

## CONTENTS

1	1.	Introducing Ash.....
2		Installing ASH.....
2	2.	What commands work with Ash?.....
3	1.	What about Run?.....
3	2.	Shell features.....
3		Resident programs.....
4		Builtins.....
4		IO Redirection.....
5		Alias.....
7	1.	Summary of Alias syntax.....
7		Script Language.....
8		Environment Variables.....
10	1.	Special Variables.....
11		Pipes and PipeLines.....
12	1.	Show me more pipelines.....
13	2.	Show me how to use pipes with scripts.....
15		Command Substitution.....
18		Prompt enhancements.....
19		Using Variables and Substitutions in Prompt and Alias.....
21	3.	Modifying your startup sequence to use ASH.....
22		Modifying 1.3 startup to take advantage of ARP.....

- 22 4. Appendix A: Finding a pipe device to work with ASH .
- 23 5. Appendix B: Bug reports and enhancement requests....

Introduction to ASH: The ARP Shell.

## **AmigaDOS Resource Project**

Description of shell features, variables, batch language, pipes and command substitution. Includes sample scripts and command lines.

### **1. Introducing Ash**

The ARP Shell or 'Ash' is an alternative to the standard CLI or 1.3 AmigaDOS shells. While not claiming to be the last word in shell technology, we feel that it provides a significant improvement over the customary shells in the following areas:

All improvements introduced by Commodore in the 1.3 Shell are supported by ASH in either an identical or highly compatible fashion. This includes the Alias function, auto-executing batch files and Prompt enhancements. ASH runs both BCPL and non-BCPL programs.

-Ash contains many builtin functions which save disk access as well as disk and memory bytes. In particular, the entire batch language is built into Ash, which provides for much more efficient execution of scripts.

-It is possible to replace many of the C: directory commands with their builtin equivalents in ASH, and it is interesting to compare the sizes involved. The original 1.3 commands which ASH contains as builtins total over 15,500 bytes! By contrast, ASH itself is less than 8,000 bytes. To see a list of builtins, type a single question mark at the shell prompt, then hit return.

-Ash supports true concurrent pipes and command substitution, as well as environment variable expansion. These are advanced features not available in the 1.3 shell.

-Ash uses the arp.library process functions to run your programs, and so uses the ARP resident features. These can save you greatly in memory use, since the stack size and data size are encoded inside the program. ARP resident programs are also much safer than the 1.3 Resident, because arp.library checks the program before running it to make sure it has not been

damaged, and is residentable. This means that you do not have to crash your computer to discover whether or not a program will work as resident.

## **Installing ASH**

If you have ASH as part of the ARP distribution, the ARP install program will handle this for you. Otherwise, you should copy ASH to the L: logical directory on your system disk\*.

If you are using the ARP supplied AShell program you should delete the line in the Commodore supplied startup sequence which reads something like:

```
Resident CLI L:Shell-Seg SYSTEM pure add; activate Shell
```

since AShell will locate and load ASH for you. If you choose to use the Commodore supplied NewSHELL program and still want to use ASH, you want to replace L:Shell-Seg in the above line with L:ASH:

```
Resident CLI L:ASH SYSTEM pure add; activate arp shell
```

The only difference here is that the Commodore NewSHELL command will not automatically load the shell into memory if it is not present, and so requires extra work in the startup script. (Note - you may also need to set the PURE protect bit in L:ASH to do this).

## **What commands work with Ash?**

As far as we know, all programs work fine with Ash, including BCPL programs as distributed by Commodore. The only command that you won't want to use with Ash is the Commodore Resident command, you should use Arp's ARes instead. Alternatively, if you prefer the Commodore shell, do not replace Resident with ARes.

Each shell uses a different method of managing the resident lists, and so will work only with its own Resident loader. You won't crash if you mis-match these two, but you won't be getting any use from the resident features of the shell of your choice either.

---

\*The ARP Install program won't change your startup script for you. If you intend to use Ash, you should alter your startup-sequence as described to avoid wasting memory.

## 1. What about Run?

Run (either ARP's or Commodore's) just passes your command line to the currently loaded shell. If you have loaded the Commodore shell, then using either the ARP Run or Commodore Run will invoke the Commodore Shell. If you have loaded Ash, then using either Run command will invoke the Arp shell. This means that your background processes invoked with Run (or by the AmigaDOS function call Execute ()) will take advantage of resident programs and any additional features of whatever shell is loaded.

## 2. Shell features

We cover all features of the Arp Shell here, including those which are shared by the 1.3 Commodore shell. This is done for completeness, and also gives us the opportunity to describe any differences in operation between the two. Sometimes you might execute a command that Ash needs more input to complete. If you do, then Ash will print a prompt (a single '?' mark), and wait for you to type more (remember to hit return). If you can't seem to feed the shell what it wants, type an EOF character (Control-\\), and you will usually receive some kind of message that should allow you to figure out what's going on for next time.

### Resident programs

Resident programs save you memory and loading time, even over the use of a ramdisk, by letting the system reuse the same code for multiple invocations of the program. Due to limitations of the current Commodore implementation of Resident, you cannot mix the Arp ARes command with the Commodore Resident command. You need to use ARes with Ash, or Resident with Shell-seg (the 1.3 shell). You won't crash if you mismatch these programs, but you won't be able to use Resident programs either.

To make a program resident, simply type:

```
ARes programname
```

Then, the first time you attempt to access 'programname', the system will load the program for you, and make it resident. Each subsequent access will be much quicker, since it will run directly from memory. This 'load on demand' feature is nice,

since you don't have to wait for all those programs to be made resident in your startup sequence. Also, if you do not use a program during a session, it is not loaded at all, thus saving you memory.

Arp programs (and programs written using the Arp conventions) will save you greatly in memory usage. Each program requests only as much stack as it needs (whether from disk or from resident), and can save many many bytes per-invocation on this alone. Arp resident programs are also smart, doing any data cloning required only if resident, diskloaded copies don't need to be cloned, and so they are not.

All programs which work on the Commodore list should also work on Arp's, although they won't save you as much memory as Arp's will, you can still take advantage of the load-on-demand and checksum safety factors of the Arp resident\*.

## **Builtins**

As mentioned above, the shell contains many commands already builtin to ASH. To find out which ones, use the '?' command (i.e., type ? and then hit return). A list of all builtin commands will be displayed. See the appropriate section below for a description of the commands and their descriptions.

## **IO Redirection**

You can redirect program input and output by using the '<', '>' and '>>' operators.

- 1: Type <file ; reads from file
- 2: Type >file ; creates/overwrites to file.
- 3: Type >>file ; appends/creates file.

---

\*If you want fast error message reporting, you can make the Fault program resident by doing 'Ares C:FAULT'. This will force all error routines in ASH to use the resident version of fault. This won't work with the Commodore shell, incidentally.



When the '>' operator is used, the file is created if it does not exist. If it does exist, then all its former contents are lost. When the '>>' operator is used, if the file already exists the contents are *\*not\** lost, new output is appended to the end of the old. If the file does not exist, then it is created. You can use '>>' to safely output to files (for example, in shell scripts) without worrying about destroying existing files\*.

Note that even scripts can be redirected (either in or out). See the section on scripts for more information.

Another type of IO Redirection is called piping. It is important enough to deserve a special section of its own, later on.

## **Alias**

An Alias is a way of renaming commands in the shell without actually changing the name of the command on the disk. This is generally better than renaming the command, since if you rename your command, other peoples scripts and programs that reference the command might not find it. A common use of alias's is to give commands shorter names that are easier to type, here are a few:

```
Alias cp copy
Alias ls List
Alias rm Delete
Alias s Status Full
```

Now when you type 'cp', you will get the Copy command, but with the name you defined. Other uses for Alias's are to provide special names for commonly used functions. For example, the ARP 1.3 Assign command provides a special switch which will check to see if a name is already specified as a logical device - its syntax is:

```
Assign NAME: Exists
```

---

\*This differs from the Commodore shell. Under the Commodore shell, if the file does not exist the append ('>>') operator will not create the file for you, it will abort with an error message.

which checks to see if NAME: has been assigned, and displays its assignment if found. This is a common option, and you can define an alias to shorten the typing and give it a meaningful name like this:

```
Alias Exists Assign [] Exists
```

Notice the square brackets? This is a directive to the shell which tells it to replace the brackets with any command line input you supply when you invoke the alias. For example:

```
Exists S:
```

will cause the shell to evaluate the command 'Assign S: Exists'. Notice that the S: has replaced the square brackets in the command? Some other useful alias's are listed below:

```
Alias Today     list [] Dates Sort Since Today  
Alias Recent   list [] Dates Sort  
Alias PathList list [] lformat "%S%S"  
Alias Update   Copy [] Flags cgo  
Alias Pri       ChangeTaskPri
```

Another use of Alias is to override the shell's builtin commands with a disk based version of the same name. You might want to do this because you prefer another command's behavior, or it could be that a bug could crop up in one of the builtin commands making it necessary that it be replaced. To do this, simply define the command with the full path name:

```
Alias Endskip C:Endskip
```

this will force the shell to use the version of the command stored in the C: directory, rather than the shell's internal version.



## **1. Summary of Alias syntax**

To sum up then, you can define a new alias as follows:

*Alias AliasName Body*

where AliasName is the name you want to use to refer to your alias, and Body is the command you wish to be executed. Body can contain any legal command, and can optionally contain square brackets ('[]') which tell the shell where to place additional input from the command line.

To see the current alias's, type Alias by itself. To remove an alias definition, type Alias AliasName. For example:

*Alias Exists*

will remove the alias definition for Exists. If you want to redefine an alias, it is not necessary to remove it first.

## **Script Language**

Ash contains the complete AmigaDOS batch language internally. This makes scripts with conditionals (IF ELSE) and SKIP commands execute much more quickly. You can also save a lot of disk space by deleting these commands from your disk altogether. Note however, if you have conditionals in your startup-sequence, you will have to modify your startup-sequence to take advantage of ash's builtins, so please don't Delete anything until you are sure things are working correctly.

You can execute any script from the command line by simply typing its name. Note that to do this, the 'S' and the 'E' bits must be turned on (use the protect command for this). The 'E' bit tells the shell the script is executable, and the 'S' bit tells the shell that it is a script, not a binary executable. To make a script executable from the command line, you would use a command line like this:

*Protect Scriptname +se*

To prevent a script from auto-executing, we recommend you just clear the 'e' bit, since you can use the 's' bit in the listings to remind you that it is a script file, and not a program.



Scripts executed in this way appear indistinguishable from programs, and you can use the i.o. redirection operators (<, >, and >>) to divert the input or output to or from files. Scripts can also be used as the source or the sink for pipes. See the section on piping for more information.

Other improvements to the script language include much more informative error messages and regular handling of parameter expansion including <\$\$>. Many irritating little bugs (such as not recognizing tabs as whitespace) in the current Commodore batch language have been fixed as well.

When using scripts with arguments (the .key directive and the bracket characters), we recommend that you always change your bracket characters to something other than the default using the .BRA and .KET directives. These were chosen badly by the original author of the script language since they conflict with the IO redirection operators. Perhaps the best choice are the curly braces, since these are not used for anything else currently.

## **Environment Variables**

The shell expands environment variables that you supply on the command line. To expand a variable, you use the dollar ('\$') character immediately before it. Here is an example:

```
Echo "$INCLUDE"
```

If there is an environment variable defined by the name INCLUDE, the shell will replace it with value of that variable, and the echo command will display it. Note that the quotes are not necessary for the variable expansion, but they are necessary for Echo, which doesn't like spaces in its input (the variable could expand to a value with spaces in them).

To set the value of an environment variable, just use the '=' character:

```
FOO=Bar            ; assign a value to FOO  
Echo $FOO          ; display the value  
Bar               ; echo's output.
```



You can also concatenate environment variables easily enough like this:

```
FOO="$FOO"Fly
Echo $FOO
BarFly
```

Note that the quotes around the initial environment variable are necessary for the shell to know when the variable ends. If you use the variable at the end, these quotes are not needed, since the dollar sign takes care of delimiting the variable. It follows that if you want to concatenate variables, you can do it as follows:

```
Echo "$FOO$FOO$FOO"
```

This will print three copies of the value of foo. Of course, you can also use this in an assignment statement.

To remove an environment variable, give it a null assignment:

```
FOO=
```

will remove FOO from the current environment\*.

---

\*Environment variable expansion in the shell reads both the old Manx/Rockiki/Arp environment variables and the new Commodore

ENV: definition, in that order. However, setting environment variables from the shell writes only the older environment implementation, it does not affect ENV:. If you wish to do this from the shell, contact ARP central for a patch to arp.library. You can always use the SetEnv program (ARP's or Commodores) to write to ENV:.

## 1. Special Variables

Each invocation of the shell has three environment variables which are local to that shell, and can be used by you to discover special state information about the shell. These are \$, ? and ?2. The \$ variable contains the current CLI number, ? contains the return code of the last command and ?2 contains the operating systems result2 error. Note that ? and ?2 are changed each time a command executes, so if you want to use them, you should assign their values to some other variable so that you can preserve them. Here is an example of a script which exits with an error message and reports the return code:

```
; This file fragment illustrates the use of ? and ?2
Command args      ; execute some command
RC=$?             ; get return code
R2=$?2            ; and save result 2

IF NOT VAL $RC EQ 0
  Echo "Command exited with error $RC result2 $R2"
ENDIF
```

The variable '\$' corresponds to the bracketed expression <\$ \$> introduced by Commodore in 1.3. Generally, it is preferable to use \$\$ without brackets to allow the shell to expand it without writing a temporary file (as is required by the Commodore convention). Ash supports both. This is useful in making temporary files or assigns or whatever, since it is unique to each CLI invocation, although concurrent shell processes in ash do share the same number, so you should be careful when piping to and from scripts and using this variable. Here is an example that displays all three values of these variables in an Echo command:

```
Echo "rc = $? Result2 = $?2 CLI# = $$"
```



## **Pipes and PipeLines**

Pipes are a way of connecting programs together without using temporary files. As a simple example, say you wanted a line numbered listing of your current directory. One way to do this is:

```
List >tmpfile
Type tmpfile opt n
Delete tmpfile
```

First we create a tmpfile using the list command, then we type it using the line number option of type, and then we delete our tmpfile. Using pipes, this would all be accomplished as follows\*:

```
List | Type opt n
```

The '|' character is the pipe symbol. It causes Ash to connect the output of the list command to the input of Type. Aside from not using a temporary file, pipelines such as this are much faster than the usual single command execution, since Ash takes advantage of the multitasking nature of the Amiga to run each one at the same time. This means that List can be preparing more output for Type as Type is processing the last input received from list.

More than one command can be used in a pipeline. Here is an example which creates a line numbered hexdump from a directory listing:

```
Dir | Type opt h | Type opt n
```

The basic idea behind pipes is to combine commands which do one thing with other commands which do something else to get the results you want. In the above example, List doesn't create line numbers, while type does. By combining their capabilities, you create a super list that also does line numbers. (Note that you must separate the '|'s from the commands by whitespace. While this differs from the implementation of pipes on other operating systems, it is necessary on the Amiga due to the use of '|' in wildcard patterns).

---

\*In order to use pipes with ASH you must have a pipe device. See Appendix A for information on how to obtain and mount a pipe device that will work with ash.

Once you get accustomed to the idea of pipes and connecting programs, you will wonder how you ever got along without them. In the pipelines above, the Type program acts as something called a 'filter'. A filter is any program that takes something as input, does something to the input, and outputs this altered input. In order to act as filters, programs must be able to read from their standard input (keyboard) and display to their standard output. Of the standard set of commands that comes on the Workbench disk, useful filters are Sort, Type and Search. Note that only the ARP programs can be used in pipe lines, the Commodore programs have bugs that prevent them from using the keyboard as input\*.

## 1. Show me more pipelines

The Dir and List programs have a lot of options (they probably have too many options, but that's a story for another time). One thing you can't do with List is display programs that match a certain set of protect bits, for example, you might want to see all the files in a directory which are scripts and executable. To do this, you can use a pipeline such as the following:

```
List | Search STDIN " ?s????e? "
```

Note that when you use Search in a pipeline, you must supply the STDIN keyword in caps as the filename. The search string was made by examining the usual list output, and masking out the protection bits we aren't interested in with the '?', which matches any character:

```
-s--rwed      (List output)
?s????e?      (our search string)
```

---

\*Some people might like this. One person's bug is another's feature.

Another useful example utilizes a pipeline to determine how well commented 'C' language files are by combining Search and Type:

```
Search *.c /* | type opt n
```

The idea here is to Search through all the files ending in '.c' in the current directory, looking for a comment start sequence. (Notice that we have to use the single quote to escape the star, which has a special meaning to Search as a pattern. The single quote tells search to regard the star as just a character, not a pattern.). Once Search has found the commented lines, we pass them on to Type to count. The highest numbered line will be the total number of comments found in the files. We don't use the STDIN keyword here, because Search is not reading from a pipe, it is acting as the source (beginning) of a pipeline.

There are many utilities in the public domain and elsewhere that can be used as filters, and since filters are so easy to write, they are very convenient to create as you need them.

## **2. Show me how to use pipes with scripts**

Combining scripts in pipelines provides a convenient way to create new

commands from old, the applications are limitless. One problem shared by many people is updating the software on their disks when a new revision comes out. Often only a subset of the distribution is used, or perhaps a different organization of the software on the disk. In any case, it rarely works to do a straight copy. Here is a solution in the form of a script that uses pipes and the new Arp Read command to copy only those files which already exist in the destination directory from the source disk:

```
.Key Dest/a
;Update script: CD to directory with new files, then do
;List nohead quick | Update dest:dir/
```

```
Lab Loop      ; loop back to here
Read Filename  ; read the filename from pipe
```

```
if $? EQ 1 VAL ; Read returns 1 on eof
Echo "Copy complete!"
```

```
Quit 0
Endif

if Exists <Dest>$File ; If we have an old copy
    Echo "Copying $File to <Dest>$File"
    Copy $File to <Dest>$File ; copy to new
Endif

Skip Back Loop
```

Another way to do the same thing is to use the output of the List command with its LFORMAT option to send the commands to the shell. (You can send commands to the shell by piping them into the builtin command Execute). Here is an example which does the same thing using list and LFORMAT, if more quietly:

```
cd destdir      ; change to dest
list srcdir lformat="Copy %S%S to %S" | Execute
```

Read is an interesting and useful command. It can be used to read from the keyboard as well as in a pipe, and it also does a certain limited parsing on its input. For example:

*Read First Last Middle*

Will break the input up on quoted lines or (if no quotes) on white space. The first variable gets the first portion of the input, the second gets the second, and the last variable gets whatever is left.

Here is an example using List and Read to do an interactive version of type:

```
; Interactive type: Usage: List nohead | intertype
```

*Lab Loop*

*Read Filename Size Protection DateTime*

```
if $? EQ 1 VAL
    Echo "No more files"
    Quit 0
Endif

ask "Type $Filename ?"
If WARN
    Echo "======"
    Echo "File $Filename last modified on $DateTime"
    Echo "======"
    Type $Filename
Endif

skip back loop
```



Notice how we use the capability of Read to break up lists output to display selected information about the file in a custom header. Many more custom scripts are possible, and are often easier to write the custom programs to do the same thing. If you are not a programmer, but are familiar with the commands available on the Amiga, you should also have little difficulty in customizing your environment.

### **Command Substitution**

Another advanced feature which Ash makes available to you is Command Substitution. This feature allows you to replace a command with its output, and it works like this:

```
Echo "Todays date is $(Date), have a nice day!"
```

When the shell sees the sequence '\$(Command)', it runs whatever\* Command is (Date, in the example above), and then replaces the entire string with the output of Command, the parens and dollar sign do not appear in the output at all. One possible outcome of the command line above is:

```
Todays date is Friday 16-Mar-89 10:57:12, have a nice day!
```

You can nest command substitutions, and the substitutions can be any shell command, including a pipeline. Whatever comes out of the end of the pipe is what the shell will use in place of the command. Here is an example showing nested command substitution:

```
Echo "The current time is $(Echo "$(Date)" len 8)"
The current time is 10:57:12
```

---

\*This is implemented internally using pipes, so you will need to have a pipe device mounted to use this command. Please see Appendix A for information on pipes that will work with ash.

This example uses the ability of Echo to take a substring of its input. First Date is expanded, and then the inner echo is expanded, and eventually the output of this Echo replaces the entire substitution request and the final Echo is evaluated. This is an extremely useful ability, one which often eliminates the need to write temporary files. Here is an example of a loop counter using Eval and command substitution in a script:

```
.Key loop
; Demo loop
.Default loop 10
Counter=<loop> ; initialize counter

Lab start
Echo "Loop $Counter"

Counter=$(Eval $Counter - 1)

if val $Counter GT 0
    skip back start
endif

Echo "All done"
```

The important line is the one containing Eval. Eval's default behavior is to print its result, so we can use it in a command substitution expression to update the value of the variable Counter.

Another use of environment variables involves the Arp programs Basename and TackOn. Basename takes a pathname and extracts the filename portion of it:

```
Basename DF1:include/exec/types.h
types.h
```

You can also give Basename a suffix to delete. For example, to change a file ending in .c to one ending in .o, try this:

```
Echo "$(Basename file/program.c .c).o"
program.o
```

TackOn takes a pathname and a filename, and tacks the filename onto the end of the pathname\*.

---

\*You could do this with normal variable concatenation except that some pathnames contain a ':' character. TackOn handles this condition more efficiently than a shell comparison could.

Ash Users manual.    V1.3.0            March 30, 1989

```
Tackon DF1:include exec
DF1:include/exec
```

```
Tackon DF1:include/exec types.h
DF1:include/exec/types.h
```

You can use these two commands to make your scripts much more robust. Here is another example of the update script given earlier. This one handles filenames much better, and also has a better user interface:

```
.Key Dest/a
; Update script: version 2.
; usage: pathlist | update dest
;

Lab Loop
Read Sourcename ; read source filename from pipe

if val $? EQ 1
    Echo "Copy complete!"
    Quit 0
ENDIF

tmp="$(Basename $Sourcename) "
Destname="$(Tackon <Dest> $tmp) "

if Exists $Destname
    Echo "Copying $Sourcename to $Destname"
    Copy $Sourcename to $Destname
Endif

Skip Back Loop
```

Notice that this uses the alias PathList shown earlier to generate full pathnames for the pipe. This version of the update script is much more robust than the earlier one, since the user can type the destination directory in a number of ways, and does not have to CD to the source directory he wants to update from. Again, the interesting lines are where the destination filename is generated. It is quite simple, first we use Basename to extract the filename from the piped input, and then we use TackOn to append it to the destination directory name. You can take advantage of the fact that command substitution nests to

eliminate the temporary variable by doing:

```
Destname="$ (Tackon <Dest> $(Basename $Sourcename)) "
```

Another useful thing you can do with command substitution is to use it to generate a list of filenames for a command that does not accept wildcards. There are many programs available in the public domain and elsewhere that accept multiple commands but do not let you use wildcards. To generate a list of files for them to use, you can use the statement:

*Program "\$ (Dir) "*

Of course, you can use any of the options to `Dir`, or the `list` command to generate exactly the file list you want. To check on your file list before passing it to the program you can do:

*Echo "\$ (Dir) "*

first. Note that there is currently a limit of 1000 characters to the total length of the argument line in ash (the total command line length is 1256 characters, including the name of the command).

### **Prompt enhancements**

Ash supports all the prompt enhancements introduced by Commodore in their 1.3 shell. You can use `%S` in the prompt definition to display the name of the current directory, and you can use `%N` to display the current CLI number. Ash supports all these, as well as `%P` which acts the same as `%S*`. All of these can be in upper or lower case.

---

\*This was done to be compatible with earlier releases of the ARP software. Until Commodore released the 1.3 shell, one of the only ways to get the current directory string in your prompt was with ARP's Prompt command, and it used %P.

You can also have the prompt expand environment variables and utilize command substitution in prompt lines as explained below.

### **Using Variables and Substitutions in Prompt and Alias**

The shell will handle environment expansion and command substitution as well as pipes in prompt strings and aliases. The methods for including these in each one are the same, which is why they are discussed together here.

You can use command substitution in your shell prompt to do things like display the current time by using the Date command. Initially, you might try something like this:

```
Prompt "$ (Date) "
```

which would appear to work. In fact, you might not notice it didn't work until the morning, when the bright light of the sun came pouring through the windows of your computer room. What you really need to do is this:

```
Prompt "\$ (Date) "
```

Notice the escape character before the \$? This tells the shell to delay evaluation of the prompt string until the prompt is actually displayed\*. Without the escape character, the date command gets expanded immediately, and so the time in your prompt gets set permanently to whatever it was when you issued the prompt command. If you want to            only see the time portion of the string, you can do it like this:

```
Prompt "\$(Echo len 8 \"\$ (Date)\")\n> "
```

This might look a little daunting, but most of the escape sequences (with the exception of \n, which is there for cosmetic purposes to print a newline as part of the prompt), are there to force delayed evaluation on the shell. Quotes are necessary around \$(Date) because the 1.3 Echo requires them. This means they must also be escaped, otherwise, the prompt command will regard them as delimiters and give you a "too many arguments" error.

---

\*These examples use the \ character for an escape character. If you have another preference, or if you use the default ('\*'), then just replace the backslashes in this section with the escape character of your choice.

The same thing is true as far as environment variable expansion is concerned. It is interesting to be able to display the return codes from the programs you run in the shell prompt. This is easy to do using the escape trick shown above:

```
Prompt "RC = $? R2 = $?2\n> "
```

Any environment variable can be displayed in this manner.

Alias's are similar. In fact, aliases are a little clearer, because you can see what is actually stored there by typing Alias without any arguments. Here is an example that might clarify the above discussion:

```
Alias ThisTime Echo "$(Date) "
```

```
Alias CurrentTime Echo "\$(Date) "
```

If you now type 'Alias' <ret>, this is how the display looks:

```
Alias <ret>
CurrentTime=Echo "$(Date) "
ThisTime=Echo "Saturday 18-Mar-89 11:49:31"
```

Notice that the ThisTime definition contains the expanded string from when the alias was defined. The CurrentTime definition, because we used an escaped dollar in its definition, contains a command that will actually display the current time when it is invoked\*.

---

\*Of course, if you really want an alias to just display the output of Date, this is much less efficient than just using Date as in "Alias dt Date".

You can use pipes inside of Alias's if you need too, but the same principles apply. You must escape all the pipe characters you use in order to prevent the shell from trying to pipe away immediately:

```
Alias Allc Dir [] opt a \| Search STDIN *.c
```

### **3. Modifying your startup sequence to use ASH**

Before trying to modify your startup-sequence in this way, please make a copy of your disk and work from that. Then, when you are sure it is working, you can make the changes on the main disk. We are not trying to scare you, but it is easy to make mistakes when doing things like this.

The main advantage to modifying your startup-sequence to use Ash is to take advantage of the faster executing conditionals language builtin to ash and/or some of the nice features of ash such as variable expansion or command substitution. If you don't use any conditionals in your startup-script, or don't want to use any of the Ash features in your startup-script, then you can skip this section, but please see the section on the 1.3 startup.

What you will want to do is break out a portion of your startup-script file into a special init file called shell-boot. (You can call it whatever you want, but don't call it shell-startup unless you want it to be executed whenever you invoke a new shell). Place most of your commands in this shell-boot file, and then let your startup-sequence file have the single command:

```
AShell From S:Shell-Boot
```

That's all there is to it!



### **Modifying 1.3 startup to take advantage of ARP**

In general, you can speed up your startup sequence file by combining more commands on one line. Most of the Commodore commands still do not accept multiple input arguments, while most of ARP's always have. By doing all of your assigns and mounts on one line you save a little bit of time.

If you are using ASH, then you should definitely remove all the Resident stuff from StartupII (on a virgin 1.3 workbench disk), and replace it with ARes command lines. Note that a lot of the things Commodore suggests you make resident are built into ash, so you shouldn't just do a global search and replace. Replacing Resident with ARes will dramatically speed up your startup-sequence times, since the Arp resident is 'load-on-demand'. The command will only be loaded when you invoke it, not when you use ARes (unless you use the Force keyword). This also saves you memory, since if you don't use the command it is not made resident.

### **4. Appendix A: Finding a pipe device to work with ASH**

The only pipe device known to work with ash at this time is PIP:, which is part of conman, a console handler replacement by William Hawes. If you have conman, you already have PIP, otherwise, in order to use the full features of ASH you will have to obtain a copy. Fortunately, this is easy to do, it is widely available on bulletin boards and from user groups as shareware.

Once you have installed conman as described in the conman documentation, make sure you have an entry such as the following in your mountlist file:

```
PIP:    Handler = L:ConHandler
Stacksize = 600
Priority = 5
GlobVec = 0
#
```

Once you have done this, you can place a mount pip: command in your startup-sequence, and start playing with pipes.

Note that the Commodore PIPE: device supplied with the 1.3 distribution does not work with ASH's internal pipes (using the '|' character). PIPE: uses only named pipes, ASH requires unnamed pipes internally. You can use both PIP: and PIPE: together if you need both named and unnamed pipes and we have also heard rumors of someone working on an enhancement to PIPE: to allow it to support unnamed pipes as well, but nothing definite is known about this new super pipe.

## **5. Appendix B: Bug reports and enhancement requests**

If you find a bug in Ash or any of the Arp software, it is important that you report it, so that we can fix it. Note that ARP is distributed without a Warranty of any kind, but we still want to fix our bugs. When you make a bug report, it helps if you can state at the top of the page which Arp program is the problem, this helps to route the report to the person(s) currently working on that aspect of the ARP project. It is also important to give enough information for us to reproduce the problem. Right or wrong, if we can't reproduce the problem we tend to assume it is operator error and not worry about it.

Enhancement requests are similar, we are very interested in requests for enhancements or ideas for new Arp software. Things we would like to add to ash (for example) are:

1. Complete subshell implementation. This is only partially implemented currently, pipes and command substitution run in subshells.
2. String operations on variables. Probably not csh model. Note that you can do a limited amount of this with Echo.
3. Enhancements to batch language, things like a for command, possibly a case statement, others.

