Hacked Shell+ v2.1 Documentation
For L-II CoCo-3 Only

By: Ron Lammardo

BUG FIXES:

Will now unload (unlink) the correct name. Previously, if the module name did not match the command filename you typed, shell would not unload the module, and it would stick in memory until manually unlinked. This shell reads in the actual module name and uses it instead of whatever was on the command line. Example: "/d0/cmds/bob" will work correctly now: the module name within the "bob" file will be the one to unload.

To prevent attempted execution of a write-only device as a procedure file, and to help with use of L-II window startup, this shell checks modes and attempts write of a null to std out when shell starts up.

You can redirect >>> to a write-only device. Before this, the shell would open the path in UPDATE mode.

The quote bug is fixed. That is, leaving off the second quote mark in lines like - format /d0 r "Disk - will not crash the shell.

The EX bug is fixed..if you type 'EX' with no parameters (to kill a shell) it will not attempt to fork any module or return an error to the parent process.

Standard error path is now 'un-redirected ' after using a pipe.

ADDITIONS:

NOBLOCK

The shell now runs in what is commonly called 'No Block Mode'. Under previous versions of shell / shell+ if you had a shell running on , say, T2 and you were running on Term there was no convenient way to send messages back and forth. If you did an 'echo whats new >/t2' from the Term it would wait around until the other shell recieved keyboard input. While it was waiting, Term would be hung up waiting for the echo command to complete. This problem was caused by the other shell doing a read call, thereby 'blocking out' any other input.

This release of Shell+ gets around that problem by putting itself to sleep while waiting for keyboard input, allowing other input to get through & display. Once the first key is hit for a line, the shell goes into its regular read routine until the <ENTER> key is pressed.

The noblock feature can be turned off by : modpatch <noblock.off.scr
and then saving the shell.

It can be turned back on by : modpatch <noblock.on.scr
and then saving the shell.

EXECUTION DIR SCRIPTS

A favorite feature of preliminary shell+ users, this allows global shell scripts to be placed in your execution (CMDS dir). Simply build or copy a shell procedure file to your cmds dir, then set the execution permission bits on it (attr script e pe). This makes it easy to add some commands that you can access almost all the time.

The shell search path becomes: memory, execution dir modules/scripts, data dir scripts.

Some people use this for printer setup cmds (using display >/p), others use it with the IF/THEN/ELSE and GOTO commands to make sophisticated procedures. I use it a lot for window commands, and program startups.

For example, I have a script in my cmds dir called "fsm". It opens a VDG screen on W6, and starts Flight Simulator. When I exit FS, it resets W6 back to a window type (all done in the background):

```
 * FSM - Procedure Command File in CMDS that starts FS-II.
 xmode /w6 type=1; display c >/w6
 chd /dd/games/fs
 (fs <>>>/w6; xmode /w6 type=80)&
```

PROMPTS

Settable and useful prompts. Ron had a great idea: why not add the device name you're working from to the prompt (OS9[T2]:)? Loved it, but everyone liked different method of presentation. So added user-settable prompt:

p=prompt	The prompt may be up to 21 chars, extra ignored. A space, return, or semicolon terminates the prompt. By putting the prompt in quotes, scan will continue until second quote, return, or 21 chars is reached. Options include: # - show shell's decimal proc id # (00-99). @ - show device name current std out $ - show current working directory (- show current date) - show current time More than one #,@,$,(,) is ignored.

Examples:	Prompt Resulting:
p=OS9/@:	OS9/Term:
p="Hi There!"	Hi There!
p=OS.#.@:	OS.06.T2:
p=$>	/d0>
p=") [@]:"	18:30:14 [Term]:
p=	<uses default setup in shell header>:

Default prompt- stored starting from offset $003D in this shell. As it comes, it's set to 'OS9[@]'. You may set using debug to something else. Terminate your prompt with a $20 (blank). Offset $0054 is the longest you should go before you run into the rest of the shell code. The longest settable prompt will stop at $0051 anyway (21 chars worth).

It's easy to start up new shells with custom prompts. Just pass it as param. To start a shell on say, /W4 with a special prompt, you might type:

```
'shell p=OS9/@?  i=/w4& ' ... results in shell on W4 with prompt 'OS9/W4?'
```

However, if you wish to use the # process id option as a shell parameter like the above, then since #!&; etc normally means something special, you must put the string in quotes, which are ignored by P=. Thus instead:

```
'shell p="Hello.#[@]" i=/w4& ' ... gives prompt of 'Hello.07[W4]'
```

PATH REDIRECTION

z={path} - Same as i=/1 except that the parent process is killed. This would be used in a shellscipt that does a chd,path=,or variable defining that you want held over when the shellscript ends.

r=[redirect chars]{path} - redirects the specified paths to the input path. All allowable path mnemonics are supported : <, >, >>, <>, <>>, <>>>, >>>

Examples:

```
r=>/w    - redirects std out to next available window
r=<>>>/w - redirects std in,std out, std err to next available window
r=</h0/shellscript - redirects std in from /h0/shellscript. If the last
                     line in the file was 'i=/1' , the shellscript would end
                     with all settings preserved. This method is preferrable
                     to the 'z=' or 'GPERE' methods as it does not involve
                     forking any additional shells.
```

MEMORY SCRIPTS

One of the big problems encountered by floppy disk users is 'where did I put that shell script ?'. The shell will now allow a data module, either resident in memory or from the execution directory (or from a full path list if specified) to be executed as if it were a text file containing a shell script. Using this method, you could convert all your commonly used shellscripts to data modules (See the DATAMOD utility) , pack them all into one file and load during startup. This would allow access to all those shellscripts without accessing the disk. And since shellscripts are generally very small, you should be able to merge quite a few into an 8K block.

RUNB BASIC09 PARAMETER SETUP

A common complaint, and perhaps the reason why more people don't write packed Basic09 commands, is that Runb requires any passed parameters to enclosed in parenthesis and quotes. For example, to execute a packed command, you might have to type: cgp220sd ("/term","/p"). A royal pain.

This shell recognizes packed procedures, and will do this automatically for you. Using the same example, you can just type: cgp220sd /term /p . The shell will still recognize parameters passed in the normal format: cgp220sd ("/term","/p").

I expect this feature to allow the user to go nuts making packed basic09 custom commands. All parameters of course must be strings. If you need to pass a number, simply take it in as a string and use the VAL command to convert it to a number.

Here's a quick and dirty program to pack. It allows you to see how things are passed. Examples:

```
parmtest 1 2 3
parmtest hello kevin darling
parmtest only two
parmtest more than three params
```

```
PROCEDURE ParmTest
PARAM a,b,c:STRING
DIM aa,bb,cc:STRING
ON ERROR GOTO 100
aa=a
PRINT "Param one=*"; aa; "*"
bb=b
PRINT "Param two=*"; bb; "*"
cc=c
PRINT "Param three=*"; cc; "*"
END
100 (* Come here on Param Error *)
PRINT "Less than three params given"
END
```

LOGGING

The logging feature is toggled by use of the L / -L commands. If logging is on (default is off) , any non-comment line will be written to log file /dd/log/uxxx where xxx is last 3 digits of userid prefixed by the date/time the line was processed. In addition, a line will be written when a shell starts or terminates (+++START+++ / +++END+++) .

Logging can be turned on permanently by : modpatch <loglock.on.scr and then saving the shell. To turn it back off (and enable the L / -L functions) use : modpatch <loglock.off.scr and then save the shell.

SHELL VARIABLES

The shell now supports up to 10 shell variables of up to 80 characters long each. These variables remain set as long as the shell is running, and can be used as all or part of a shell command line. The variables are loaded by :

```
var.1          - accepts up to 80 chars from std err to load variable %1

var.="......." - loads %0 variable with data between quote characters
```

An example might be a shell script (or MEM script !) used to call the assembler. Since most programs use the same standard commands, you could set up a generic script and supply the input when run as in the following:

```
* Comp - generic assembler call file
prompt Program to ASM :
var.0
var.3="/dd/output"
t
asm #16k %0 L O=/dd/asm/obj/%0 >-%3
errchk <%3
unload %0
load /dd/asm/obj/%0
```

When this is run, the following output will be executed:

```
Program to ASM ?    - displays on screen
testprog            - user types in
asm #16k testprog L O=/dd/asm/obj/testprog >/dd/output
errchk </dd/output
unload testprog
load /dd/asm/obj/testprog
```

To view all current variables, enter 'var.?' on the command line to produce:

```
User Variables :
var.0=shell
var.1=
var.2=
var.3=/dd/output
  .
  .
var.9=

Shell Sub Variables :
var.0=
  .
var.9=
```

To turn off variable expansion, use the '-V' option ('V' turns expansion on). Note however that variable expansion is performed before a line is processed. So if you have a line like '-v prog "50%"' the % will still be expanded for that line, but proceeding lines would not expand variables.

If a variable is referenced but not defined, no expansion takes place but the %# is removed from the input line.

As shown in the VAR.? example, there are two sets of variables. The first set, the User Variables, can be set & examined by the user. The second set, the Shell Sub Variables, can only be set by a ShellSub. These variables, however, can be used to load User Variables by specifying them as %%# .

Example : The Sdate shellsub returns the month name in the first shellsub variable. If we just want to display this on the current line we could use :

```
echo The current month is %%0
```

Or to run something in a certain month :

```
IF %%0=July
THEN
 .
 .
ENDIF
```

There is one additional variable that can not be set by the user. The variable '%*' will translate to the error code returned by the last line processed. This is extremly useful in conjunction with the on error command.

INC. / DEC.

Shell variables can be incremented / decremented by 1 using the inc.# / dec.# commands. Result will be a 5 digit number, right justified and zero filled. If you try to inc./dec. a variable containing a non-numeric field it will treat the initial field as 00000 and act uppon it accordingly.

Values can be in the range of 0 - 65535 rolling over in either direction. If you attempt operations on a value > 65535 it will roll over.

Examples :

```
var.0=1   sets to 1
inc.0     sets to 00002
dec.0     sets to 00001

var.8=002 sets to 002
dec.8     sets to 00001
dec.8     sets to 00000
dec.8     sets to 65535
```

PAUSE

Pause will display a message than wait for key press or mouse click. The message will be sent from the first non-space to the end of the line.

Example : pause 'Hit any key when ready' <- displays & waits for key/mouse

WILDCARDS

Seemed to be one of the most asked for enhancements , so here they are.
Wildcard expansion is performed AFTER variable expansion but before any other line checking is done. In order for wildcard expansion to take place the first character on the line must be a colon (':'), otherwise the line will be processed as is. This option can be reversed (: will prevent expansion) by:

```
   modpatch <wild.on.scr
```

and then saving the shell. To restore back, use: modpatch <wild.off.scr
and then save the shell.

The following characters are supported :

```
*     -  Matches any string of characters
?     -  Matches any one character
[a-z] -  Matches one character in the range within the brackets
```

Upper & lower case are treated equally - 'a*' matches ASM , aif.ctl , ...

To pass a wildcard character to a program,quote it by using the '\' chracter immediately preceeding the desired chracter.

Examples :

```
FSTAT *      - runs FSTAT on every file in the current data dir
FSTAT [c-g]* - runs FSTAT on every file beginning with c thru g
FSTAT she*   - runs FSTAT on every file starting with 'she'
FSTAT *.a    - runs FSTAT on files starting ending with '.a'
LS a\*       - passes 'a*' to LS for wildcard expansion
```

The expanded buffer size is 2048 characters. If the expanded line size is greater than this, error #191 will be returned (Buffer size to small).

A utility program 'PARAM' is included in this archive (source and binary) to reformat an expanded wilcard line. The program takes the parameter line passed to it and writes it to standard out , one word per line (similar to the LS command except that wilcard expansion is taking place in the shell).

```
Example : param she*                          <-- input
          param shell shell.io                <-- expanded to by shell

          shell                               <-- output line from PARAM
          shell.io                            <--   "           "
```

If desired, this output could then be piped to a utility requring paramerters passed one per line (e.g. - param * ! call "copy $ /d1/$"!shell).

OUTPUT APPEND AND OVERWRITE

Similar to OS9/68000. Great for appending to logs or help files, or merging modules. Or for using the same temporary filename by overwriting.

```
  >+filename   - also >>+ and >>>+. Appends output to end of [filename].

  >-filename   - also >>- and >>>-. Overwrites contents of [filename].
```

IF / THEN / ELSE / ENDIF / FI / CLRIF

Condition testing is supported as follows :

```
IF -Y   - read one char from std err : y=TRUE, n=FALSE
IF -F <file> - TRUE if file exists and is a file
IF -R <file> - TRUE if file exists and is readable
IF -W <file> - TRUE if file exists and is writable
IF -E <file> - TRUE if file exists in execution directory
IF -D <file> - TRUE if file exists and is a directory

IF ....<condition>....  where <condition>
IF +###<condition>####  where <condition> is one of the following :

  =   - TRUE if left side is equal to right side
  <   - TRUE if left side is less than right side
  >   - True if left side is greater than right side
  <= or =<  - True if left side is less than or equal to right side
  >= or =>  - True if left side is greater than or equal to right side
```

Without the + symbol in front,the two sides of the equation are compared character by character from left to right,with any unused characters being translated to nulls for comparison. If the + symbol is present, the two sides of the equation are right justified and zero filled, allowing for accurate numeric condition testing. If either side of the equation is greater than 80 characters or the test condition is missing , the command will error. Note that if error exit is turned off (by the -X command) processing will continue on the next line as if the IF statement was true.

Examples:

if a=b	false	if a=b	false
if a<=b	true	if a<>b	true
if hello=HellO	true	if what=what?	false
if what<=what?	true	if what<>what?	true
if 09<010	false	if 09>010	true
if +09<010	true	if +09>010	false

If a condition is true, following lines will be processed until an optional ELSE is encountered. Lines will be skipped until a ENDIF or FI is encountered.

If a condition is false, following lines will be skipped until an ELSE , ENDIF or FI is encountered.

The reserved word CLRIF can be used to clear any IF in effect, even if lines are to be skipped because of a false condition or between ELSE/ENDIF for a true condition.

If the input line echo option is on ('T'), the word TRUE or FALSE will be printing following the if statement depending on how the statement evaluates.

The condition being tested may be placed in brackets (e.g. - if [-y]) to maintain compatability with existing IF utilities.

IF statements can be nested up to 255 deep. The word THEN is optional and will be ignored if present.

GOTO

Goto [label] - searches for label from the beginning of the file.
Goto [+label] - searches for label from current file position on.

Label must be an alphanumeric word (up to 40 characters) and must immediatly follow a '*' comment marker. Label must exactly match the label on the goto line (upper & lower case are unique).

If a line contains '*\' as the first two characters, any goto in effect will be cancelled with processing continuing on the line following.

ONERR GOTO

Onerr goto [label]
Onerr goto [+label]

Same as GOTO except that it does not execute until a command errors. After the error, the GOTO command specified is performed. Note that while executing a GOTO, the error variable ('%*') is not updated, allowing for error identification at the goto label.

Entering ONERR alone on a line causes the on error trap to be cancelled.

As with the goto command, '*\' causes the goto to be cancelled.

Example:

```
onerr goto lab1
dir sd       <---- errors with #216
pmap         <---- not executed
mmap         <---- not executed
*lab1        <---- label matched
if %*=216    <----- expands to 'if 216=216' which evaluates true
goto lab2    <----- jumps to label lab2
else         <----- ignored because condition was true
echo ERROR   <-----          "             "
endif        <----- tells shell o.k. to process lines again
smap         <----- ignored cause we're going to lab2
*lab2        <----- found
.....        <----- processing continues here
```

PATH=

Path= allows you to specify alternate directorys to search for commands. If the current execution directory does not contain the desired module, the alternate paths will be searched in the order specified. This is especially useful for floppy disk users with limited space, as it allows you to have your CMDS directory spread across 2 or more disks. The specified paths are retained when subshells are forked, either by directly calling the shell (shell i=/w&) or by running a shellscript/memscript.

Pathlists must be separated by spaces only. To clear alternate paths, type 'path=' with no parameters.

To display the currently assigned paths, use 'path=?' .

Example:

path=/d1/cmds /d2/cmds /d2/newcmds

path=? <** produces
/d1/cmds
/d2/cmds
/d2/newcmds

SECURITY

If the userid is not 0, any '@' characters will be stripped out to prevent os9 security being overriden by being able to dump a disk. The potential problem with prompt setting (@=device name) is taken care of by allowing an [alt]@ to be accepted in addition to '@'.

If this feature is undesirable, it can be removed by 'modpatch <atcheck.off.scr' and then saving the shell. It can be turned on again by 'modpatch <atcheck.on.scr' and saving.

SHELL SUBS

A new type of module is supported by this release of Shell+ - the ShellSub module. These modules are asm subroutines which pass data back to the shell in the 10 reserved shell sub variables (%%0 - %%9). For instance, the Sdate shellsub (included in this archive) returns the current month name and year,month,day,hours,minues,seconds all in separate variables. These variables can then be examined and acted upon by the shell (possibly for displaying or IF/THEN/ELSE testing). Note again that the shell sub variables can NOT be modified by the user, only by shell subroutines. They can, however, be loaded into the regular user variable set for manipulation.

Technical notes :

When the shell determines that a ShellSub is to be run (by a $51 in the Type/Language byte in the module header) it loads the registers in a manner similar to those used when forking a module :

- X - parameter area start
- D - parameter area size
- Y - subroutine entry point
- U - first ShellSub buffer

It is the responsibility of the shellsub to ensure that each variable ends with a <cr> and start at the correct locations (10 variables of 81 bytes each). The shellsub must also end with an rts to return control back to the shell. The shellsub is therefore also responsible for ensuring that the stack points to the same place at exit as it did upon entrance. The shell has about 750 bytes available for stack use. All registers used by the shell are saved before calling the shellsub & restored after the shellsub returns.

USER STARTUP FILE EXECUTION

The reserved word 'S.T.A.R.T.U.P' will cause the shell to attempt running a file called 'startup' in the CURRENT directory. This is very useful when running Tsmon/Login and you are changing the users working directory. If the users directory is changed and then the program forked is 'shell s.t.a.r.t.u.p' the shell will try to run a startup file in the new directory and then remain active. If no startup file is present, no error is returned.

MISC

A command called PROMPT is included in this archive. It functions the same as the ECHO command except that it does not do a linefeed after displaying the line. See the example script under SHELL VARIABLES.

Also included in the archive is a shellscript called SHELLSCRIPT. This is what I use to assemble the various versions of Shell+. It is included as an example of variables,goto and if/then/else use. It is NOT needed to install or use Shell+.

You may modify the priority of commands using the "^" (up carat) symbol. Example: list file >/p ^100 &

Since all the coding for PWD/PXD is part of the shell for the $ option on prompt setting, the commands .PWD and .PXD are included with the shell. Once you do a .PWD or .PXD the directory will be kept in a buffer so that next time you execute the command it won't have to check the disk. It will also stay in the buffer if the $ option is used in the prompt. Note that if a pathlist is over 129 chars long, it will be printed prefixed with an '*', indicating that the beginning of the pathlist has not been printed. This also applies when using the working directory as part of the shell prompt. Note : Even if you use the .PWD/.PXD functions you should keep those commands in your execution directory, as other programs (notably basic09 routines) may use the shell to get the current directorys. It is for this reason that the commands are prefixed with a '.' .

CD and CX are also allowed for CHD and CHX for Nuxi people. Also '|' allowed along with '!' as pipe character.

Echoed input lines ('T' option) will now display the full input line. Why print only 80 chars if the input buffer is 200 ?

All modules are now forked with a minimum of 7936 bytes (8k - 1 page) to handle the greatly expanded parameter line size. If the module needs > 8k or a memory modifier > 8k (#16k) is specified, that size will be used.

If you find any bugs, please let Ron Lammardo (75706,336) know through the CIS OS9 forum, so that we can take care of them. Suggestions, bitches and kudos appreciated also.

Some of the enhancements in Shell+ came either directly from or were derived from existing utilities. The ones I can remember are :

Goto - Kevin Darling
Z= - from GPERE by Kent Meyers
Wildcards - Simmule Turner
NOBLOCK - Kent Meyers
Logging - Karl Krieder

Shell+ Release 1.0 by Kent Meyers
Shell+ Release 1.1 by Kent Meyers,Kevin Darling,Ron Lammardo

V2.0 additions/changes

- Current date/time displayed when shell starts

- Current date / time can be used as all or part of the shell prompt.

- Standard error redirection return fixed for pipes / EX bug fixed

- Memory scripts

- Shell Variables (VAR.# , %# , %*, VAR.? , V , -V) / INC. / DEC.

- PATH= command (PATH=?) / PAUSE command

- IF / THEN / ELSE / ENDIF / FI / CLRIF

- GOTO / ONERR GOTO / *\

- Wildcarding

- Path redirection (Z= , R=)

- @ removal for users other than user 0 / Logging (L / -L)

- Shell Subs (%%#)

- User Startup execution

V2.1 additions/changes

- Wildcard Quoting ('\' as escape character)

- Wildcarding rewritten (fix comment line errors,upper/lower match,range)

- Packed BASIC09 memory size error fixed

- Large parameter line into shell fixed