from a user state process as well as from a driver or other module running in system state, OS-9 installs this system call in both the user and system tables. On the other hand, a privileged mode system call such as `F$AProc` (Level Two: pp. 8-74; Level One: pp. 141) can only be called from system state and therefore a user state process attempting to call it will receive an error.

Notice that in both OS-9 Level One and OS-9 Level Two, codes $80-$8F are reserved for I/O system call codes. When the OS-9 kernel receives one of these codes, it passes the code along to `IOMan` for processing. I/O system calls cannot be added since they are under the control of `IOMan`.

Installing a new system call involves selecting a free system call code, determining whether the call will be accessible from both user/system state or from system state only, and building a table in assembly language that will be used to install the system call. Interestingly enough, the method of installing a system call is by calling a system call! It's called `F$SSvc` and is documented in your respective OS-9 Technical manual.

3. Installing a System Call in OS-9 Level Two

The source code in Listing 1 is the system extension module, os9p3.a, which contains the code to install the system call, as well as the system call code itself. Incidentally, this is virtually the same code that is found in the OS-9 Level Two Technical Reference Manual on pp. 2-2 to 2-4. I've eliminated the comments for brevity since they are already in your manual, as well as changed the `use` directive. Instead of including `/dd/defs/os9defs`, I include `/dd/defs/os9defs.l2`. The reason for this is that I do compiling of both OS-9 Level One and OS-9 Level Two modules on my CoCo 3 development system. Since the OS-9 definitions are different for each operating system, I have renamed their respective `os9defs` files with an extension indicating which operating system they belong to. Even if you just develop for one operating system or the other, I strongly suggest following the same naming convention; it will save you headaches in the long run.

This module, called `OS9p3`, installs the `F$SAYHI` system call. A process making this call can either pass a pointer to a string of up to 40 bytes (carriage return terminated) in register X, or set X to 0, in which case the system call will print a default message. In either case, the message goes to the calling process' standard error path. While not very useful, this system call is a good example of how to write a system extension.

The asm program is used to assemble this source code file. Notice that the entry point for the module is the label `Cold`, where Y is set to the address of the service table, `SvcTbl`. Each entry in this table contains three bytes. The first is the system call code that we have selected from a range that Microware says is safe to use for new system

calls, and the remaining two are the address of the first instruction of the system call. The table, which can contain any number of entries, is terminated by byte $80. After setting Y to the address of the service table, a system call to F$SSvc is made, which takes the table pointed to by Y and installs the system calls.

The code for the F$SAYHI system call in listing 1 is for OS-9 Level Two only. It determines whether or not a valid string pointer has been passed in register X. If indeed the caller has passed a valid pointer, then control is routed to the label SayHi6 where Y is loaded with the maximum byte count and the process descriptor of the calling process is used to obtain the system path number of the process' standard error in register A. The separation of user and system state paths is an important concept to understand; however, we will discuss it in detail in another article. For now, let's continue analyzing the code.

The I$WritLn system call then prints the string at register X to the caller's standard error path. If on the other hand, register X contains a zero, then room is made on the caller's stack for the default message, which is then copied into the caller's address space using the F$Move system call. The moving of the default message from the system address space to the caller's address space is necessary due to the separation of a process' address space in OS-9 Level Two.

Once the module has been compiled, it should be included in your OS-9 Level Two bootfile. Reboot with the new bootfile, and the OS9p2 module will find OS9p3 then jump into the execution offset (the Cold label in this case). This will install the F$SAYHI system call and make it available for programs immediately.

4. Installing a System Call in OS-9 Level One

Listing 2 is similar to the code in Listing 1, except that the code to move the default message from system space to the caller's address space has been removed. Also, the code to install the system call has changed, and the module type is not of type Systm, but instead of type Prgrm. This is due to the lack of separation of address space in Level One, which makes writing system extension modules much easier than in Level Two.

The common address space between the system and all processes in OS-9 Level One also makes the F$SSvc system call available from user state as well as from system state. Unlike OS-9 Level Two, where the system extension module must be placed in the bootfile, installing a system extension in OS-9 Level One takes a different approach. Placing a module called OS9p3 in an OS-9 Level One bootfile will NOT cause the system extension to be called because there are no provisions for that in the kernel. Instead, system extensions are installed by creating a module of type Prog that contains both code to install the system call and the system call itself. Installing the system call entails executing the module from the command line.

Besides the sayhi.a source in listing 2, another example of the this is the Printerr command that comes with OS-9 Level One. This is a program that actually installs a newer version of the F$PErr system call. To install the new system call, you simply run Printerr from the command line. It then installs the call and exits. There is an advantage to OS-9 Level One's approach to installing system calls: it can be done at run-time without making a new bootfile and rebooting the system. However, additional care must be taken not to unlink the Printerr module from memory. Why? Because the code for the replacement F$PErr call is in that module, and if the module is unlinked, the memory it occupied is made available subsequent reallocation and at some point, a system crash will ensue.

5. Exercising Our New System Call

Listing 3 is a small assembly language program, tsayhi, which calls the F$SAYHI routine. It will work fine under both OS-9 Level One and Level Two. If you fork the

`tsayhi` program without any parameters, then the `F$SAYHI` system call is called with register X set to $0000, which will cause the system call to print the default message. Otherwise, you can pass a message on the command line as a parameter and up to 40 of the message's characters will be printed to the standard error path.

6. Summary

Extension modules give us an effective way of altering the behavior of OS-9 by allowing us to add a new system call or modify the behavior of an existing one. Writing extension modules requires an extremely good understanding of the internals of OS-9. The particulars of writing a system extension vary under OS- 9 Level One and Level Two primarily due to the differences between memory addressing.

Example 1. Source for os9p3.a for OS-9 Level Two

```
Type     set   Systm+Objct
Revs     set   ReEnt+1
         mod   OS9End, OS9Name,Type,Revs,Cold,256
OS9Name  fcs   "OS9p3"

         fcb   1                       edition number

         ifp1
         use   /dd/defs/os9defs.l2
         endc

level    equ   2
         opt   -c
         opt   f

* routine cold
Cold     leay  SvcTbl,pcr
         os9   F$SSvc
         rts

F$SAYHI  equ   $25

SvcTbl   equ   *
         fcb   F$SAYHI
         fdb   SayHi-*-2
         fcb   $80


SayHi    ldx   R$X,u
         bne   SayHi6
         ldy   D.Proc
         ldu   P$SP,y
         leau  -40,u
         lda   D.SysTsk
         ldb   P$TASK,y
         ldy   #40
         leax  Hello,pcr
         os9   F$Move
         leax  0,u
SayHi6   ldy   #40
         ldu   D.Proc
         lda   P$PATH+2,u
         os9   I$WritLn
         rts

Hello    fcc   "Hello there user."
         fcb   $0D
```

```
         emod

OS9End   equ   *

         end
```

Example 2. Source for sayhi.a for OS-9 Level One

```
Type     set   Prgrm+Objct
Revs     set   ReEnt+1
         mod   OS9End, OS9Name,Type,Revs,Cold,256
OS9Name  fcs   "SayHi"

         fcb   1                       edition number

         ifp1
         use   /dd/defs/os9defs.l1
         endc

level    equ   1
         opt   -c
         opt   f

* routine cold
Cold     equ   *
* The following three instructions are important.  They cause the link
* count of this module to increase by 1.  This insures that the module
* stays in memory, even if forked from disk.
         leax  OS9Name,pcr
         clra
         os9   F$Link

         leay  SvcTbl,pcr
         os9   F$SSvc
         bcs   Exit
         clrb
Exit     os9   F$Exit

F$SAYHI  equ   $25

SvcTbl   equ   *
         fcb   F$SAYHI
         fdb   SayHi-*-2
         fcb   $80

* Entry point to F$SAYHI system call
SayHi    ldx   R$X,u
         bne   SayHi6
         leax  Hello,pcr
SayHi6   ldy   #40
         ldu   D.Proc
         lda   P$PATH+2,u
         os9   I$WritLn
         rts

Hello    fcc   "Hello there user."
         fcb   $0D

         emod
OS9End   equ   *
         end
```

Example 3. Source for tsayhi.a

```
Type     set   Prgrm+Objct
Revs     set   ReEnt+1
         mod   OS9End, OS9Name,Type,Revs,start,256
OS9Name  fcs   "TSayHi"

         fcb   1                        edition number

         ifp1
         use   /dd/defs/os9defs
         endc

level    equ   2
         opt   -c
         opt   f

F$SAYHI  equ   $25

* routine cold
start    equ   *
         lda   ,x
         cmpa  #$0D
         bne   SayHi
         ldx   #$0000
SayHi    os9   F$SAYHI
         bcs   error
         clrb
error    os9   F$Exit

         emod

OS9End   equ   *
         end
```