Engineering Support Processor (ESP)

604 User's Reference Manual

Document Dated June 7, 1996

(For the entire 604 family)

Kent D. Thompson
T/L 678-6708
(512)838-6708
AUSVM6(KENTTHOM)

John Bordovsky
T/L 678-5682
(512)838-5682
AUSVM6(BORDOVSJ)

Dept. F87/045-3
Austin, Texas

Hard copies of this document are valid on date printed. For the latest version of this document contact one of the authors. This document is also located in /afs/awd/public/esp/userdoc/esp604manual.ps.

FORWARD 11

PART 1 -- ESP DESCRIPTION 13

1.0: OVERVIEW OF THE ESP PROJECT 14

1.1: INTENT AND PURPOSE OF THE ESP PROJECT 14

1.2: BRIEF COP OVERVIEW 14

1.3: HARDWARE REQUIRED 15

1.4: SOFTWARE OVERVIEW 16

1.4.1: FUNCTION LIST 16

1.4.1.1: Basic Functions 16

1.4.1.2: Additional Supported Functions 17

1.4.1.3: Machine Under Test (MUT) Hardware Restrictions 18

1.4.2: STOPPING THE PROCESSOR 18

1.4.2.1: Hard and Soft Stops 18

1.5: SOFTWARE INSTALLATION 19

1.5.1: COMMAND FILES 19

1.5.1.1: Definition Sequence 19

1.5.1.2: Editor producing 'flat file' 19

1.5.1.3: profile.x 20

1.5.1.4: menu.x 20

1.5.1.5: screen.x (Screen Definition) 20

1.5.1.6: chipst.x (Scan Table) 20

1.6: RUNNING THE ESP PROGRAM 20

1.6.1: HOW SCREENS SHOW UP IN THE PROGRAM 21

2.0: USER INTERFACE 22

2.1: The ESP CONSOLE 22

2.1.1: Options 23

2.1.2: Custom Menus 24

2.1.3: Quickscript 24

2.1.4: Refresh 24

2.1.5: Help 25

2.1.6: Log 25

2.1.7: Utilities 25

2.1.8: Command History List 26

2.1.9: Command Line 26
2.1.10: TTY Window 26
2.1.11: Machine Under Test Status 26
2.2: SCREENS 26
2.2.1: WHAT A SCREEN IS AND HOW TO MAKE ONE 27
2.2.1.1: An example Screen Definition 28
2.2.1.2: Start rows and columns 28
2.2.1.3: Data TYPES defined 28
2.2.1.4: Size of screen DATA cells 29
2.2.1.5: References to CHIPs and FIELDs 29
2.2.2: HOW TO CALL UP A SCREEN 29
2.2.2.1: Nicknames 29
2.2.2.2: Full Titles 29
2.2.2.3: The 'screen' command 29
2.2.2.4: By name on the command line 30
2.2.2.5: Menu invocations 30
2.3: KEYBOARD / EDITOR 30
2.3.1: CURSOR MOTION 30
2.3.2: EDITING A CELL 31
2.3.2.1: Originating the edit 31
2.3.2.2: Ending the edit 31
2.3.2.3: SPECIAL meaning 31
2.4: MENUS 31
2.4.1: WHAT A MENU IS AND HOW TO MAKE ONE 31
2.4.2: AN EXAMPLE MENU DEFINITION 32
2.4.2.1: HOW TO CALL UP A MENU 32
2.4.2.2: Nicknames 32
2.4.2.3: Full Titles 33
2.4.2.4: The 'menu' command 33
2.5: COMMAND LINE SYNTAX 33
2.5.1: COMMANDS 33
2.5.2: LEXICAL STRUCTURE 33
2.5.3: COMMENTS 34
2.5.4: VARIABLE SUBSTITUTION 34
2.5.5: PREDEFINED VARIABLES 35
2.5.6: PROMPTING 35

bt 62
caa 63
cac - chip add chip 67
cacopcmd 69
cad 70
campg 76
capg 77
caport 79
cass 80
cat 83
cbuf 84
cd 87
cecho 88
clearbreak 89
cls 90
configure 91
cop 92
copcmd 97
coplog 98
copstub 99
cs 100
dirty 101
display 102
drtrymode 104
dump 105
dynload 107
echo 108
enable 109
equip 110
err 111
esp 112
exit 114
expect 115
faclist 117

2.5.7: COMMAND ERRORS 35
2.5.8: COMMAND SEARCH ORDER 35
2.6: COMMAND FILES 35
2.6.1: COMMAND FILE NAMES 36
2.6.2: NESTING COMMAND FILES 36
2.6.3: BREAK SIGNAL 36
2.6.4: SPECIAL FEATURES 36
2.6.5: SPECIAL COMMAND FILE: PROFILE.X 36
2.7: INVOCATION 37
2.8: MISCELLANEOUS 37
2.8.1: RUN 37
2.8.2: STOP 37
2.8.3: SETTING THE IAR 38
2.8.4: PROFILES AND FILE SYSTEM 38
2.8.5: LOADING MEMORY 39
2.8.6: UPDATING A SCAN TABLE 40
2.8.7: DOES THE SCREEN DISPLAY CHIP DATA OR IMAGE DATA 40
2.8.7.1: When does a scan string get written 41
2.8.7.2: When do the chips get read 41
2.8.8: REXX INTERFACE AND SYNTAX 41
2.8.8.1: REXX Examples 41

PART 2 -- COMMAND REFERENCE 45

1.0: COMMAND REFERENCE 46
1.1: INTERRUPTING THE ESP PROGRAM 46
1.2: RETRIEVE FUNCTION 46
1.3: PRIMARY ESP COMMANDS 46
! 47
aet 50
alter 52
autoupdate 57
beacon 59
bells 60
bp 61

filefinder 119
flush 120
get 122
gexpect 123
gread 124
gwrite 126
help 128
hreset 129
in 130
in16 132
in32 134
ioflag 135
ipl 136
iplrun 137
is 140
isd 142
jtag 143
layout 144
list 145
listall 146
load 147
load_cntr 149
log 150
ls 151
mal 152
mat 153
meminit 154
memread 155
memwrite 156
menu 157
mmior 158
mmiow 159
move 160
number 161

ocs 163
out 164
out16 166
out32 168
pages 169
pause 170
pg 171
pinread 173
pinwrite 175
pioe 177
pior 178
piow 179
por 180
prs 181
pwd 182
quit 183
readcon 185
reducedpinmode 186
refresh 187
reset 188
resetint 189
run 190
saa 192
sabreak 193
saco 194
sacs 195
sad 196
saespco 199
saespvar 201
sais 202
sal 203
sam 204
sanai 205
sao 206

sapb 207
sapn 209
sat 210
sav 211
save 212
saveLayout 213
scanread 215
screen 216
set 218
setclock 220
setvar 221
sjipl 222
sjreset 223
sleep 224
sreset 225
stat 226
stop 227
time 228
treset 229
tty 230
unset 231
update 232
userbreak 233
ver 234
verbose 235
wait 236
which 239
xlist 240

FORWARD

This document is the authoritative description of the Engineering Support Processor (ESP) program for the 604 family of processor chips. The functions specified in this document will be supported as indicated, and functions not specified in this document will not be supported.

We look forward to your comments on both the tool and the documentation. Since it is our experience that documentation can sometimes be more frustrating than actually using the tool, please give us your comments even if you think they are trivial, such as information order or items not found in the index. This will help us produce a tool that is better for all our customers. Please direct your questions or comments to:

Kent D. Thompson
Tieline 678-6708, External (512) 838-6708
AUSVM6(KENTTHOM)
kentthom@austin.ibm.com

John Bordovsky
Tieline 678-5682, External (512)838-5682
AUSVM6(BORDOVSJ)
johnb@austin.ibm.com

C2M/045-3
Austin, Texas

NOTE: Various groups have indicated a desire to take the "base" ESP program and implement various changes and extensions for their particular needs. We do not recommend doing this. We will not give out the source code unless directed to do so by our managment.

PART 1 -- ESP DESCRIPTION

NOTE: ESP is supported on AIX version 3.2.5.

1.0 OVERVIEW OF THE ESP PROJECT

1.1 INTENT AND PURPOSE OF THE ESP PROJECT

The Engineering Support Processor (ESP) is a debug tool which connects to test boards for the 604 processor. The ESP will be used to debug and verify both hardware and software. The ESP consists of a hardware interface adapter and a program. The interface adapter, which plugs into an RS6000, is used to connect to the JTAG bus and through the JTAG bus to all the latches, registers, and arrays in the processor unit. The software is written in the C programming language and runs under the AIX operating system.

The ESP allows the user to:

1. Read, display, alter, and write all registers, arrays, latches, and memory in the target processor.
2. Set break points, single step, start, and stop the processor.
3. Run diagnostic software.

NOTE:The 604 ESP uses new ESP hardware that supports both COP and JTAG protocol. The 604 ESP software also supports both COP and JTAG chips, but currently only the 604 chip has been connected and used with 604 ESP.

1.2 BRIEF COP OVERVIEW

The 604 processor has built into it a Common On-chip Processor, or COP, and JTAG TAP, or Joint Test Action Group Test Access Port. The JTAG TAP has five physical connections that form a synchronous serial port. It is through this JTAG port that 604 ESP connects to the 604 processor for all activities. (The JTAG circuit is IEEE 1149.1 compliant.)

The JTAG TAP has a synchronous serial port. It can receive commands and data through its serial input pin and can shift out status and data through its serial output pin. Commands and data are used by the JTAG circuits as well as the COP circuits. These commands direct it to do such things as control the chip's clocks (for scanning and testing,) or release control of the clocks (for normal running,) configure the chip's LSSD scan strings for scanning or normal run mode, enable/disable the check stop, and a series of control functions of this nature. The scan strings can be configured as one long scan string and the scan string data clocked out through the serial output port. The long scan string can also be filled with data by clocking data in through the serial input port. By way of the scan string data and certain control lines, all arrays can be written or read from the COP. The COP can initialize the logic and arrays on the chip in order to put the chip into a known, good starting condition.

1.3 HARDWARE REQUIRED

The following summarizes the hardware required to run the ESP.

- RS6000

  Any RS6000 machine including the model 220.

- Hard disk

  ESP can be run at IBM Austin off of AFS, therefore, a large amount of disk space is not a necessity. If a user desires to have ESP locally on his machine then he should allow 100 Meg for all the ESP versions. A user who desires only one version of ESP needs only 10 Meg of free disk space.

  The user should consider how much space AIX, TCP/IP, SNA, Multi-User Services, X-Windows, Motif, other software packages, and testcases will use before determining how much disk space is necessary for the machine.

- Graphic Terminal

  A large graphics display (IBM 6091 or similar) is necessary to display the ESP screens for 604.

- Printer

  Any printer that is compatible with the RS6000. ("PRINT WINDOW" option in the menu requires a 3812 printer.)

- ESP Hardware

  The hardware required to debug the 604 chip is:

  1. An adapter card that goes into the RS6000 (micro channel bus).
  2. A 40 wire (RS6000) ribbon cable about 10 feet long. One end of this goes into the card in the RS6000 and the other end goes into the buffer card.
  3. A buffer card that is intended to sit as close to the test vehicle as possible.
  4. A 16 wire ribbon cable about 10 inches long. One end of this cable plugs into the buffer card and the other end plugs into the test vehicle.

- Token Ring connection

  A Token Ring connection is not required. However, most applications have found

the token ring to be an extreme advantage.

- Connection to a VM host

  Test cases can be down loaded from VM if required.

- Manufacturing

  Manufacturing requirements are determined on a case by case basis.

1.4 SOFTWARE OVERVIEW

The ESP hardware and "core" software provide access to the JTAG bus, and through the JTAG bus to all latches, registers, and arrays in the CPU, as well as external memory. Which facilities are displayed on the screen, how the screen is laid out, what labels are on the screen, what menus are available, etc, can all be set by the user, and indeed, must be set. Some initial screens will be set up by the ESP designers, but there are NO default screens in the ESP program.

When the ESP program is run, it looks for and executes any commands in a file called "profile.x". This file contains ESP commands that directly personalize the program, and points to other files which contain more commands to be read and executed. All screens to be used and all chip scan strings to be accessed must be defined in these files. A more detailed discussion of these files is in the following sections.

NOTE:These files contain a series of ESP commands. There is no branching (conditional or unconditional), no do-while, and so forth. These commands are in exactly the same format as commands typed onto the command line of the ESP program display and are explained in the "Command Reference" section of this document.

1.4.1 FUNCTION LIST

1.4.1.1 Basic Functions

1. Set processor to known state and start.
2. Load memory with data from support processor's hard file.
3. Set breakpoints (hard, soft, and interrupt).
4. Dump and display complete processor state (all scan string and array information)
5. Display and alter registers
6. Display and alter arrays
7. Display and alter memory (real).
8. Instruction single step.
9. Display and alter GIO addresses

10. Display and alter memory mapped IO.
11. Display, alter, and expect PIO.

1.4.1.2 Additional Supported Functions

1. Electronic means of updating scan string information from BDLCS.
2. All devices in the scan tables are accessible.
3. Configure for different EC levels without re-compiling
4. Support testing single chips (alone).
5. Chip to chip wiring test).
6. Self Test .
7. User programmable push buttons perform any ESP command
8. Display registers, arrays, and memory in ASCII.
9. Instruction decode (display opcode for next instruction)
10. Print screen (data and control registers to printer).
11. Scroll through memory or arrays
12. Run for N instructions.
13. Write/Read Scan String Without running.
14. Log all commands to a file and run again.
15. Command file "till_status" (wait) function
16. User programmable screens and menus.
17. Alter Memory in ASCII.
18. Display memory from a programmable origin and/or offset.
19. Issue COP commands directly.
20. Log all COP commands.
21. Command syntax help available (many commands).
22. Error messages can be logged.
23. Incredibly easy to add a lot of functions.
24. Adapter card's address is selectable.
25. Specify which chips are present.
26. Memory access by 4 byte word.
27. Automatically access arrays.
28. Automatically set parity .
29. Multiple processor support

30. Rexx front end for looping ESP command testcases.
31. Alter devices in ASCII.
32. Retrieve previous commands.
33. Display registers, arrays, and memory in EBCDIC.
34. Display registers, arrays, and memory in scientific notation.
35. Loads programs into memory that are in XCOFF or TOC formats.
36. Run from remote location.
37. Save processor state to hard file and load at a latter time.
38. Motif Interface allows the display of more than one ESP screen at a time.

1.4.1.3 Machine Under Test (MUT) Hardware Restrictions

Functions the hardware does not allow.

- Run for N instructions in real time.
- Run in real time and break when a register contains a certain value (except IAR).
- Count the number of cache misses.
- Set multiple breakpoints.
- Interface directly with any I/O device.
- Stop the I/O chip and restart.

1.4.2 STOPPING THE PROCESSOR

1.4.2.1 Hard and Soft Stops

The 604 processor supports two types of stops. The hard stop will freeze the clocks on all chips simultaneously. This will allow the exact state of the machine to be examined. However, because this can happen during some time critical sequences, normally the MUT cannot recover from this type of stop.

The second type of stop is a soft stop. For this type of stop, the 604 will stop dispatching instructions and will wait for all CPU activity to stop. Then it will stop the CPU chips. The ESP will then insure that all DMA and memory activity has ceased prior to completing the stop sequence. This type of stop does not allow for the exact processor state to be examined during the middle of a step, but it does allow the system to be restarted.

There are a number of combinations of these stops. The user should be aware that some stops MUST be followed by a system reset.

The processor will stop PRIOR to executing the breakpoint instruction.

1.5 SOFTWARE INSTALLATION

Since all the ESP files are stored on AFS, the ESP code can be run without installing it locally. Once a machine has access to AFS and the "awd" cell then some minor changes to the .profile will allow ESP to run.

1. klog into afs on the AWD cell (usr/afs/bin/klog username -cell awd)
2. Edit your $HOME/ .profile by adding to the PATH variable "/afs/awd/public/esp/rios/bin".
3. Execute the .profile with ". .profile". The .profile will be run automatically after a logoff and a logon.
4. Change directory (cd) to your working directory.
5. Type "esp".

1.5.1 COMMAND FILES

All ESP definition is accomplished by the program reading files which contain a series of ESP commands. The sequence of these ESP commands is critical in some cases, and in other cases is not. Menu definitions may be done at any time. However, the chip definitions and the screen definitions must be done in sequence.

1.5.1.1 Definition Sequence

Menu definitions can be done any time. Before a chip or chip facility can be referenced on a screen, the chip and facility must be defined. The chip must be defined in the following order:

1. cat
2. cass
3. cad
4. cac

NOTE: The "User Interface" and "Command Reference" sections below discuss these commands in detail. A screen must be defined before labels and data fields can be defined on the screen. Therefore, the sat command for a screen must be completed before either sal or sad commands for that screen can be used.

1.5.1.2 Editor producing 'flat file'

The command files are all plain text, meaning that there are no imbedded non-printable or non-ASCII characters in them. A text editor (such as VI, e3 or INed) is required only if creation or modification of the command files is to be performed.

1.5.1.3 profile.x

When the ESP program starts, it looks for a command file called profile.x. If this file exists, the ESP program executes all the commands in it. The profile command file is typically used to initialize the menu system for the application at hand. It is, however, not required.

1.5.1.4 menu.x

Profile typically runs commands that will configure the ESP program. Among them might be a command file to configure the menu system, like menu.x. This file name is only used to illustrate the concept of configuring menus. The ESP program does NOT look for 'menu.x', and does not care if menus are configured or not.

The point is: there are NO default menus in ESP. ALL menus must be built with commands that are TYPICALLY performed from command files. The command file is TYPICALLY executed by the profile.x command file. Menu contents are totally programmable.

1.5.1.5 screen.x (Screen Definition)

All screens used to manipulate data in the MUT come from Screen Definition files, which are just another command file. There are NO default ESP program screens. Screen names are totally user definable, however, the extension '.x' IS required for a screen command file.

1.5.1.6 chipst.x (Scan Table)

Screens in the ESP program are defined with Screen Definition command files that MUST refer to the data they will display by chip name and field, or device name. These two names, chip and field, MUST exist in a data base that is known as a Scan Table. The scan table file, one for each chip to be referred to, is presented to the ESP program in yet another command file that uses ESP commands like chip add type (cat) and chip add device (cad) (e.g.604st.x).

In addition to the scan table files, there is an additional file for most chips which contains a list of devices that have been concatenated together for convenience. This allows latches that have different names, but are functionally related to be addressed as one device on a screen. (e.g. 604CAD.x)

There are also a series of files which contain the information required to access arrays. These files contain information such as what devices must contain the control bits and data to write an array. (e.g. ITLB.DEF)

1.6 RUNNING THE ESP PROGRAM

The ESP program is run by entering 'esp' <enter> on the keyboard (without the single quotes). The command follows the typical AIX convention of using lower case letters. This executes a shell script called "esp".

1.6.1 HOW SCREENS SHOW UP IN THE PROGRAM

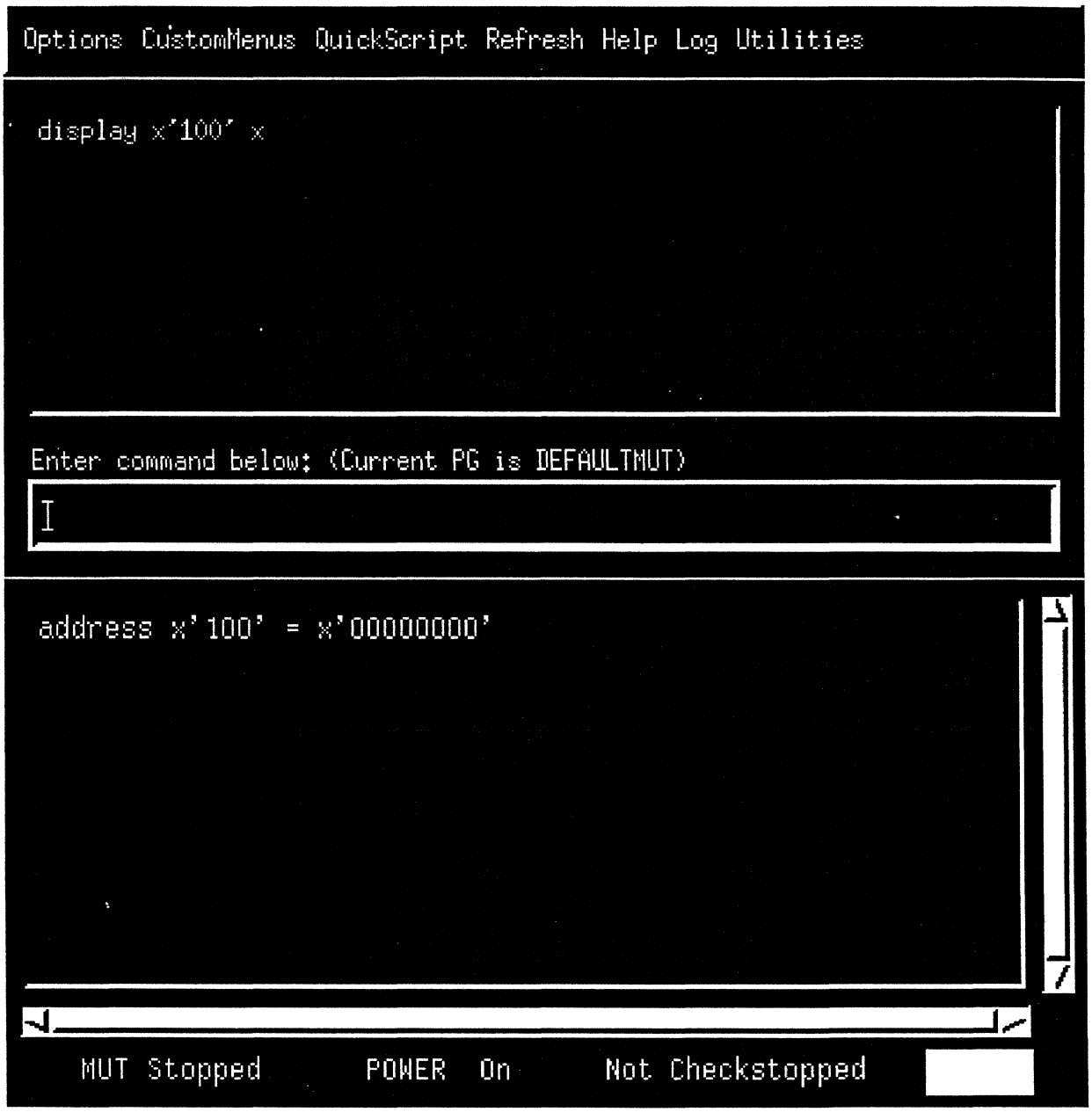
There are NO default screens in the ESP program. If you want a screen to show up, you must build a Screen Definition command file, which specifies the look of the terminal screen, and refers to the Scan Table(s) for the chip(s) data being displayed.

Menus, command files, and Command Lines can be used to make the screen appear on the terminal screen.

2.0 USER INTERFACE

2.1 The ESP CONSOLE

Figure 1:ESP Screen Example



ESP has a Motif Window Interface. The first thing the user sees when ESP is executed is the console window. (See Figure 1.) This X-Window is comprised of five sections: The Menu Bar at the top, the Command History List just below, followed by the Command Line, the TTY Section, and finally the MUT Status.

The console window can be resized using the standard X method of grabbing the window by an edge or corner and dragging it to a new size. This is performed with the left-most mouse button. Normally, clicking the left-most mouse button once will position the cursor. Two clicks will select a word, where words are delineated by white space. Three clicks selects a line. The middle mouse button will paste what has been selected at the current cursor position.

The Menu Bar on top of the ESP console allows the user to select frequently used ESP commands and Custom Menus.

2.1.1 Options

The 'Options' button will pull down a menu of selections that allow you to choose:

Exit ESP

Save Layout

Set

Coplog

Pushing 'Exit ESP' will cause the ESP program to immediately terminate.

Pushing 'Save Layout' will cause the present location of any screens you have up, including the ESP Console, to be remembered. The next time you run ESP, these same screens will come up at the same locations. The remembered layout information is stored in file $HOME/.esplayout.x .

Pushing 'Set' will cascade another menu with the following selections:

Echo on

Echo off

Verbose on

Verbose off

TTY on

TTY off

File Finder on

File Finder off

Pushing 'Echo on' will turn on echo so that, after any user input is macro expanded, it will be echoed to the TTY window. (Echo is turned on using the ESP command 'set echo'.)

Pushing 'Echo off' cancels the echo command. (Echo is turned off using the ESP command 'unset echo'.)

Pushing 'Verbose on' will establish a mode where any input to ESP is immediately sent to the TTY window. This happens before macro expansion of ESP variables occurs.

Pushing 'Verbose off' cancels the verbose mode.

Pushing 'TTY on' means that output from a ESP command that is shown in the TTY window will be seen in the window.

Pushing 'TTY off' means that even though output is sent to the TTY window, it will not be presented there. (This does not affect logging.)

Pushing 'File Finder on' means that each time ESP executes a command file, its full path and file name are written to the TTY window.

Pushing 'File Finder off' means that if ESP executes a command file, it will do it without mentioning the fact.

Pushing 'Coplog' will cascade another menu with the following selections:

Coplog off

Coplog 1

Coplog 2

Coplog 3

Coplog 4

Coplog 5

The coplog command allows visibility of the COP commands being sent to the MUT.

2.1.2 Custom Menus

Contains the ESP debug screens. The selections in the menu can be changed. See Menus.

2.1.3 Quickscript

This will open a window that allows the user to write, modify and run ESP command "programs" without exiting the ESP program. Editing is performed with a few simple commands:

add a line - <enter>

delete a line - select a line with mouse and <delete>

add a character - type a character

delete a character - <delete>

2.1.4 Refresh

This button updates all ESP debug windows (screens).

2.1.5 Help

This button lists all ESP commands in the TTY window.

2.1.6 Log

The 'Log' button will pop up a window with selections dependent on the current logging state. If logging is currently turned off then the window will look something like this:

```
Logging is currently turned OFF
    New Log -- Append          SAME      New Log -- Truncate
```

If logging is currently turned on then the window will look something like this:

```
Logging is ON to file:/usr/fred/junk.log
    Clear Log     Close Log      Print Log      Edit Log
    New Log -- Append          SAME      New Log -- Truncate
```

In either case 'New Log -- Append' will allow you to pick a log file that already exists so that logging will append to the file.

'New Log -- Truncate' will allow you to pick a log file that will be truncated if it already exists.

The 'SAME' button allows you to leave the pop up window without changing anything.

The 'Clear Log' button will close the log file and stop logging.

The 'Print Log' button will make a temporary copy of whatever is currently in the log file, print the temporary copy to the default system printer, and then delete the temporary copy. Logging continues during and after this button is pressed.

The 'Edit Log' button will make a temporary copy of whatever is currently in the log file, and then cause a window to come up running the editor described in the AIX environment variable EDITOR. When you leave your editor, the temporary file is deleted. Logging continues during and after this button is pressed.

2.1.7 Utilities

The 'Utilities' button will pull down a menu of selections that allow you to choose:

Clear TTY window

Print TTY window

Editor

Print a Window

Open a Shell Window

Pushing the 'Clear TTY window' button will cause the TTY window to be cleared of all stored text.

Pushing the 'Print TTY window' button will cause the entire TTY window (not just the vis-

ible part) to be sent to the default system printer as ASCII text.

Pushing the 'Editor' button causes you editor to be run in a window. That is, the editor to be run is the editor described in the AIX environment variable EDITOR.

Pushing the 'Print a Window' button will change the cursor to a white cross. Move the white cross cursor to any window up on your display and press the left mouse button once. You should immediately hear a beep, and after a pause, a double beep. The cursor should then change back to its original shape. The window you clicked on will be turned into a 3812 format bitmap, and then sent to the system default printer via qprt. (This requires a 3812 printer.)

Pushing the 'Open a Shell Window' causes an aixterm window to be opened. The window inherits the ESP environment and will exist in the same directory that ESP was run from.

2.1.8 Command History List

The Command History List allows you to re-execute previously typed ESP commands. You can 'single click' an old command and then press <enter> to execute it, or 'double click' on the old command to execute it right away. If you prefer, you can use the up and down arrow buttons to retrieve old ESP commands.

2.1.9 Command Line

The Command Line is the place to enter your ESP commands. The commands are explained in the Command Reference section of the manual.

2.1.10 TTY Window

The TTY window is the ESP output. Whenever you execute a ESP command that has an output expected, the output will be placed into the TTY window.

2.1.11 Machine Under Test Status

The Machine Under Test (MUT) Status portion of the ESP console informs the user of hardware status. In the case of ESP, hardware status refers to the status of two physical lines connected to the MUT. The POWER on/off indicator monitors the +POWERGOOD signal, and the Checkstopped indicator monitors the -CHECKSTOP hardware line.

2.2 SCREENS

Although the ESP has no built in screens, the ESP product has packaged with it many useful screen definitions. You are encouraged to build your own screens to help look at your chip set in a way most useful to you!

Every screen is created from a collection of normal ESP commands such as 'sat', 'sal', 'sad', 'sam', and others. Each of these commands may be found in the Command Reference section of this manual.

The grouping of these ESP screen commands into a single file is not required, but is usually the most useful way of creating a screen. In fact, every screen supplied with the ESP package is one-screen-to-a-file, where the screen name and the file name match (but where the file name has a '.x' tacked on to it).

This is a summary of all the ESP screen commands. Look these up in the Command Reference section of this manual for a complete explanation of their abilities.

Table 1: Summary of ESP Screen Commands

saa	Screen Add Array display
sabreak	Screen Add Breakpoint field
sacs	Screen Add Cycle Step field
sad	Screen Add Data field
sais	Screen Add Instruction Step field
sal	Screen Add Label field
sam	Screen Add Memory display
sanai	Screen Add Next Assembler Instruction field
sao	Screen Add Origin field
sapb	Screen Add Push Button
sat	Screen Add Title
sav	Screen Add Variable field

2.2.1 WHAT A SCREEN IS AND HOW TO MAKE ONE

A screen always starts with the 'sat' command: sat regs "This is a register screen"

The 'sat' command does two things: creates a named screen, in this example "regs", and establishes a title for the screen that will be shown at the top of the screen when it becomes visible. The (literal) title of the screen can later be used to bring the screen up, or the nickname ("regs") can be used to bring up the screen.

Any furthur screen commands will use the same nickname, "regs", to relate their action onto the same screen. For example, to add a label to the screen: sal regs 0 1 "These registers

are found in the CACHE"

Label fields cannot be moved to using the cursor keys or mouse. Data fields not only display data but can be moved to and edited. An example data field: sad regs 1 0 x ?MSR

In the 'sad' example we said "create a data field on the 'regs' screen at row 1 column 0. Present the data from the architected register MSR there, and do it in hexadecimal.

2.2.1.1 An example Screen Definition

Here is an example of how a screen is created using sat, sal, and sad:

```
-------------------------------------------------------
sat      fpu     "604 Floating Point Execution Unit"
#        screen  row     col     label
sal      fpu     0        2      "Decode Op "
sal      fpu     0       30      "XER "
sal      fpu     1        2      "Exec Op    "
#        screen  row     col     type    chip    field
sad      fpu     0       12      X       604     reg0
sad      fpu     0       34      X       604     reg1
sad      fpu     1       12      X       604     reg2
screen   fpu
-------------------------------------------------------
```

The entries above would create a screen like this

Figure 2: Screen Definition Example

2.2.1.2 Start rows and columns

Start row and column values in the sal and sad commands will begin at 0. The top left corner of a screen is at coordinates 0,0.

2.2.1.3 Data TYPES defined

Data may be displayed as 'A' for ASCII string, 'E' for EBCDIC, 'X' for Hexadecimal, 'D' for Decimal, 'B' for Binary, 'O' for Octal and 'F' for floating-point. Note that decimal does not work for devices with more than 32 bits.

2.2.1.4 Size of screen DATA cells

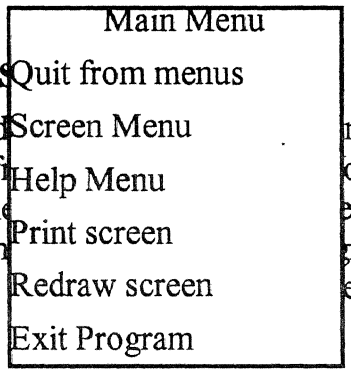
Another fine point about the screen that was created is the size of the data cells created (designated by 00000000 in this example). The size of the data cell is determined by the 'type' field in the sad command, and the number of bits as specified in the Scan Table entry for the 'chip' and 'field' as specified in the sad, and existing in the Scan Table.

Since a Hexadecimal digit represents four bits, then if the device is 32 bits, it can be represented in 32/4 = 8 hex characters. If a field is 'type' 'B', then the same device, holding 32 bits, must be represented on the screen in Binary, and therefore would require 32 characters of space. Similarly, Octal for 32 bits would require 32/3 = 11 (rounded up) characters to represent the device contents. (The un-used bits would be set to zero automatically).

2.2.1.5 References to CHIPs and FIELDs

The sad command refers to a 'chip' and 'device' as the source of the data that will be displayed/modified in the screen cell. These references must already exist in the Scan Table prior to creating the screen. If they do not exist, the sad command will reject the request for creating the data cell, and print an error message. The screen being created will simply NOT have the referenced device on the screen.

2.2.2 HOW TO CALL UP A S



In general, screens can be 'called nickname', by 'title of screen', using the 'screen' command, or fi ou can simply type the name of the screen in on the command line ely on the ability to communicate WHICH SCREEN you are talkin ram you can refer to a screen by either its 'nickname' or its 'title'. e given to the screen when it is created with the sat command.

2.2.2.1 Nicknames

Nicknames allow screens to be referred to by a short, perhaps meaningful, word instead of the screens 'title'. In the example given above, notice that the sat command specified a nickname 'fpu' for the screen. Also, as shown, the sal and sad commands took advantage of the shorter nicknames by using them to specify which screen they wanted to add a label or data cell to.

2.2.2.2 Full Titles

Screens may be displayed by asking for the title. In our example the title is ' 604 Floating Point Execution Unit'.

2.2.2.3 The 'screen' command

When a screen is asked for, by either its 'nickname' or its 'title', it is asked for in one of two

ways: using the 'screen' command, or by inference. You can use the 'screen' command as in: 'screen fpu'. This would make the 'fpu' screen appear.

2.2.2.4 By name on the command line

If you are on the command line you can simply type the name of the screen. The program will first look for the 'name' as a command, and failing to find the command, it will then look for the 'name' as a REXX program, and if not found, it will then look in the ESPPATH directories for "'name'.x". If the command file exists, then it is executed to display the screen.

2.2.2.5 Menu invocations

Menus are user programmable, and any menu can include a command 'screen fpu' (as an example), or simply have the name of the screen as the command; 'fpu'.

2.3 KEYBOARD / EDITOR

The ESP keyboard is a standard RS6000 keyboard.

Here, we focus on the keyboard as it is used during modification of data on a screen cell, which of course is done to implement a change to some MUT hardware register or device.

While on a SCREEN, the cursor will ALWAYS be in some current cell. While in that cell, you are either editing or you are not. Cursor motion over the cells is possible without ever changing anything.

2.3.1 CURSOR MOTION

Cursor motion is the act of moving the cursor from cell to cell with any of the following keys:

TAB	Move to the next editable cell.
BACKTAB	Move to the previous editable cell.
HOME	Move to the most upper-left cell editable.
END	Move to the most bottom-right cell editable.
UP ARROW	Move up a cell.
DOWN ARROW	Move down a cell.
LEFT ARROW	Move left a character.
RIGHT ARROW	Move right a character.
ENTER	Move to the next editable cell.

2.3.2 EDITING A CELL

The edit of a cell begins when you overtype the value of some cell. The following list of keys are useful in the EDIT mode:

2.3.2.1 Originating the edit

Depressing any of the following keys starts the edit:

OVER WRITE	Over-typing a character at the cursor.

2.3.2.2 Ending the edit

The edit is over if you press a cursor motion key (other than left or right arrow keys) or you change focus from the edit cell to some other location. This includes the Enter key and the Tab key, but they have other special significance also. Once the edit is over, an 'alter' command is automatically generated and performed by the ESP program for the device just changed. By the time you get to the next cell, the previous cell will have been modified in the ESP programs version of the Scan String.

2.3.2.3 SPECIAL meaning

ENTER	Can terminate an edit, and causes cursor motion like tab.
PAGE DOWN	When a screen is displayed that has cache or memory displayed, and the cursor is in the data portion of the screen, the page down key will cause the displayed data to page down by the amount set by the 'sam' command for that screen.
PAGE UP	When a screen is displayed that has cache or memory displayed, and the cursor is in the data portion of the screen, the page up key will cause the displayed data to page up by the amount set by the 'sam' command for that screen.

2.4 MENUS

2.4.1 WHAT A MENU IS AND HOW TO MAKE ONE

Menus are created by the execution of multiple menu add line (mal) commands. Menus are created using the menu add title (mat) command, specifying the (optional) nickname and (not optional) title of the menu. Once the menu exists, then mal will add lines to the menu.

2.4.2 AN EXAMPLE MENU DEFINITION

Here is an example of how a menu is created using mat, and mal:

Figure 3: Menu Definition Example

```
#cmd     menu nickname     menu title
mat      main              "Main Menu"

#cmd     menu nickname     line text              command text
mal      main              "Quit from menus"      ""
mal      main              "Screen Menu"          "menu screen"
mal      main              "Help Menu"            "menu help"
mal      main              "Print screen"         "prs"
mal      main              "Redraw screen"        "redraw"
mal      main              "Exit Program"         "exit"

The entries above would create a menu like this:
```

2.4.2.1 HOW TO CALL UP A MENU

In general, menus can be 'called up' in several ways; by 'nickname', by 'title of menu', by using the 'menu' command or by clicking on the "Custom Menus" button. All of these methods rely on the ability to communicate WHICH MENU you are talking about. In the ESP program you can refer to a menu by either its 'nickname' or its 'title'. Both these monickers are given to the menu when it is created with the mat command.

2.4.2.2 Nicknames

Nicknames allow menus to be referred to by a short, perhaps meaningful, word instead of the menus 'title'. In the example given above, notice that the mat command specified a nickname 'main' for the menu. Also, as shown, the mal command took advantage of the shorter

nickname by using it to specify to which menu it wanted to add a line.

2.4.2.3 Full Titles

Menus may be displayed by asking for the title. In our example the title is 'Main Menu'. The spelling, including spaces, would have to be an exact match.

2.4.2.4 The 'menu' command

When a menu is asked for, by either its 'nickname' or its 'title', it is asked for by using the 'menu' command. The 'menu' command allows you to call up a menu command (.x) file.

2.5 COMMAND LINE SYNTAX

The command processor is a command programming language. It reads and executes commands from either the terminal device or a file.

2.5.1 COMMANDS

A command is a sequence of words, separated by blanks. The first word is the name of the command. The remaining words are arguments for the command.

2.5.2 LEXICAL STRUCTURE

The command processor uses the following delimiters to split lines into words:

```
space tab new-line < > | ;
```

Space and tab are used only as delimiters. New-line marks the end of a command.

The following form separate words which are reserved for future use. These special cases could be used to separate several commands on one line or to redirect input and/or output for a command.

```
< > << >> | ;
```

Quoting any of these delimiters removes their special meaning. Any character may be quoted by preceding it with a \. A new-line preceded by \ is ignored, and the command may be continued on the next line. A \ is represented by \\.

All characters, except new-line, enclosed between a pair of grave accents are quoted. The following three rules apply to new-lines within grave accents.

1. A new-line preceded by a \ is a true new-line character.

2. A new-line not preceded by a \ is an error (unmatched quote).
3. When it precedes any character other than new-line, \ represents itself.

All characters, except new-line, enclosed between a pair of double quotes are quoted. The following three rules apply to new-lines within double quotes.

1. A new-line preceded by a \ is a true new-line character.
2. A new-line not preceded by a \ is an error (unmatched quote).
3. When it precedes any character other than new-line, \ represents itself.

Note that quoting does not delimit a word. Quoted strings may be only part of a word.

2.5.3 COMMENTS

The # symbol (number sign or pound sign) can be used to mark a line as a comment. The comment lines are discarded and take up no space in memory. A comment begins with a non-quoted # and causes all characters up to, but not including, a new-line to be ignored.

2.5.4 VARIABLE SUBSTITUTION

User defined command variables may be created and removed with the set and unset commands. These are general purpose macros.

As the command processor parses a command line, it scans for variables and substitutes their values in place of the variable names. These variables are introduced by the question mark character. They may be represented in one of the following ways.

?NAME	Name is the name of a user defined variable. The characters allowed in name are 'a' through 'z', both upper and lower case, underscore and decimal digits. Name must not begin with a digit . To expand variables whose names include characters other than the ones allowed here, use the following notation.
?{NAME}	Braces may be used to separate the name of a variable from other characters. For example, ?{chipa} expands the variable, chipa. And, ?{chip}a expands the variable, chip. Braces also allow name to contain any characters other than braces.

Except within quotes, the substitution of variables may be disabled by preceding the ? with a \. Within double quotes substitution always occurs. Within grave accents substitution never occurs.

If a variable is expanded within double quotes or grave accents, then embedded blanks will not delimit words. If a variable is not within quotes then embeddèd blanks will divide the substituted variable into separate words.

2.5.5 PREDEFINED VARIABLES

See the "set" command in the command reference section of this document.

2.5.6 PROMPTING

If commands are being read from the terminal device then the command processor prompts for each line that is read. The value of the variable, prompt1, is used to prompt for a new command line. If a line is continued then the value of the variable, prompt2, is used to prompt for the remainder of a line.

If the variable, noexecute, is set then its value is used instead of the value of prompt1.

2.5.7 COMMAND ERRORS

All commands return an exit status to the command processor. The status of each command tells the command processor whether the command completed normally or exited because of an error. Normally the returned status is ignored.

When running a command file, it is often desirable to stop the file if one of the commands exits because of an error. If the variable, exitonerr, is set then the command processor will exit after any command which returns a non-zero error status. Control will be returned to the user's command line.

2.5.8 COMMAND SEARCH ORDER

After the command processor has parsed a command line, it must decide how to execute the command. First, the command processor checks the command name to see if it is one of the internal commands. If it is not found the command processor checks to see if the command name is the title or nick name of a screen that has already been referenced in the current ESP session. If it is, then that screen is displayed on the terminal screen. If it is not a command or screen the command processor looks for a file with the same name as the command name (with a ".x" extension). If the file is found the command processor will read commands from the file. If an executable file without a ".x" extension is found, ESP passes the command to REXX. REXX will properly execute the command if it is a REXX program.. When a command is not found, the command processor prints a message (command not found).

2.6 COMMAND FILES

Command files are sometimes called 'batch' files. Command files are command lines that could be typed in while running the ESP program, but that are instead imbedded in a disk file to be read in and executed, one at a time, automatically, by the program.

NOTE: REXX will run AIX files also.

2.6.1 COMMAND FILE NAMES

Command files must have a file name extension of ".x" or ".X". To run a command file, its file name may be entered either with or without the ".x" extension. If you try to ask for a screen by name, and the screen is not in memory, the ESP program will create a filename, "name.x" and search for the screen command file.

2.6.2 NESTING COMMAND FILES

Command files can call other command files. The number of nesting levels is limited only by the amount of memory available.

2.6.3 BREAK SIGNAL

Pressing the break key sends a program interrupt to the command processor. This signal can be used to stop the execution of a command file. If commands are being read from a command file, break causes the command processor to return to the interactive mode.

NOTE: If a command file has called another command file (nested command files), the break key will exit one level of command nesting. Pressing the break key from the 0 level of nesting (the main ESP program) will cause the program to halt and return the user to the AIX shell.

Because this sequence is programmable, it can vary from machine to machine. Most machines have a default profile (.profile) that is programmed to interpret control-C as the break signal or as the program interrupt.

2.6.4 SPECIAL FEATURES

Command files have the same capabilities as the user has when typing commands in on the command line of the ESP program. However, a thing or two might be mentioned here which might make it easier to use while creating and using command files.

Use the last line of a command file to call up a screen. This is THE WAY to do it for a screen definition command file, and could be used to advantage in other applications.

Use a command file, as a Screen Definition, but where all the commands in the file are sat and sal's. This would create a SCREEN that is nothing but label cells, and nothing on it to edit (no data). This might be useful as a NOTE of explanation, or as a HELP screen.

2.6.5 SPECIAL COMMAND FILE: PROFILE.X

Profile is the only 'special' file the ESP program uses. It is looked for when the ESP program initializes, and if it exists then it is read in and executed. Profile.x does not have to exist. However, menus will be useless until they are programmed, and profile typically takes care of this task. Profile could also put up the first screen.

Here is an example of a typical profile.x to start up an ESP program:

```
get ?ESP/scantables/bin/604dd1.bin    #read scantables
menus1                                #read menu definitions
arch                                  #set architected names
equip 604                             #establish 604 as physically present
meminit                               #initialize ESP memory handler
configure "?ESP/bin"                  #Cause the hardware to be initialized
```

2.7 INVOCATION

_SYNTAX esp
open esp

2.8 MISCELLANEOUS

Many functions performed by the ESP can be done by an operator entering a few key strokes. Often, these few key strokes cause the program to issue hundreds or thousands of commands to the MUT. Memory and array access typically performs the most operations.

2.8.1 RUN

The following is some of the things that the ESP must do to start the system running.The system under test must NOT have hardware lines -checkstop or +powergood in an error state.

- Any cached memory is flushed to the system.
- Any cached scan strings are flushed to the system.
- (All ESP flags about the data from the system are marked invalid.)
- The following COP commands are issued to start the processor:

  to all chips: RESUME

2.8.2 STOP

The following is some of the things the ESP must do to stop the system.

NOTE:The system under test must NOT have hardware lines -checkstop or +powergood in an error state.

The command 'stop', shown with no arguments, defaults to 'soft' stop.

A soft stop is done as follows:

- COP command HALT sent
- The ESP waits for run/stop status to indicate a stopped condition
- COP command FREEZE is sent.
- The ESP internal flags are set to indicate data is available from the processor

If 'STOP -H' was issued, indicating a request for a HARD STOP, then:

- A COP FREEZE command is BROADCAST to all chips in system
- The ESP waits for run/stop status to indicate a stopped condition
- The ESP internal flags are set to indicate data is available from the processor

2.8.3 SETTING THE IAR

The following is the sequence the ESP performs to set the instruction address register at address x'100'.

```
reset                                   /*resets the scan latches to zeros*/
bp x'00000100'                          /*stop before instruction at address */
                                        /*x'100' is performed*/
alter ?MSR x'00000000'                  /*sets up MSR to begin at address*/
                                        /* x'100'*/
alter x'100' x'[branch instruction to code start]'
alter 604 ZAA_604.SAA.SDY.SCE.HRESETREG.THELATCH.L2 b'1'   /*needs*/
                                        /*to be here*/
alter 604 ZAA_604.SAA.SDY.SCE.REXCEPTREG.THELATCH.L2 b'1' /*needs*/
                                        /*to be here*/
iplrun 1                                /*initial run */
```

2.8.4 PROFILES AND FILE SYSTEM

ESP will search for files in these ways:

1. While trying to find a .x file
2. While trying to find a rexx file
3. While trying to find a file to be loaded

As a brief overview of how ESP tries to figure out what the user is asking of it, lets start by having a user type something in on the command line. ESP will first try to find the typed line as an ESP command. If the command is not found, then ESP will try to match the line as if it were a screen name, and ESP will search its existing screen list for a match. If the line typed is not a screen name, then ESP will assume it is a '.x' file on disk, and add the '.x' extension to the command, and look for the file first in the current directory, then via the path as specified in the AIX environment variable ESP1PATH. The dot-x file will be searched for in all directories specified, first with a lower case '.x' and then with an upper case '.X'. If the file is never found, then ESP passes the command line to REXX and lets REXX look for the command. REXX will use the AIX environment variable PATH to search for the REXX file. If REXX cannot find the file, it will return to ESP and ESP will print 'command not found: xxxx' and terminate the search.

Summary of search order:

1. is it a command
2. is it a screen in ESP memory
3. is it a .x file in current directory
4. is it a .x file in ESPPATH
5. is it a .X file in current directory
6. is it a .X file in ESPPATH
7. is it a REXX file in current directory
8. is it a REXX file in PATH

So we have just identified two paths searched from ESP; the ESPPATH used by ESP to look for dot-x files, and the PATH which is used by REXX to look for REXX files. A third path is used when ESP is loading or preloading MUT binary files.

When the load command is used, ESP will try to find the MUT binary file using the AIX environment variable ESP1AVPBIN. Inside ESP, ESP1AVPBIN is read and sets ESP variable ESPAVPBIN. The ESPAVPBIN path is used to point to where the MUT binaries are located.

Summary

1. is source file in current directory
2. is source file in ESPAVPBIN
3. is .pre file in current directory
4. is .pre file in ESPAVPBIN

2.8.5 LOADING MEMORY

The ESP loads memory by loading 8 bytes of data into the 604 scan string and commanding the MUT to run. It actually takes longer to load the data into the 604 scan string than it

takes to flush the data to the MUT and for it to run.

NOTE: All data is loaded into memory, not into the dcache or the icache. When loading a program into memory, there is always a chance that the icache and the dcache could have a current shadow of some of the memory. Normally the caches should be cleared prior to loading and executing a program.

2.8.6 UPDATING A SCAN TABLE

The ESP has a file $ESP/scantables/bin/604ST.x. ($ESP corresponds to a value in the environment which gets set by the esp shell script.) These files contain examples of how to construct a binary scan table for whatever combination of chip EC level scan strings are desired. To construct a scan string for a MUT, do the following:

1. Change your current directory to $ESP/scantables/bin.
2. Copy the 604.x file to a file with a different name.
3. Edit this file to show the chip scan string files that you want to include and the output file name you want used.
4. Start the ESP program and follow the prompts.
5. Update your profile.x file to get the new binary scan string file name.

2.8.7 DOES THE SCREEN DISPLAY CHIP DATA OR IMAGE DATA

The ESP program has two flags which are used to control the reading and writing of the scan strings.

DIRTY_FROM FLAG — The "dirty_from" flag indicates that the MUT has run since the scan string for a particular chip was read.

SET — This bit is set when the MUT is told to run or when the dirty command is issued.

CLEARED — This bit is cleared when the scan string for that chip is read.

CHECKED — This bit is checked when there is a request for the data from that chip. If an operator wants to have some data from that chip displayed, the ESP program will check this flag to see if the scan string must be read. If the ESP has a current image of the scan string data in memory, it will not read the scan string again.

DIRTY_TO FLAG — The "dirty_to" flag indicates that the image of the scan string in the ESP's memory has been changed and the data in the chip is no longer current.

SET This bit is set when the scan string data is altered.

CLEARED This bit is cleared when the scan string data is read or written.

CHECKED This bit is checked prior to starting the MUT processor. If the "dirty_to" flag is set, the ESP program will write the data to the chip prior to starting the MUT running. This bit is also checked when the flush command is issued.

See the 'flush' command, the 'dirty' command, the 'display' command, and the 'expect' command.

2.8.7.1 When does a scan string get written

A scan string is written to a chip when the MUT is told to run a cycle step, run an instruction step, run or iplrun is issued, or when the flush command is issued.

2.8.7.2 When do the chips get read

The chips are read when data is requested which is contained in a scan string that has the dirty_from bit active.

2.8.8 REXX INTERFACE AND SYNTAX

Although there are some limitations, REXX is a very powerful addition to ESP. The current version of REXX will satisfy most of our needs.

The guide for REXX on the RS6000 is the manual "Common Programming Interface Procedures Language Level 2 Reference". It can be obtained through IBM "genreq" with the number SC24-5549-0.

NOTE:REXX will not allow file names to be longer than 10 characters.

NOTE:Be very careful about upper and lower case. REXX likes upper case and AIX likes lower case. REXX programs called as functions must have lower case names.

Our version of REXX uses the 'address esp' command to cause REXX to redirect the commands to ESP. Address command causes REXX to direct commands to AIX or REXX.

NOTE:You can call a REXX file from a .x file, but not a .x from a REXX file.

2.8.8.1 REXX Examples

The following example is a program that was written by Robert Golla to test the arrays on the fixed point chip.

Figure 4: REXX Example - Testing arrays on fixed point chip.

```
-------------------------------------------------------------------
/* single testcase to check all arrays on fxpt chip */
parse arg chip

if chip="" then do
   say "you must enter an output file name ... goodbye"
   exit 99
   end

data32.0='00000000'; data32.1='55555555';
data32.2='AAAAAAAA'; data32.3='FFFFFFFF';
data20.0='00000'; data20.1='55555'; data20.2='AAAAA'; data20.3='FFFFF';
p4.0='0'; p4.1='5'; p4.2='A'; p4.3='F';

/* the following two statements are relevant only for the xds array of
the fxpt chip since what you write to it is not what you read back from
it */

rdata20.0='0000F'; rdata20.1='1555A'; rdata20.2='2AAA5';
rdata20.3='3FFF0';
rp4.0='0'; rp4.1='5'; rp4.2='2'; rp4.3='7';


tty on
address xed "log " chip'log'

do j=0 to 3              /* test out four different patterns */
     do i=0 to 127   /* 127 is size of largest array on fxpt chip */
   echo 'Currently verifying row ' i 'of all fxpt arrays with pattern '
data32.j
 if i<16 then do
                verify(fxpt,xt0_xtx_$ad,i,data32.j);
                verify(fxpt,xt0_xtx_$ap,i,p4.j);
                        end
        if i<32 then do
                verify(fxpt,xj0_xjr_$ad,i,data32.j);
                verify(fxpt,xj0_xjr_$ap,i,p4.j);
                        end
        if i<64 then do
                verify(fxpt,xt0_xtp_$ad,i,data32.j);
                verify(fxpt,xt0_xtp_$ap,i,p4.j);
                verify(fxpt,xt0_xtq_$ad,i,data32.j);
                verify(fxpt,xt0_xtq_$ap,i,p4.j);
                verify(fxpt,xt0_xtr_$ad,i,data32.j);
                verify(fxpt,xt0_xtr_$ap,i,p4.j);
```

```
                verify(fxpt,xt0_xts_$ad,i,data20.j);
                verify(fxpt,xt0_xts_$ap,i,p4.j);
                verify(fxpt,xt0_xtt_$ad,i,data32.j);
                verify(fxpt,xt0_xtt_$ap,i,p4.j);
                verify(fxpt,xt0_xtu_$ad,i,data32.j);
                verify(fxpt,xt0_xtu_$ap,i,p4.j);
                verify(fxpt,xt0_xtv_$ad,i,data32.j);
                verify(fxpt,xt0_xtv_$ap,i,p4.j);
                verify(fxpt,xt0_xtw_$ad,i,data20.j);
                verify(fxpt,xt0_xtw_$ap,i,p4.j);
                        end
                verify(fxpt,xd0_xdy_$ad,i,data20.j);
                verify(fxpt,xd0_xdy_$ap,i,p4.j);
                verify(fxpt,xd0_xdz_$ad,i,data20.j);
                verify(fxpt,xd0_xdz_$ap,i,p4.j);
                verify(fxpt,xd0_xea_$ad,i,data20.j);
                verify(fxpt,xd0_xea_$ap,i,p4.j);
                verify(fxpt,xd0_xeb_$ad,i,data20.j);
                verify(fxpt,xd0_xeb_$ap,i,p4.j);
                checkit(fxpt,xd0_xds_$ad,i,data20.j,rdata20.j);
                checkit(fxpt,xd0_xds_$ap,i,p4.j,rp4.j);
        end
end
exit
-------------------------------------------------------------------------
verify: procedure
parse arg chip,facility,row,patt
address xed "alter" chip facility"["row"]" "x'"patt"'"
address xed "expect" chip facility"["row"]" "x'"patt"'"
return ""

checkit: procedure
parse arg chip,facility,row,patt,result
address xed "alter" chip facility"["row"]" "x'"patt"'"
address xed "expect" chip facility"["row"]" "x'"result"'"
return ""
-------------------------------------------------------------------------
```

Figure 5:REXX Example

The following example was written by Duane Cawthron to demonstrate some of the different REXX functions and syntax. This program calls the following file for input data.

```
-------------------------------------------------------------------------
/*
@(#) rexexample -- example REXX exec which can be run by ESP
**
**  Remember:  REXX execs must begin with a comment
```

PART 2 -- COMMAND REFERENCE

1.0 COMMAND REFERENCE

The following section describes the various commands supported by the ESP program. The ESP program can execute these "primary" commands, or it can execute ESP command files, or it can execute REXX files. The user should understand the ESP profile and file system structures to understand how to execute extended command file sequences.

1.1 INTERRUPTING THE ESP PROGRAM

Commands can be interrupted by entering the AIX interrupt sequence. Because this sequence is programmable, it can vary from machine to machine. Most machines have a default profile that is programmed to interpret control-C or control-BS as the program interrupt or the break signal. To interrupt the ESP program you must type Control-C on the window from which you invoked ESP.

1.2 RETRIEVE FUNCTION

The ESP command line is part of a MOTIF 'command' widget. The last 8 of up to 100 ESP commands you have entered appear above the command line and may be retreived in several ways. The up arrow and down arrow keys may be used to move to a previously executed command. Just using the arrow keys places the command into the command line where you may press Enter to execute it or may edit the command as desired. You do not have to use the arrow keys at all, but instead, may use the mouse to select a command. One click of the left mouse button selects a command and places it into the command line. (You could then edit the command or just press Enter to execute it.) A double click of the left mouse button not only selects the command but also executes it as if you had pressed enter. (Recommended)

1.3 PRIMARY ESP COMMANDS

!

PURPOSE: Allows AIX shell access from the ESP.

SYNTAX: ! AIXcommand [&]

DESCRIPTION: This ESP command allows execution of an AIX command or commands from ESP. This command is not intended to be a method of gaining a new window to perform AIX actions, but is intended to, for example, allow an ESP script program to run an AIX program.

When called, this function first places the **AIXcommand** into a 512 byte buffer. If the **AIXcommand** is too long a message is printed and the **AIXcommand** is not run, otherwise, ESP executes the AIXcommand in one of two ways, depending on an optional ampersand, "&", flag.

If the ampersand is specified, then ESP 'forks' and performs an 'execlp' of the Korn shell. The -c option is passed to the Korn shell to tell it that the next argument is a command to be performed. The users **AIXcommand** to be performed is passed last.

What this means is that the **AIXcommand** will be performed by the Korn shell in a separate process from ESP (in the background so to speak). ESP will not wait for the command to complete, but continues as soon as the fork is done.

NOTE: When the ampersand flag is used ESP does not know if the AIXcommand was found, completed, or what the return status of it was.

If the ampersand flag is not specified, then ESP uses the system() command to execute the **AIXcommand** specified. ESP then waits for the system() command to complete and sets the **AIXcommand** exit status as the ESP status. This is very useful if an AIX process must complete before ESP should continue.

NOTE: If you must use the question mark, "?", in your AIX program invocation, then AIXcommand must be surrounded by grave accents, "`" to prevent ESP from interpreting the question mark.

NOTE: If you wish to use the semi-colon in your AIXcommand (list) then you must use double quotes surrounding AIXcommand to keep ESP from interpreting the semicolon(s).

EXAMPLES:

To have ESP run a program 'gpib' to cause a GPIB device to be set:

! gpib 5 43 92 (Note: no grave accents or double quotes needed here)

ESP would run gpib and wait for its termination. The exit status of gpib would be available in ESPSTATUS.

To run the same gpib program in the background:

! gpib 5 43 92 &

To grab the TC6xx test card three AIX programs must be run, but sequentially. (That is, subsequent programs must not run until previous programs are finished running.) This can be done two ways from ESP. First, we might create one ESP command with the three AIX commands as the argument, each seperated from the other by semicolons.

! "tcinit; ntstrst; nrst" (Note: double quotes required!)

ESP would wait for all three AIX commands to complete and make the return code from the last AIX command available as status.

The second method is to create three ESP commands:

! tcinit

! ntstrst

! nrst

ESP would run each program and each time wait for the program to complete. Exit status from each program would be available.

RELATED INFORMATION:

none

aet

PURPOSE: Have ESP generate an All Events Trace file

SYNTAX: aet (off|outFile) [iolist]

DESCRIPTION:

ESP can produce an AET file, **outFile**, that has the same format as that produced by TEXSIM. The AET file produced by ESP can therefore be read by any of the standard AET viewers available (e.g. XVS).

On the other hand, ESP is incapable of producing SIGNAL values. Only L2 latch devices and array data can be logged to the aet file.

The *aet* syntax allows three things to happen. Typing *aet* by itself will produce the current status: OFF or ON to file **outFile**. Typing *aet* off will turn aet logging off if it is on, and typing *aet* **outFile** will turn aet logging on to file **outFile**.

When aet logging is turned on, if no **iolist** is specified, then ESP will register the names of every device for every equipped chip into the aet file. If the **iolist**[1] is specified, then only the names in the iolist file will be registered into the aet file.

Once aet logging has been turned on, ESP will generate aet data after completion of either the cs command or the is command. At these times ESP will try to log the current value of all registered devices into the aet file. (But only changes are actually logged in the file.)

IOLIST FILE FORMAT:

The iolist file format understands blank lines and lines beginning with a pound sign, #, as comment lines. Any other lines must be valid.

A valid iolist line consists of three fields that may be seperated by spaces or tabs:

- The Processor Group name
- The Chip name
- The device name and optional bit range in parenthesis

Example iolist file:

```
DEFAULTMUT 620 P1.WAA.IAA.IAC.ICS.ICIQ.L2D
DEFAULTMUT 620 P1.WAA.IAA.IAC.ICS.ICIR.L2D(0:7)
DEFAULTMUT 620 gpr[5]
DEFAULTMUT 620 P1.WAA.IAA.IAC.ICS.ICIT.L2D(3)
```

1. The file format for the iolist file is specified in paragraph labeled "IOLIST FILE FORMAT"

DEFAULTMUT 620 gpr[6](2:4)

DEFAULTMUT 620 P1.WAA.IAA.IAC.ICS.ICIV.L2D

DEFAULTMUT 620 P1.WAA.IAA.IAC.ICS.ICJG.XXX0.L2D_0

EXAMPLES:

To capture output into aet file JONG1 using the iolist illustrated above:

aet JONG1 iolist

To see if aet logging is on:

aet

To turn aet logging off:

aet off

RELATED INFORMATION:

alter

PURPOSE: Allow memory, arrays, or scan string facilities to be modified.

SYNTAX: alter [(mpg|pg)] radix'address' radix'value'

alter [(mpg|pg)] l2[radix'address'] radix'value'

alter [(mpg|pg)] chip array_name[radix'array_address'](bit:range) radix'value'

alter [(mpg|pg)] chip device_name(bit:range) radix'value'

[(mpg|pg)] means an optional parameter which may specify a Multi-Processor Group or a Processor Group. If a MPG or a PG is not specified by the user, then the current PG is accessed, as set by the "pg" command. In single processor environments, this parameter may be safely ignored.

If MPG is specified, then the same chip and device in all PG's of the MPG will be modified.

NOTE: Specifying the optional [(mpg|pg)] for a memory alter is ineffectual.

radix may be any of: x= hex, d= decimal, b= binary, o= octal, a= ASCII, f= float, e= EBCDIC

NOTE: The float is the IEEE single or double precision number format. It is a double word (8 byte) format.

Bit range expressions, as in (bit:range), must be expressed in decimal.

DESCRIPTION: The alter command allows modification of a chip's internal data via the scan strings accessible to ESP. The scan strings are directly accessible from the COP, and arrays and memory can be accessed if ESP does a sequence of COP commands and scan string manipulations

ESP can tell from the alter syntax if the change is to be made on a facility, an array element, on L2 memory or on memory. If there are four arguments then the following device name might be either a facility on the scan string or an array element. If the device name has square brackets surrounding an address, then ESP knows the device is an array element, else the device is just an ordinary facility on the scan string. If the request is made with only three arguments, then the second argument is considered to be the MUT memory address or an L2 memory address. If an L2 memory access, then "l2" and the square brackets inform ESP of this.

ESP accesses chip module scan strings only on demand. That is, ESP will not read (and certainly not write) to or from the MUT until a demand from the user is made via commands, such as alter.

ESP keeps MEMORY IMAGES of the scan strings. When ESP tries to commit the alter data supplied by the user to the memory image of the scan string, ESP checks to assure it has the latest data available. If it does not, then ESP scans in the string, and until something happens to change the state, ESP will work into and out of the memory image of the scan string from then on. If a request to alter is made on array data, then ESP does such work immediately. There is NO cache of any type implemented for array data. An alter for anything less than the full size of an array element will force ESP to do a read-modify-write sequence as well.

If a request to alter memory is made, then ESP could possibly work into a 64 byte buffer cache. This would be possible only if the address of the data to be altered is in the range of a block of 64 characters presently cached in ESP memory.

L2 memory is also cached by ESP into 64 byte buffers.

SPECIAL CONSIDERATIONS:

- Memory access is limited to REAL addresses only.
- Memory access is always performed on 32 bit words, and the address specified should be given on a word boundary (0, 4, 8, 12,.... etc.). Any address specified that is NOT on a word boundary will be modified to the lower word boundary for that address. (ANDed with 0xfffffff0)
- Memory access does not support bit ranges. All 32 bits of the memory address are always operated on.
- Memory is cached in 64 byte blocks by ESP. All alters (and displays) performed by ESP will be done out of the ESP cache if possible. The *flush run*, *iplrun*, *cs*, and *is* commands will flush the cached memory before they execute if the memory cache is holding valid data.
- Memory access within a 64 byte block is treated as read/modify/write.
- Bit range alterations performed on arrays or normal facilities are handled with read/modify/write code.
- A device that is specified without a bit range will be entirely modified. Thus, 'alter chip device 0' will force the entire device to be set to zero. The size of the modification is obtained from the data

specifed by the user. (E.G. x'45' would modify 8 bits). The bits unspecified by the user are simply padded with zeros. (This means that, for example, you could not alter a device to all ones.)

EXAMPLES:

Figure 1: Examples of Alter Commands

```
-----------------------------------------------------------------
alter x'100' x'0f0123'                  Alter memory
alter 601 XD0_XRAA_$AD[15] x'12345678' Alter array
alter 601 XD0_XRAA_$AD[15](0:3) x'1'   Alter array bit range only
alter 601 long 0          Force all bits in the long portion of the
                          scan string to all zeros.

alter b'100' x'11111111' - alter memory address binary 100 to
                             hex 11111111
alter o'10'  x'22222222' - alter address octal 10 to hex 22222222
alter   12   x'33333333' - alter address decimal 12 to hex 33333333
alter x'10'  x'44444444' - alter address hex 10 to hex 44444444

alter l2[x'456'] x'feedbeef' - alter L2 memory at hex address 456
alter l2[119] 77964          - alter L2 memory at decimal address 119

alter x'14'  b'01010101010101010101010101010101' - alter memory
                             address hex 14 to binary value
alter x'18'  o'14631463146' - alter address hex 18 to octal 14631463146
alter x'1c'    2004318071 - alter address hex 1c to decimal 2004318071
alter x'20'  x'8abcdef8' - alter address hex 20 to hex 8abcdef8
alter x'24'  a'3333' - alter address hex 24 to ASCII 3333

alter 601 XP1_XPCC$LD x'12345678'       alter 601 device XP1_XPCC_$LD to
                                       hex 12345678

Multiple Processor Environment (You must specify which PG to modify)

alter pga 601 XD0_XRAA_$AD[15] x'12345678' alter array on processor PGA
alter pgb x'100' x'0f0123'                  alter memory using PGB
pg    pgc                                  change "current PG" to PGC
alter x'100' x'0f0123'                     alter memory using "current PG"
-----------------------------------------------------------------
```

RELATED INFORMATION:

See display, and expect commands.

For single chip considerations, see 'flush'.

For multiple chip considerations, see 'run', 'cs', and 'is'.
For memory mapped io, see 'mmior' and 'mmiow'

autoupdate

PURPOSE: Turn screen update on or off

SYNTAX: autoupdate [(on|off)]

DESCRIPTION:

This command allows the user to test or control screen updates. If the optional keywords "on" or "off" are omitted, then ESP responds by printing the current autoupdate status: "Autoupdate is ON" or "Autoupdate is OFF".

RELATED INFORMATION:

none

beacon

PURPOSE: Blink the LED on an ESP buffer card

SYNTAX: beacon buffer seconds

DESCRIPTION:

Sometimes it's hard to tell which RS/6000 host computer is controlling which ESP buffer card. The *beacon* command assists in finding the physical location of an ESP buffer card by blinking the green LED on the buffer card of **buffer**.

ESP can have up to 16 buffer cards connected to one RS/6000 host adapter. The *beacon* command can be used to tell which buffer card is one, which is two, etc.

When the *beacon* command is executing, no other ESP work is being performed, such as scripts, shared memory or socket interfaces, rexx, or whatever. Basically ESP is tied up blinking the LED for however many **seconds** specified.

RELATED INFORMATION:

cbuf

bells

PURPOSE: Turn ESP Error message sound on/off

SYNTAX: bells [(on|off)]

DESCRIPTION: Certain conditions in ESP are reported as an error message, which means the console bell is sounded. The sound may be turned off with 'bells off', or back on again with 'bells on'. To find out if bells are on or off type 'bells' without any arguments.

RELATED INFORMATION:

none

bp

PURPOSE: Enable or disable breakpoints (execution stop just prior to execution of the instruction at the breakpoint address)

SYNTAX: bp [(mpg|pg)] clear - clears all break-point functions

bp [(mpg|pg)] address - sets breakpoint to 'address' and defaults to soft stop.

bp [(mpg|pg)] -q - queries if a breakpoint is set, and if it is, what that breakpoint address is

bp [(mpg|pg)] address [(hard|soft|int)][virtual|real]

DESCRIPTION: This command enables or disables the break point function. The user can set hard stop, soft stop, or trap to interrupt vector.

A hard stop means that the processor cannot be restarted via ESP 'run' or 'is' commands.

A soft stop means that the processor is 'asked' to stop, and when it has stopped, ESP will take control. The processor is supposed to have quiesced the system it is in and then halted in such a way that it can be re-started.

The 'int' parm sets the 604 such that it will branch to interrupt vector 0x1300 if the breakpoint address is reached. The 604 will not stop.

NOTE: The 604dd1.0 does not have instruction step implemented.

NOTE: After a soft stop (at a breakpoint) instruction steps must be preceded by a call to the ESP command 'bp2is'.

NOTE: Breakpoint, branch trace, and instruction step all share (and set) the 604 stop mode.

NOTE: If using the interrupt vector, the code at the interrupt vector must clear the IABR before it returns or a loop will be set up between the breakpoint address and the interrupt vector address.

RELATED INFORMATION:

is - instruction step

bt - branch trace

bt

PURPOSE: Enable Branch Trace Mode

SYNTAX: bt [(mpg|pg)] [(clear|soft|hard|int)]

DESCRIPTION: This command enables or disables the branch trace function. The user can set hard stop, soft stop, or trap to interrupt vector.

This mode establishes what happens when the 604 executes a branch instruction. The 604 will trap after the branch instruction completion. The trap is set by the user to one of three options: soft stop, hard stop, or trap to interrupt vector.

A hard stop means that the processor cannot be restarted via ESP 'run' or 'is' commands.

A soft stop means that the processor is 'asked' to stop, and when it has stopped, ESP will take control. The processor is supposed to have quiesced the system it is in and then halted in such a way that it can be re-started.

The 'int' parm sets the 604 such that it will branch to interrupt vector 0x0d00 when a branch is taken. The 604 will not stop.

The 'clear' parm allows the user to disengage the branch trace mode.

NOTE: Breakpoint, branch trace, and instruction step all share (and set) the 604 stop mode.

RELATED INFORMATION:

is - instruction step

bp - break point

caa

PURPOSE: Add an array definition to a chip type

SYNTAX: caa chip_type infile

DESCRIPTION: This 'chip add' command is used to add an array definition to a chip type. The 'Array Definition' must be in a file, denoted by 'infile'. This file name can be any valid AIX file name.

These are the rules or guidelines for Array Definitions used by ESP:

- There should be one array definition per file
- The array definition file name should be named after the array so that it is easily found among other array definitions, but the file name should end in the letters '.DEF'.
- Although Array Specification Format allows multiple ports to an array, ESP can only understand one read and one write port per array definition. (If you wish to access the same array using another port, you must create a new array definition with a different array name using the other port.)
- An Array Definition can be for an array that is read only or write only, but if it is read/write, then ESP expects the read and write ports to have the same bit width. Only bits in the long scan string, boundary scan string, or cop self test register can be specified in the array definition.
- Any scan string bits used for array access that are zero need not be specified (ESP creates chip wide scan strings for array access where each bit is zeroed).
- Any scan string bits in the COP scan string, which is inaccessible to ESP, must not be specified in the array definition. ESP can only set COP scan string bits using special COP commands (example: COP scan string bit XAZ_CP$TSTNF$L is set by ESP sending the cop command TSTNF).
- ESP treats READ(CONTROL) and OUTPUT_(CONTROL) devices the same way. Also WRITE(CONTROL) and INPUT_(CONTROL) devices are treated equally. For example, ESP would create a zeroed out long scan string for read access, rlss, and another for write access, wlss. As the array definition is

read, devices mentioned under READ(CONTROL) and OUTPUT_(CONTROL) would cause bits to be set in the read scan string rlss. Devices mentioned under WRITE(CONTROL) and INPUT_(CONTROL) would cause bits in wlss to be set.

- ESP expects devices defining INPUT_0(SOURCE) and OUTPUT_0(SINK) to be previously defined devices. Concatenation are not allowed here. (Some times it is necessary to create a new device, or PSEUDO device, in the Scan Tables, (see ESP command 'cad'), so that it might be referred to in the array definition as an INPUT or OUTPUT.)
- ESP expects array ADDRESS_ devices to be 'whole' as well. If the address is a concatenation of bits from more than one device, then a PSEUDO device should be created and the new PSEUDO device name should be used in the array definition.
- Pseudo devices created for use in array definitions are, by tradition only, placed in a file named in the same manner as to the scan string file. This time, instead of "604ST.x" we would name the file "604CAD.x". '604' because the file defines chip type 604 devices, 'CAD' for chip add device, and '.x' because it is a ESP command file. (Who ever did this the first time used the 'CAD' and we've stuck with it.)
- Observation ports are not used or understood by ESP.

Figure 2: Example of an Array Definition

```
-----------------------------------------------------------------------
/* comments may be used in the array definition file like this */

+BEGIN(ARRAY)

+ARRAY(NAME)
  XD0_XDFB_$AD(128,0:127); /* width must match INPUT_0(SOURCE) and */
   /* OUTPUT_0(SINK) specifications */

+ARRAY(SPEC)
  INPUTS          = 1,
  OUTPUTS         = 1,
  ADDRESS_R       = 1,
  ADDRESS_W       = 1,
```

```
  WIDTH          = 128,
  HEIGHT         = 128,
  WRITE_THROUGH = 1;

+INPUT_0(SOURCE)  /* must be a single device, bit range ok, no concate-
nations */
  TLBIO;       /* Bit width must match ARRAY(NAME) specification */

+INPUT_0(CONTROL)
   ;

+INPUT_0(OBSERVE)
  ;

+ADDRESS_R_0(SOURCE)/*must be a single device, bit range ok, no concat-
enations*/
  XN1_XNAM$LD(13:19);

+ADDRESS_R_0(CONTROL)
  XWF_XWFG$L(0) = B'0',  /* unnecessary but ok */
  XSD_XWFE$L = B'0',     /* unnecessary but ok */
  XSD_XSDV14$L = B'0';   /* unnecessary but ok */

+ADDRESS_R_0(OBSERVE)
  ;

+ADDRESS_W_0(SOURCE)/*must be a single device, bit range ok, no concat-
enations*/
  XDD_XDDJ$L(0:6);

+ADDRESS_W_0(CONTROL)
  ;

+ADDRESS_W_0(OBSERVE)
  ;

+READ_0(CONTROL)
  XF0_XFAE$LH = B'1',
  XF0_XFAE$LD(2) = B'1',
  XF0_XFAE$LD(13) = B'1',
  XF0_XFAC$LD = X'20041000';

+READ_0(OBSERVE)
  ;

+WRITE_0(CONTROL)
```

```
  XT0_XTAC$LH = B'1',
  XT0_XTAD$LH = B'1',
  XDD_XDDL$L(0) = B'1',
  XF0_XFAE$LD(2) = B'1',
  XF0_XFAE$LD(13) = B'1';

+WRITE_0(OBSERVE)
  ;

+OUTPUT_0(SINK)  /* must be a single device, bit range ok, no concate-
nations */
  TLBIO;       /* Bit width must match ARRAY(NAME) specification */

+OUTPUT_0(CONTROL)
XF0_XFAC$LD = X'20041000',
XF0_XFAE$LD(2) = B'1',
XF0_XFAE$LD(13) = B'1',
XF0_XFAE$LH = B'1';

+END(ARRAY)
```

RELATED INFORMATION

cac

cad

cass

cat

cac - chip add chip

PURPOSE: Add an instance of a chip to a system

SYNTAX: cac chip_type chip_name (CBAorJPOS | "port list" ["PG list"])

DESCRIPTION:

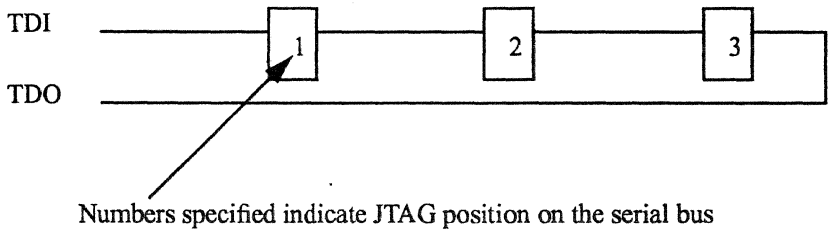
This command is used to declare an instance of a type of chip.

chip_type must be the TYPE of chip that is being created. This is a reference to a chip type created previously with the *cat* command.

chip_name is the name of the chip being created. This name must not collide with Processor Group names or Multi-Processor Group names.

CBAorJPOS is used differently depending on the type of chip. If the chip type uses the COP protocol then *CBAorJPOS* must express the chip COP Bus Address.

If the chip type uses the JTAG protocol then *CBAorJPOS* must express the chip position in the JTAG serial bus where the first chip in the serial line is in position one.



CBAorJPOS is an integer that MUST BE EXPRESSED AS A DECIMAL NUMBER (radix 10).

"port list" declares the new chip's membership in however many ports it might be in. *"portlist"* is a single string (in double quotes) consisting of entries that are seperated by commas. One entry consists of two fields: the port name and the JTAG position of the chip (or CBA of the chip if it is COP), seperated by a colon. (Ports are created with the *caport* command.)

"PG list" declares the new chip's membership in however many Processor Groups it might be in. This field is a single string (in double quotes) consisting of entries that are seperated by commas.

EXAMPLES:

```
---------------------------------------------------------------------
These examples use  simple syntax where no PORT or PG is referenced. All
these chips will default to membership in the DEFAULTMUT Processor Group
and the DEFAULTPORT Port.
cac 603  alpha 1     # The first 603
cac 603  beta  1     # The second 603
cac 603  gamma 1     # The third 603

These examples use the "port list" syntax. Both chips will default to
the DEFAULTMUT Processor Group since no PG is specified.
cac dcu dcu1 "p1:10,p2:1"
cac dcu dcu2 "p1:2,p2:2"

These examples use the "port list" and "PG list" syntax.
cac scu scu "p1:1" "pga"
cac icu icu "p1:2" "pga"

In this system no Port has been specified by the user, as there is only
one port, but there are four Processor Groups.
cac coral coral "DEFAULTPORT:1" "pga,pgb,pgc,pgd"
cac rattler rat "DEFAULTPORT:2" "pga,pgb,pgc,pgd"
cac cobra cobra "DEFAULTPORT:3" "pga,pgb,pgc,pgd"
cac viper viper "DEFAULTPORT:4" "pga,pgb,pgc,pgd"
cac 620 620 "DEFAULTPORT:6" "pga"
cac 620 620 "DEFAULTPORT:5" "pgb"
cac 620 620 "DEFAULTPORT:8" "pgc"
cac 620 620 "DEFAULTPORT:7" "pgd"
---------------------------------------------------------------------
```

RELATED INFORMATION:

Other "chip add" commands such as cat, cass, cad.

cacopcmd

PURPOSE: Add a COP command (or JTAG instruction) to ESP

SYNTAX: cacopcmd mnemonic value

DESCRIPTION: 'This ESP command supports adding COP commands or JTAG instructions to the list of COP and JTAG commands known to ESP. Once the command has been created, the command can be sent to the MUT using the ESP command *cop*. A list of commands known to ESP is available using the *cop -l* command.

The 'value' specified can be in binary, decimal, octal, or hex using the usual ESP syntax. (E.G. x'45')

If *cacopcmd* is used to specify a COP command, then the command must be no more than 8 bits wide. If a JTAG instruction is specified, then the value must be valid within the limit set by the bit width specified in the *cat* command.

EXAMPLES: c

RELATED INFORMATION:

cop, cat

cad

PURPOSE: chip add device

SYNTAX: cad chip_type device_name [expression] definition

DESCRIPTION: This ESP command defines a new device that is a collection of bits from a scan string, another device, or a constant, or, creates a device that is really an expression.

The 'chip_type' must be a chip type that has already been defined using the 'cat' command.

The 'device_name' can be any length but must be composed of the following characters:

Table 1: Valid 'device_name' characters

Characters	Description
a-z	The alphabet in lower case
A-Z	The alphabet in upper case
0-9	The decimal numerals
.	The period character .
_	The underscore character
$	The dollar sign character

The new device can only be created from previously defined objects. (Sorry, no forward referencing.) Array members and memory cannot be part of the new device definition. Devices concatenated together to form the new device must all reside on the same scan string.

If the optional 'expression' keyword is used, then the 'definition' field is expected to contain an expression as a single string. See the section below entitled "Entering Expressions Into ESP".

The 'definition' of a new device can be in three parts where each part is separated from the other by one or more spaces; data parity exception.

The data part of a 'definition' must be present and must be one or more existing scan string based device names or constant values. These devices or constant values may be concatenated together using the double bar symbol '||', however, no blank spaces are allowed between devices being concatenated due to the syntax of the cad command.

The creation of the data part of a 'definition' may specify bit ranges, inverted bit(s) using the tilde symbol, and may concatenate many devices together using the double bar symbol. Constant values may express up to 32 bits and no more! If more than 32 bits of constant value is desired, simply use the double bar to concatenate another constant value. Constants may only be expressed as hex or binary values.

The second part of a 'definition' is the parity device and is optional. If present, it must be separated from the data part of the definition by at least one space. If the parity device is a compound device, that is, composed of more than one device concatenated, then the parity device should be created first using the cad command into a single pseudo device. The pseudo device name should then be used as the parity device for a new cad command.

The third part of a 'definition' is an exception code for the parity device. This field is optional, but if used must fall after the naming of the parity device and must be separated from the parity device by at least one space. ESP supports many kinds of parity, each having a unique id or number.

Since the cad command can be very long indeed, the use of general ESP syntax allowing line continuation is recommended. At any time the cad command may be split to the next line with the back slash symbol, '\'. Be very careful, however, that you do not add any unintentional blank spaces as you use this feature. Blank spaces to the cad command mean change of line parts. (data to parity for example, or parity to exception.)

Entering Expressions Into ESP

The method of describing an expression to ESP will be an extension of the existing CAD command as follows:

cad ctype dname [expression] "a|b&c(your_expression)"

[expression] is a new optional argument to the CAD command telling ESP that what follows is not really a true device, but an expression instead.

Features And Limitations

Once the device is created with the CAD command, it can be used with the DISPLAY command or on a screen with the SAD command.

The ALTER command will not allow an expression device to be modified. Furthur, a screen displaying an expression device will not allow the screen field to be modified by the user.

An expression device may not be concatenated with any other device.

The CAD command will support the following operators in an expression : AND, OR, and NOT. Basic evaluation of an expression will be left to right, with grouping performed by parenthesis.

The symbols to be used for operators are:

Table 2: Valid symbols as expression operators

Symbol	Description
&	(ampersand) for the AND operation
|	(vertical bar) for the OR operation
^	(carot) for the NOT operation
(	(left parenthesis) for beginning a grouping of expressions
)	(right parenthesis) for ending a grouping of expressions

Spaces and tabs in an expression are currently not accepted.

The CAD command will accept any scan string device names, with bit ranges allowed, as well as array names, with bit ranges allowed. References to memory in the expression is NOT allowed. Constants are not supported.

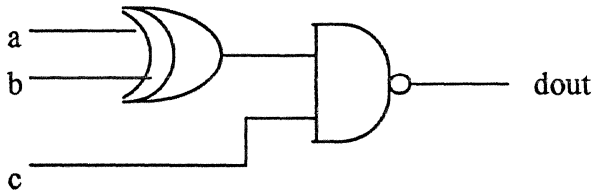
All device operands mentioned in an expression are expected to reside in the same chip.

Only 1 bit operands are supported.

Expression devices are only supported outside of the binary scantable. This means that ESP will come up, read the binary scantable, and finally must read a file(s) with CAD commands that are expressions. CAD devices with expressions may not be saved into a binary scantable file.

Example:

If we have this circuit:



The expression for the circuit should be:

dout = ^(a|b&^(a&b))|^c

To describe the output, dout, as a device in ESP enter the following:

cad 604 dout expression "^(a|b&^(a&b))|^c"

To display the value of the expression (in binary), type:

display 604 dout b

To put the expression on a screen (in binary), type:

sad screen row col b 604 dout

Hierarchy Of Expressions

It is possible to create expression devices that are composed of one or more previously created expression devices.

Other CAD Examples

```
-----------------------------------------------------------------------
cad fpu FBC_ FBA_||~FBB_
  I can refer back to something previously defined and call it
  anything I want.  Concatenate FBA_ and FBB_ inverted.

cad fpu statusreg FBA_
  I can refer back to something previously defined and call it
  anything I want.

cad fpu FB0_FBA_$LD long(480:511)
  Device FB0_FBA_$LD is in the long scan string of the fpu chip and
  is in bit positions 480 through 511.  Bit 480 is to be treated as
```

```
  the high order bit or the most significant bit.

cad fpu FB0_FBL_$LF long(1183:1152)
  Device FB0_FBL_$LF is in the long scan string of the fpu chip and
  is in bit positions 1183 through 1152.  Bit 1183 is to be considered
  to be the high order bit or the most significant bit.

cad fxpt XT0_XVG_$LP long(1245:1248)
cad fxpt XT0_XVG_$LD long(1213:1244) XT0_XVG_$LP 0
  Device XT0_XVG_$LD is in the long scan string of the fxpt chip and
  is in bit positions 1213 through 1244.  This device has parity.  The
  parity is in device XT0_XVG_$LP and has a parity exception code of 0.
  (Normal, even parity)

cad scu OMB_R13$ML_0_23 long(920:943) # 8 2 8 2 data, 4 parity
cad scu OMB_R13$ML_0_23P long(940:943)
cad scu OMB_R13$ML_0_23D long(920:939) OMB_R13$ML_0_23P 12
  Device OMB_R13$ML_0_23 is in the long scan string of the scu chip
  and is 24 bits long.  This device has 20 bits of data (bits
  920 through 939) and has parity.  The parity is in bits 940 through
  943.  The parity is generated by exception rule 12.  That is, the
  first parity bit is even parity over device bits 0 through 7.  The
  second parity bit is even parity over device bits 8 and 9.  The
  third parity bit is even parity over device bits 10 through 17.  The
  fourth parity bit is even parity over device bits 18 and 19.

cad scu OGR_RC$ML_0_23 long(1112:1135) # 1p, 6d, 1p, 6d, of 24
cad scu OGR_RC$ML_0_23P OGR_RC$ML_0_23(0:0)||OGR_RC$ML_0_23(7:7)
cad scu OGR_RC$ML_0_23D OGR_RC$ML_0_23(1:6)||OGR_RC$ML_0_23(8:13) \
OGR_RC$ML_0_23P 14
  Device OGR_RC$ML_0_23 is in the long scan string of the scu chip
  and is 24 bits long.  This device has 12 bits of data (device bits
  1 through 6 and 8 through 13) and has 2 bits of parity (device
  bits 0 and 7).  The parity is generated by exception rule 14.  That
  is, bit 0 is parity.  It is calculated as even parity over bits
  1 through 6.  Bit 7 is parity.  It is calculated as even parity over
  bits 8 through 13.  Bits 14 through 23 are physically in the scan
  string but are not functional.  This allows us to refer to all
  the actual bits in the scan string as OGR_RC$ML_0_23.  We can also
  refer to just the parity bits as OGR_RC$ML_0_23P, and to all the
  data bits as OGR_RC$ML_0_23D.
# pseudo devices for screens
cad icache IB01 IB0_IBH_$LD(0:31)||IB0_IBI_$LD(0:15)
cad icache IB02 IB0_IBJ_$LD(0:31)||IB0_IBK_$LD(0:15)

# pseudo devices for architected names
```

```
cad icache MSR IJ2_IJT_$LD(0:2)||~IJ2_IJT_$LD(3)||\
IJ2_IJT_$LD(4:8)||~IJ2_IJT_$LD(9)||IJ2_IJT_$LD(10:15)
  This creates a pseudo device called MSR which displays two bits
  inverted.

# Use of a constant value in a device definition expression
cad 601 RESRV XD0_XDFT$LD(0:26)||b'0000'||XGA_RSVD$L
-------------------------------------------------------------------------
```

Figure 3: cad example

RELATED INFORMATION:

Other 'chip add' commands such as cat, cass, cac.

campg

PURPOSE: Add a Multi-Processor Group to ESP

SYNTAX: campg MPGname "PGlist" port

DESCRIPTION: This command is used to associate the name *MPGname* with activation of multiple processor groups in *PGlist*.

MPGname is a name to associate with a Multi-Processor Group. The name must not be the same as any CHIP name or PG name, must not be "broadcast" or "bc", and must not begin with a dash or a number. There is no size limit to the name.

PGlist is a single argument that can contain one or more processor group names that should be acted upon whenever *MPGname* is used in an ESP command. The list is a comma delimited list of existing processor groups.

port specifies which PORT should be used to access all the PG at the same time.

NOTE: ESP assumes that activation of *port* will cause every chip of all Processor Groups to be connected to ESP as one JTAG loop or COP bus. Only in this way can ESP send a syncronizing command to all chips of all PG at the same time.

There is no ESP limit to the number of MPG's that you can create. However, a maximum of 256 PG's can be grouped together under one MPG name.

In a single processor environment this command does not need to be used.

EXAMPLES: If there were two processor groups, 'pga' and 'pgb', that could be selected to respond on the COP bus if port 'alphaport' were enabled, then this example would associate selection of these two processors with the name "setAB".

campg setAB "pga,pgb" alphaport

RELATED INFORMATION:

capg, pg, enable, caport

capg

PURPOSE: Add a Processor Group name to ESP

SYNTAX: capg PGname "action"

DESCRIPTION:'PGname' is a name to associate with a Processor Group. The name must not be the same as any CHIP name or PG name, must not be "broadcast" or "bc" or "mut", and must not begin with a dash or a number. There is no size limit to the name.

'action' is a single ESP command to be used to switch ESP to the cop bus of the intended processor.

Processor groups are used to name a group of chips associated with one processing unit. Once a processor group, or PG, is created, chips can be added to ESP mentioning the 'PGname'.This associates the 'action' required to switch ESP to that processor with each chip in the processor complex.

The first 'capg' command issued to ESP will not only create the Processor Group, but will also set the default or current PG (typing 'pg' will show the current PG is the PG just created).

There is no ESP limit to the number of PG's that you can create. However, a maximum of 256 PG's can be grouped together with the 'campg' command (but you can have as many MPG's as you want).

In a single processor environment this command does not need to be used.

EXAMPLES: capg pga "copcmd x'8000'"

RELATED INFORMATION:

campg, pg, enable

caport

PURPOSE: Add a Port to ESP

SYNTAX: caport pname "enable"

DESCRIPTION:'

enable is one or more ESP commands to be used to switch ESP to the COP or JTAG bus of the intended chip. (Multiple commands must be seperated by semicolons.)

There is no ESP limit to the number of PORT's that you can create. However, a maximum of 256 CHIP's is supported by ESP on any one PORT.

NOTE: Chips are associated with PORT's using the ESP command *cac*.

In a single processor environment this command does not need to be used. Any chips created where no PORT's are mentioned will automatically belong to a DEFAULTPORT created by ESP.

EXAMPLES:

caport p1 "! aixpgm"	Select a port by running an AIX program.
caport p2 "enable 3"	Select a port by enabling an ESP buffer

RELATED INFORMATION:

cac

cass

PURPOSE: Add a scan string to a chip type

SYNTAX: cass chip_type ssname #bits persistence "preput=" "postput=" "preget=" "postget=" [scananytime=ok]

DESCRIPTION:. This ESP command not only defines a scan string but also associates it with a particular **chip_type**.

The **chip_type** must be a chip type that has already been defined using the *cat* command.

The scan string name, **ssname**, is used to identify this scan string.

#bits defines how many bits are in the scan string and also defines a possible relationship between this scan string and another. If this scan string is a subset of another scan string then **#bits** must be specified as a bit range in the other scan string. (Example, the boundary scan string is typically a subset of the long scan string. **#bits** for the boundary scan string definition would then be specified as 'long(xxx:yyy)' instead of just a bit count.)

persistence tells ESP if the scan string being defined is to be automatically preserved by ESP during times when the chip is stopped. The value 'P' establishes persistance, while 'NP' means that the scan string is not to be treated in a persistent manner.

preput=, **postput=**, **preget=**, and **postget=** is each a list of zero or more COP commands that will be executed by ESP when the scan string is either written into the chip, i.e. 'put', or read from the chip, i.e. 'get'. The list syntax has the following rules:

1) Upper or lower case is supported

2) COP commands may be delimited with commas, spaces, or semicolons

3) Up to 16 'pre' commands may be specified, and up to 8 'post' commands may be specified. The COP commands specified will be executed in the order given, starting with the COP command closest to the equal sign.

4)If there are no COP commands to execute then the list keyword should be specified but no COP commands included. (Example: **"preput="**)

5) Lists must be enclosed in double quotes.

The list of COP commands in **preput=** is executed just prior to sending bits into the chip. The list of COP commands in **postput=** is executed just after sending bits into the chip.

The list of COP commands in **preget=** is executed just prior to receiving bits from the chip, and the list of COP commands in **postget=** is executed just after receiving bits from the chip.

Optional flag **scananytime=ok** tells ESP that it is ok to scan the indicated scan string at any time, even if the chip is running. (However, the scan anytime scan string will still not be scanned if OCS is defined and currently on.)

NOTE: ESP cannot access the entire COP scan string.

EXAMPLES:

```
----------------------------------------------------------------------
cass 603 long 6534 P \
        "preput=FFRZ,A_RNW,HIZ,LSRL" \
        "postput=RS_A_RNW,RS_HIZ" \
        "preget=FFRZ,A_RNW,HIZ,LSRL" \
        "postget=RS_A_RNW,RS_HIZ"

cass 603 exmem long(0:321) P \
        "preput=FFRZ,A_RNW,HIZ,EXMEM" \
        "postput=RS_A_RNW,RS_HIZ,RS_EXMEM" \
        "preget=FFRZ,A_RNW,HIZ,EXMEM" \
        "postget=RS_A_RNW,RS_HIZ,RS_EXMEM"

In this example FFRZ, A_RNW, HIZ, EXMEM, RS_EXMEM, and LSRL are all COP
commands known to ESP.

The exmem scan string is a subset of the long scan string and is 322 bits
long.
----------------------------------------------------------------------
```

Figure 4: cass example

RELATED INFORMATION:

caa

cac

cad

cat

cat

PURPOSE: Adds a chip type to ESP.

SYNTAX: cat chip_type [(COP|JTAG)] [regsize] [IO]

DESCRIPTION:

This command defines a type of chip to ESP, of which there may be many chips created using the *cac* command.

The CAT command informs ESP that there will be future references to the **chip_type** name mentioned. Scan strings and arrays will be added to the chip type, using the **chip_type** name. Eventually a chip will be created modeled after this chip type.

When the chip type is created it is also designated as a chip type that will support either the COP or the JTAG protocol. If the chip type supports JTAG protocol, then the **regsize** may specify the JTAG Instruction Register width in bits. (default=16)

If the chip should be handled as an IO chip, then the **IO** paramater should be specified. To date, this means that ESP will not send a clock-stopping FFRZ command to IO chips after instruction steps or stops occuring after RUN or IPLRUN. It also means that IO chips will not be scanned unless the *ioflag* command is used to set the chip status to STOPPED.

EXAMPLES:

```
-----------------------------------------------------------------
cat 601 cop
                    # Creates a chip type called '601' that is COP pro-
                    tocol.
cat 603 jtag 8
                    # Creates a chip type called '603' that is JTAG,
                    and has an 8 bit JTAG instruction register.
cat rattler jtag 8 IO
                    # Creates a chip type called 'rattler' that is
                    JTAG, has an 8 bit JTAG instruction register, and
                    is also an IO type of chip.
-----------------------------------------------------------------
```

RELATED INFORMATION:

caa, cac, cad, cass or other "chip add" commands, and *ioflag* used to specify IO type chip run/stopped status.

cbuf

PURPOSE: Associate a buffer card with a PG and assign buffer card pins

SYNTAX: cbuf BufferNumber PG pinconfig

DESCRIPTION: This ESP command configures a hardware buffer card's physical pin arrangement.

The buffer card to be operated on is 'BufferNumber', an integer from 1 to 16, where buffer card one is the buffer card closest to the ESP feature card plugged into the host computer.

Any one buffer card can support more than one Processor Group, or PG. Any one 'cbuf' command can specify only one PG, but there can be multiple 'cbuf' commands executed, each specifying a different PG, if desired.

The 'pinconfig' argument is used to specify the physical pin configuration to be used between the buffer card and the MUT. Two default pin configurations are supported: COP and JTAG.

If you specify 'pinConfig' with the word "COP" then the pins will be configured as follows:

1=cs, 2=hr, 3=sr, 4=ctl, 5=clk, 6=rs, 7=si, 8=so, 14=pg, 15=ocs

If you specify 'pinConfig' with the word "JTAG" then the pins will be configured as follows:

1=cs, 2=hr, 3=sr, 4=tms, 5=tck, 6=rs, 7=tdi, 8=tdo, 14=pg, 15=trst

If you wish to assign physical pins to their functions then you may do so with the following rules:

1) This command understands sixteen pins, 1 through 16.

2) Command syntax requires the pin number followed by an equal sign followed by a reserved word describing the pin follwoed by a comma if there are more pins to be defined. Example: "5=trst, ..., 14=si". Spaces and new lines may be employed to make the pin assignment list more readable, but new lines must be preceeded by the back-slash character (the normal ESP command line syntax requirement for line continuation.)

3) Table 1 presents reserved words which may be used to assign a pin to a function:

Table 3: Pin Mnemonics

Mnemonic	Meaning
trst	JTAG reset
tms	JTAG mode select
tck	JTAG clock
tdi	JTAG serial data in
tdo	JTAG serial data out
ctl	COP control
clk	COP clock
si	COP serial in
so	COP serial out
cs	checkstop
rs	run/stop
pg	power good
hr	hard reset
sr	soft reset
ocs	ocs override

EXAMPLES:

Use of default COP pin assignments

```
cbuf 1 DEFAULTMUT COP
```

Assignment of pins on a buffer card

```
cbuf 2 DEFAULTMUT \
   12=ctl, \
   14 = ctl , \
   2  =  si, \
   3= so, \
   7= cs, \
```

8= rs, \
10= hr , \
16=sr

RELATED INFORMATION:

none

cd

PURPOSE: Change ESP current working directory

SYNTAX: cd directory

DESCRIPTION: This command allows the user to change ESP's notion of the current working directory.

RELATED INFORMATION:

none

cecho

PURPOSE: echo command and command line arguments

SYNTAX: set cecho - turns cecho on

unset cecho - turns cecho off

DESCRIPTION: This command echos command and command line arguments.

RELATED INFORMATION:

none

clearbreak

PURPOSE: Clear any break signals from the user

SYNTAX: clearbreak

DESCRIPTION: ESP counts the number of times that the user presses the break key. Clearbreak resets the count to zero. Before using the userbreak command, clearbreak should be called.

RELATED INFORMATION:

userbreak

cls

PURPOSE: To clear the TTY window.

SYNTAX: cls

DESCRIPTION: This command clears everything from the TTY window.

RELATED INFORMATION:

none

configure

PURPOSE: Configure ESP hardware

SYNTAX: configure path

DESCRIPTION: This command causes the ESP hardware to become configured. The 'path' specified must point to the directory where ESP 'bit' files are located. These files are used by the ESP hardware.

RELATED INFORMATION:

none

cop

PURPOSE: Send low level instructions or data to MUT modules.

SYNTAX: cop -l (lists command mnemonics available)

cop [port] (DATA | DATA_EOF) (x | b | a) 'value'

cop (CMD | CMD_EOF) (x | b | a)'value'

cop [(mpg|pg)] (chip_name | BROADCAST | bc) command

DESCRIPTION:

Optional parameter [(mpg|pg)] specifies the name of a Multi-Processor Group or processor group to which the COP command will be sent. If this parameter is omitted, the "current PG" will be used, as specified by the user with the "pg" command. Notice that this is a case where more that one processor can be addressed with the same command using a MPG name.

It is possible for ESP to be connected to a system where there are multiple JTAG or COP circuits. In ESP each circuit is called a port. A port must be enabled to become connected to the ESP connector.

When cop data, cop data_eof, cop cmd, or cop cmd_eof are used, no chip is specified since the object of the command is to directly exercise the COP or JTAG bus. By default, such a command will be sent out the DEFAULTPORT. This means that the enable for DEFAULTPORT will be executed to configure the DEFAULTPORT onto the ESP connector.

For more information about ports, see the 'caport' and 'cac' commands.

If the port argument is specified then that port is enabled instead of DEFAULTPORT before the COP or JTAG command or data is placed onto the ESP connector.

The lowest level communication from the ESP program to the Machine Under Test (MUT) is performed over the COP bus or JTAG bus of the MUT. This communications consists of either commands or data.

For the COP bus, a protocol is established where ALL COP commands will start with the COP bus address of the chip module to be communicated with, and the opcode of the COP command to be done. If data is to be passed, it follows at this time.

For the JTAG bus, a protocol is established where the JTAG instruction is sent to the chip module to be communicated with, and if data is to be passed it follows at this time. (When a particular chip module is being communicated with, all other JTAG chips are placed into the BYPASS mode.)

NOTE: Persons using this command should be very familiar with COP functions. Entering COP commands in the wrong order can cause hardware damage (OCDs can be burned out).

Most ESP functions hide the COP commands used to get the job done. However, low level debug REQUIRES such functionality.

Since there are a great many COP commands, it is convenient to refer to them by name. Therefore, the ESP supports mnemonic naming of all COP commands. To get a list of such names, use the -l option of the cop command. The mnemonics that were used for the COP commands were the same ones used by the MAC simulation test cases.

When a COP command is sent to the MUT, it must start with the COP bus address of the chip involved. Each MUT chip has its own unique COP bus address, but the COP protocol allows the use of address zero as a BROADCAST address to all chips. ESP allows you to specify the chip's name, or the word BROADCAST to address each chip individually, or as a group.

Some COP commands require that data be passed along. Such commands are LSRL and LD_CP. The form of the 'cop' command that allows you to deal with data is 'cop DATA value'. The DATA keyword, or DATA_EOF keyword, not only tells ESP about sending the END sequence after the data, but also tells ESP you are using this form of the command. The value must be either hex, binary, or an ascii representation of the data you will send, or a hex, binary, or ascii dummy argument to specify the number of bits to collect from a receive only type of command.

JTAG data passed with the 'cop DATA value' syntax leaves the JTAG chip module in the SHIFT-DR state. Further 'cop DATA value' commands will be interpreted by the chip module as more JTAG data. The 'cop DATA_EOF value' command returns the JTAG controller back to the RUN-TEST/IDLE condition.

The 'cop CMD value' and 'cop CMD_EOF value' forms of the syntax allows sending/receiving bits on the JTAG bus while in the SHIFT-IR state. The first 'cop CMD value' command places the chip(s) into the SHIFT-IR state. Subsequent 'cop CMD value' commands start from

the SHIFT-IR state and leave the JTAG state there when finished. The 'cop CMD_EOF value' syntax causes the chip(s) to return to the RUN-TEST/IDLE condition.

EXAMPLES:

Get a list of all COP/JTAG mnemonics understood by ESP

```
cop -L
```

Send the Force Freeze command to all chips

```
cop broadcast ffrz
```

Send the Halt command to the 604 chip

```
cop 604 hlt
```

Send 33 bits through the boundary scan string of 604

```
cop 604 smpl_pld
cop data b'1'
cop data_eof x'ffffffff'
```

Multiple Processor Environment

In the following examples 'blue', 'red' and green' are processor groups and 'all' is a multi-processor group consisting of PG's blue, red, and green.

```
cop blue   604 run_abist        - 604 chip    in blue  PG run_abist
cop red    604 ffrz              - 604 chip    in red   PG ffrz
cop green  bc  ffrz              - all chips   in green PG ffrz
pg blue                         - set "current PG" to blue
cop        bc  ffrz              - all chips in "current PG" ffrz
cop all    bc  ffrz              - all chips in all PG's ffrz
```

Figure 5: Examples of cop commands

RELATED INFORMATION:

none

This page intentionally left blank.

This page intentionally left blank.

copcmd

PURPOSE: Allow any 16 bits to be sent on the COP bus as a COP command

SYNTAX: copcmd data

DESCRIPTION: A COP bus command is always 16 bits long, and is sent with specific protocol over the COP bus. This ESP command allows the user to send any 16 bits on the COP bus as a COP command sequence.

'data' is any 16 bits and may be expressed with any of the normally supported ESP radix.

A current need for this command is to send a particular bit sequence to a device known as a cop switch. These bits would force the cop switch to select one or more processors onto the COP bus.

NOTE: This ESP command is not implemented for JTAG chips.

EXAMPLE: copcmd b'1010000101010001' or copcmd x'1234'

RELATED INFORMATION:

none

coplog

PURPOSE: logs commands that go to the cop

SYNTAX: coplog on - turns cop logging on, level 3 log

coplog 1 - turns cop logging on, level 1 log

coplog 2 - turns cop logging on, level 2 log

coplog 3 - turns cop logging on, level 3 log

coplog 4 - turns cop logging on, level 4 log

coplog 5 - turns cop logging on, level 5 log

coplog off - turns cop logging off

DESCRIPTION: This command will cause every command that goes to the interface card to also go to a file. This allows a designer to see the exact sequence of command that were sent to å chip, or that are planned to be sent to a chip.

LEVEL 1 logs all cop commands

LEVEL 2 Same as LEVEL 1 plus hex values of scan strings in the COP_DATA format

LEVEL 3 Same as LEVEL 1 plus the equivalent alter commands that would generate the scan strings.

LEVEL 4 logs all of the above. All cop commands, hex values of scan strings, and the equivalent alter commands that would generate the scan strings

LEVEL 5 logs all of the above with addition comments to provide clarification.

RELATED INFORMATION:

none

copstub

PURPOSE: Assign fake status to ESP when chip or driver is not present

SYNTAX: copstub [on | off | fakestatus]

DESCRIPTION: This command can be used to "fake-out" ESP concerning MUT status. Normally, ESP would fetch real status from the MUT and use it to determine if the MUT is powered up, running or stopped, check-stopped, etc. This command allows the user to specify any status that could be returned by the real thing.

Please refer to the *wait* command for definition of bit values.

EXAMPLE *copstub* x'550' - sets STOPPED no CHECKSTOP, and POWER-GOOD.

copstub on - turns the *copstub* function on with a value of 0x550

RELATED INFORMATION:

The *wait* command defines the status bits that can be set by *copstub*.

cs

PURPOSE: Force MUT to run N cycles and stop

SYNTAX: cs [(mpg|pg)] [(-n|n)]

DESCRIPTION:

NOTE: This is a "high level" command which means that any cached scan strings or memory will be flushed to the MUT before the MUT is set running. When the MUT stops ESP screens will be updated.

Causes up to 2^24 processor cycles to be executed. Once 'n' cycles has been run by the processor, the processor simply stops. This is considered a 'hard' stop, which means the processor may not be able to continue running once 'cs' has been issued.

'cs -n' can be used to set how many cycles will be run the next time 'cs' is executed. (No cycles are run at the time the dash-n option is used.)

NOTE: Don't forget to clear a breakpoint before trying to cycle-step off of an address where a breakpoint is set.

cs does not enforce what phase the cycles are started from. If L2PHASE has been set then performing *cs 2.5* will cause C2/C1/C2 cycles to be run. If RS_L2PHASE has been executed, the cs 2.5 will cause C1/C2/C1 to be run. Mixing the use of L2PHASE with the cycle count allows starting the cycle-step on either clock, and odd or even number of clocks to be executed.

EXAMPLES:

cs 25 Processor runs 25 cycles and stops.

cs -14 The next time 'cs' is executed it will perform 14 cycles

cs Perform previously set number of cycles and stop

cs 2.5 Execute 2 and a half cycles.

RELATED INFORMATION:

load_cntr

dirty

Read From Proc into mem of ESP

PURPOSE: Forces ESP to invalidate any buffered MUT data

SYNTAX: dirty [(mpg | pg)] (chip_name | broadcast | bc | mem | l2) [SSname]

DESCRIPTION: If the argument **broadcast, or bc,** is passed, then ESP will mark invalid any buffered scan string images or external memory that it has for all chips (optionally restricted by the **mpg|pg** argument).

If a chip name is passed then only that chips' data is marked as invalid.

If optional argument **SSname** is specified, then only that scan string is marked invalid.

NOTE: The SSname argument only works when a chip_name is specified.

After the *dirty* command scan strings or memory will be read into ESP only on the NEXT DEMAND for such data. Such a demand might come from a command file, a user request, a screen that is up, or whatever else that might be a demand source to ESP.

NOTE: Buffered data is merely marked invalid. No scanning takes place due to this command.

In full system mode, meaning that all chips are working on the CEC (Central Engineering Complex), ESP will normally take care of such data manipulation.

EXAMPLES:

dirty 604	- invalidates ESP's copy of all 604 scan string data
dirty mem	- invalidates ESP's cached memory
dirty l2	- invalidates ESP's cached L2 memory
dirty 604 long	- invalidates only the 604 long scan string

RELATED INFORMATION:

See ESP command *flush*, used in single chip testing also.

display

PURPOSE: Allow scan string facilities, arrays, or memory to be displayed.

SYNTAX: display [pg] address radix

display [pg] which_chip array_name [array_address] (bit:range) radix

display [pg] which_chip device_name(bit:range) radix

radix may be any of:

x= hex, d= decimal, b= binary, o= octal, a= ASCII,

f= float, e= EBCDIC, i= instruction decode

DESCRIPTION: The display command allows display of machine under test (MUT) internal data via the scan strings accessible to ESP. MUT scan strings are directly accessible from the COP, and arrays and memory can be accessed if ESP does a sequence of COP commands and scan string manipulations.

ESP can tell from the display syntax if it is to display a facility, an array element, or MUT memory. If there are four arguments then the following device name might be either a facility on the scan string or an array element. If the device name has square brackets surrounding an address, then ESP knows the device is an array element, else the device is just an ordinary facility on the scan string.

If the request is made with only three arguments, then the second argument is considered the MUT memory address.

ESP accesses chip module scan strings only on demand. That is, ESP will not read (and certainly not write) from or to the MUT until a demand from the user is made via commands, such as display.

ESP keeps MEMORY IMAGES of MUT scan strings. When ESP tries to read the data from the device specified by the user from the memory image of the scan string, ESP checks to assure it has the latest data available. If it does not, then ESP scans in the string, and until something happens to change the state, ESP will work into and out of the memory image of the scan string from then on.

If a request to display is made on array data, then ESP does such work immediately. There is NO cache of any type imple-mented for array data. If a request to display less than the full size of an array element is made, the full array element is fetched, and then the display is performed on only the bits specified.

If a request to display MUT memory is made, then ESP could possibly work out of a 64 byte buffer cache. This would be possible only if the address of the data to be compared is in the range of a block of 64 characters presently cached in ESP memory.

EXAMPLES: If display is called directly, an English statement will be displayed on the status line. For example, "display 604 counter b" will show the following:

604 counter == b'00000000000000000'

Figure 6: Examples of Display Command

```
-----------------------------------------------------------------
display b'100' x - display memory address binary 100 in hex
display o'10'  x - display memory address octal 10 in hex
display   12   x - display memory address decimal 12 in hex
display x'10'  x - display memory address hex 10 in hex

display x'14'  b - display memory address hex 14 in binary
display x'18'  o - display memory address hex 18 in octal
display x'1c'  d - display memory address hex 1c in decimal
display x'20'  x - display memory address hex 20 in hex
display x'24'  a - display memory address hex 24 in ASCII
display x'80'  i - Decode and display the instruction at hex 80

display 604 XRA_XRAA_$AD0[16](0:31)
display 604 XP1_XPCC$LD(0:31) a
-----------------------------------------------------------------
```

RELATED INFORMATION:

See expect, and alter commands.

For single chip considerations, see 'dirty'.

setvar

drtrymode

PURPOSE: Match the ESP to the hardware configuration

SYNTAX: drtrymode [(on|off)]

DESCRIPTION:

This command is used to force ESP to match the 604 hardware configuration. When the 604 was hard-reset, pin drtry_ was sampled by the 604 and the pin state forced the 604 to be in the drtry- mode or not. If the drtry_ line was high, then the drtry-mode was turned on in the 604. ESP must then have its drtry-mode turned on using this command.

Typing *drtrymode* without arguments will cause ESP to report the mode it is currently in. Typing *drtrymode* **on** will turn on the ESP drtry-mode (but not affect the 604 chip) and typing *drtrymode* **off** will turn off the ESP drtry-mode (but not affect the 604 chip).

NOTE: If it were possible ESP would match the 604 drtry-mode automatically.

The effect of the drtry-mode on ESP is to force ESP to swap data during ICACHE or Data Queue array write accesses if the drtry-mode, the 32bit-mode, and the reduced-pin-mode are all off.

RELATED INFORMATION:

ESP commands *32bitmode* and *reducedpinmode.*

dump

PURPOSE: Upload from machine under test (MUT) memory to RS6000 disk.

SYNTAX: dump from_address #_bytes filename

DESCRIPTION: This command allows the user to get an image of MUT memory collected and stored onto the RS6000 disk as a file. The file will have a TOC96 header so that the file can later be loaded back into MUT memory with the ESP load command,

The from_address argument can be in any of octal, hex, decimal, or binary. The address specified is the REAL address and is the starting location from where the dump will commence.

The #_bytes argument can be in any of octal, hex, decimal, or binary. The number of bytes to dump includes the start address.

The filename can be any valid AIX filename and it will have the following characteristics: The first 96 bytes of the file will be zeros except bytes 4, 5, 6, and 7, which will contain the starting address as specified with the from_address argument. The remaining bytes of the file will be the words fetched from MUT memory. The 96 byte header is used by the ESP load command to allow the file to be loaded into MUT memory.

EXAMPLES:

Figure 7: Examples of Dump Command

```
dump x'f0b4' 5 avp1
```

Put the 5 bytes starting at address x'0000f0b4' to the RS6000 disk file named 'avp1'. The dump will be from x'0000f0b4' to x'0000f0bb', or 8 bytes, to force the dump to word boundaries.

```
load avp1
```

The load command to put 'avp1' into MUT memory at the address it came from.

```
dump x'f001' 1  patch1
```

Put the byte at address x'0000f001' to the RS6000 disk file named 'patch1'. The dump will be the four bytes of the word at address x'0000f000' which includes the byte specified.

```
load patch1
```

The load command to put 'patch1' into MUT memory at the address it came from.

```
dump x'f000' x'100' patch2
```

Put the 256 bytes starting at address x'0000f000' to the RS6000 disk file named 'patch2'. The dump will be from x'0000f000' to x'0000f0fc', or 256 bytes, as specified since the address and number of bytes are on word boundaries.

```
load patch2
```

The load command to put 'patch2' into MUT memory at the address it came from.

```
load -vx'0000bbbb' patch2
```

Load 'patch2' at an address other than from where it was dumped from. In this case, patch2 was dumped from x'0000f000' and the load command will cause patch2 to be loaded at x'0000bbbb'. If patch2 contains executable code, it must be relocatable to work at the new address, as ESP does nothing more than copy the patch2 file into MUT memory at whatever address is specified.

RELATED INFORMATION:

load

dynload

PURPOSE: Dynamically load a user provided function

SYNTAX: dynload command path/file

DESCRIPTION:

Use this command to load specially compiled code into ESP while ESP is running.

'command' is the name of the entry function into your code. Once the code is loaded into ESP, it may be executed just as any other built-in ESP command.

'path/file' is the complete path and file name of the executable module to be dynamically loaded into ESP.

EXAMPLES:

If you created a load module in your home directory with a file name of func1 and an entry function called timeOfDay, then you could load your code into ESP with the following syntax:

dynload timeOfDay $HOME/func1

RELATED INFORMATION:

The ESP Training Manual has a write up and examples. The manual is located in /afs/awd/public/esp/userdoc. A demo of how to write a user function is located in /afs/awd/public/esp/rios/toolSource/dynamicLoading.

echo

PURPOSE: Print text on the TTY window

SYNTAX: echo arg1 arg2 arg3 arg4

DESCRIPTION: Echo prints its command line arguments on the screen. If logging is turned on, echo will also print to the log file. Echo is similar to the AIX echo command.

RELATED INFORMATION:

none

enable

PURPOSE: Enable an ESP buffer card to talk to its MUT

SYNTAX: enable bufnum [bufnum ... bufnum]

DESCRIPTION: This ESP command supports the use of multiple ESP buffer cards connected to the same ESP program. It is assumed that each buffer card is connected to a different processor, therefore, this command would be used in a multi-processor environment.

In a MP environment we might wish to talk to any one individual processor, or PG, or to several PG at the same time. To communicate with a PG requires activation of the ESP buffer card connected to it with this command.

When establishing a MP environment, ESP commands capg and campg are used to describe the processor groups involved in the setup. An argument in each of these commands is "action". This argument must provide the ESP command to perform to select and activate the PG or MPG involved. The 'enable' command might be an appropriate "action" to perform.

RELATED INFORMATION:

capg, campg

equip

PURPOSE: Tells ESP what chips are present on the machine under test

SYNTAX: equip [pg] [chip_name [(true|false)]]

DESCRIPTION: If the optional parameter **chip_name** is omitted, then ESP lists those chips currently equipped.

If the optional parameter **chip_name** is present, then ESP knows that the chip is present or not as specified with the true or false statement.

If the optional parameter **pg** is present, then the chip is considered part of a Processor Group. In a multi-processor environment, each CPU will be given a Processor Group name and each chip will be associated with a Processor Group. If the optional parameter **pg** is omitted, then the CURRENTPG is assumed by ESP. The CURRENTPG is set by the user with the *pg* command.

When a chip is created, using the *cac* command, the chip is automatically considered (by ESP) to be equipped. It is useful to un-equip a chip if the chip is not physically present in a system.

NOTE: When a JTAG chip is un-equipped, its JTAG data path is believed (by ESP) to be shunted around as if wired-through. The chip will no long be considered part of the JTAG port and will not take even a bypass bit position.

The *equip* command is used to tell ESP which chips are present on the planar. Although a Processor Complex might have N chips, only some of them might be plugged in on a particular machine under test.

EXAMPLES:

equip	-lists all equipped chips in all Processor Groups
equip pgc	-list all equipped chips in PG pgc
equip 603	- equiv to "equip 603 true"
equip 620 true	- equiv to "equip 620"
equip 620 false	- un-equip the 620 chip
equip pga 603	-equip 603 chip in Processor Group pga
equip pgb 603	-equip 603 chip in Processor Group pgb
equip pgc 604 false	- un-equip the 604 in PG pgc
pg pbc	- set "current PG" to pgc
equip 603	- equip 603 chip in CURRENTPG, pbc

RELATED INFORMATION:

none

err

PURPOSE: Simulate an error condition

SYNTAX: err

DESCRIPTION: Used for ESP program debug. This command is for ESP tool developers only. It returns an error return code for analysis as well as printing the message "err: simulated error" with error number 0179.

RELATED INFORMATION:

none

esp

PURPOSE: Start the ESP program

LIBRARY: none.

SYNTAX: esp [-a] [-d] [-n] [-t] [-c] [-h]

DESCRIPTION: Starting the ESP program can be done in one of two supported ways. Starting the program by running the esp shell script, or starting the program directly.

The actual esp program executable will be named something like "esp604dd1.0ver1.0", and my be run directly from the AIX command line. If started in this way it is the users responsibility to first set some AIX environment variables that the esp program needs to run successfully.

The esp shell script sets all AIX environment variables that need to be set before running the esp program executable.

ESP PROGRAM OPTIONAL ARGUMENTS

Starting ESP from the AIX command line or from the ESP shell script supports using the following ESP optional arguments:

-a Set the socket address ESP will server on. The default socket address served by ESP is 0xbf00 (or 48896 decimal). A client program connecting to this socket can execute ESP commands through the socket. (For more information see documentation about the ESP Socket Programming Interface.) This socket address must be different for each ESP program running on the same host machine.

-d# Set which device driver ESP is to use. The default device driver that ESP will open is "/dev/epesp0". This option allows specification of alternate device drivers. E.g. '-d 2' means ESP should use "/dev/epesp2". Each ESP running on the same host machine must use a different device driver.

This option also establishes the base address for shared memory support. For more information see documentation about the ESP Shared Memory Programming Interface.

-n This flag tells ESP that no device driver is to be used at all. ESP will come up but it will not even try to open a "/dev/dash#" device driver. This allows users to run ESP on a host machine that is not even hooked up to a Machine Under Test (MUT), or that even has a device driver installed. This configuration is frequently usefull for debug purposes such as screen development and external pro-

gram development.

-t This flag tells ESP to run in the TTY mode. That is, ESP will behave with a command line interface. Screens will not be supported in this mode, and all input and output are from/to the standard input and standard output.

-c "command" These arguments allow the use of ESP for the purpose of executing one ESP command. Once this command is finished execution, ESP stops running.

-h This flag tells ESP to print its short help message to the standard output and then exit. A brief list of ESP optional arguments is presented.

ESP PROGRAM AIX ENVIRONMENT VARIABLES AND THEIR USE

LIBPATH Set path to location of librexx.a and other libraries.

NLSPATH Path to National Language Support files, particularily the REXX support file, rexxaix.cat.

XAPPLRESDIR Path to the ESP program's X application resource file, "EspResources".

ESPAVPBIN Path to load files (using the *load* command.)

ESPPATH Path to ESP command files. This would be the location(s) of various ESP screen and other .x files.

RELATED INFORMATION:

none

exit

PURPOSE: exit program

SYNTAX: exit

DESCRIPTION: This command is used to exit to the next higher level of program or command file. If already at level 0 then this command will cause ESP to terminate. Usually this command is embedded in an ESP command file, or 'batch' file, to early-terminate what the command file is doing. If this command is typed from the ESP command line by the user, then obviously there is no next higher level of file to, and ESP terminates.

RELATED INFORMATION:

quit

expect

PURPOSE: Allow scan string facilities, arrays, or memory to be compared.

SYNTAX: expect [pg] address radix'value'

expect [pg] which_chip array_name array_address(bit:range) radix'value'

expect [pg] which_chip device_name(bit:range) radix'value'

radix may be any of: x= hex, d= decimal, b= binary, o= octal, a= ASCII, f= float, e= EBCDIC

DESCRIPTION: The expect command allows comparison of the machine under test (MUT) internal data via the scan strings accessible to ESP, to test data. MUT scan strings are directly accessible from the COP, and arrays and memory can be accessed if ESP does a sequence of COP commands and scan string manipulations. The optional 'pg' argument allows you to specify which processor group the expect is to be performed on.

ESP can tell from the expect syntax if the compare is to be made on a facility, an array element, or on memory. If there are four arguments then the following device name might be either a facility on the scan string or an array element. If the device name has square brackets surrounding an address, then ESP knows the device is an array element, else the device is just an ordinary facility on the scan string.

If the request is made with only three arguments, then the second argument is considered the MUT memory address.

ESP accesses chip module scan strings only on demand. That is, ESP will not read (and certainly not write) from or to the MUT until a demand from the user is made via commands, such as expect.

ESP keeps MEMORY IMAGES of MUT scan strings. When ESP tries to read the data from the device specified by the use from the memory image of the MUT scan string, ESP checks to assure it has the latest data available. If it does not, then ESP scans in the MUT string, and until something happens to change the state, ESP will work into and out of the memory image of the scan string from then on.

If a request to expect is made on array data, then ESP does such work immediately. There is NO cache of any type implemented for array data. If a request to expect less than the full size of an array element is made, the full array element is fetched, and then the expect is performed on only the bits specified.

If a request to expect MUT memory is made, then ESP could possibly work out of a buffer cache. This would be possible only if the address of the data to be compared is in the range of a block presently cached in ESP memory.

EXAMPLE

Figure 8: Examples of Expect Command

```
-----------------------------------------------------------------
expect b'100' x'11111111' - expect memory address binary 100 to be
                            hex 11111111
expect o'10'  x'22222222' - expect address octal 10 to be hex 22222222
expect   12   x'33333333' - expect address decimal 12 to be hex 33333333
expect x'10'  x'44444444' - expect address hex 10 to be hex 44444444

expect x'14'  b'01010101010101010101010101010101' - expect address
                              hex 14 to be binary value
expect x'18'  o'14631463146' - expect address hex 18 to be
                                 octal 14631463146
expect x'1c'    2004318071 - expect address hex 1c to be
                                 decimal 2004318071
expect x'20'  x'88888888' - expect address hex 20 to be hex 88888888
expect x'24'  a'3333' - expect address hex 24 to be ASCII 3333

expect 604 SP0_SPN_$LD x'1234' - expect 604 device SP0_SPN_$LD to be
                                   hex 1234
-----------------------------------------------------------------
```

RELATED INFORMATION:

See display, and alter commands.

For single chip considerations, see 'dirty'.

See pioe for PIO expects.

faclist

PURPOSE: List scantable facilities known to ESP

SYNTAX: faclist [filter]

DESCRIPTION:

Use this command to get a list of facilities (created with the *cad* command) known to ESP. Use the list to search for partial fac names. Once a fac name has been located it can be selected and then used by double clicking on the selected name or by clicking the USE push button.

When a facility is used it is copied into the ESP command line at the point of the cursor.

The optional 'filter' parm is used to call up the list and specify a filter string at the same time. This is merely a convenience as the list has a filter text input area for you to use at any time.

USING THE FACLIST WINDOW

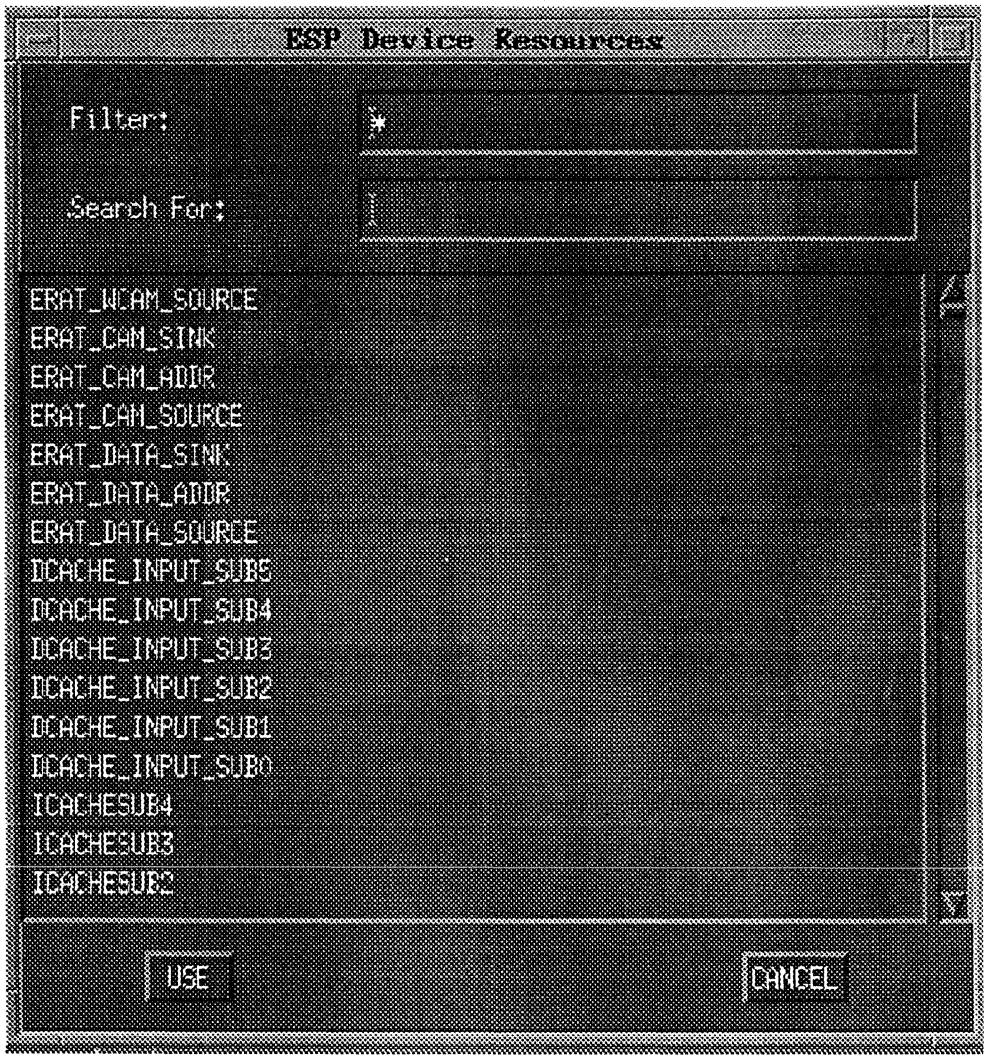
Once the faclist has been generated it is presented in a window. The window has a "Filter:" text entry area, a "Search For:" text entry area, a "USE" push button and a "CANCEL" push button.

The "Filter:" text entry area can be used to do a primary search to narrow down the selections presented, such as all facilities with XRAB in their name, or all facilities with .SPARE in their name. If ever you wish to see the entire list, type in a single asterisk and hit enter.

The "Search For:" text entry area is used to search through what ever is currently available (after the filter has worked). Search also finds facilities that have the specified string in them at any position.

Once a single facility has been found you can force it to be placed into the ESP command line (at the cursor) by double clicking the entry or by clicking on the "USE" push button.

You can leave the faclist window up during the entire ESP session and use it as desired. If, however, you wish to get rid of it, click on the "CANCEL" push button.



Example of window presented with faclist command.

EXAMPLES:

faclist - get a list of ALL facilities known to ESP

faclist SPARE - list of facilities with "SPARE" in their name

RELATED INFORMATION:

none

filefinder

PURPOSE: Show where a .x file or REXX file was found when executed

SYNTAX: filefinder [(on|off)]

DESCRIPTION: By default, filefinder is turned off in ESP. If you turn filefinder on, then subsequent execution of .x files or REXX files will cause the location of that file to be posted to the TTY window.

EXAMPLES: filefinder on - turn filefinder on

filefinder off - turn filefinder off

filefinder - get status, on or off

RELATED INFORMATION:

which

flush

write memBuf to Proc

PURPOSE: Forces buffered data from ESP to the MUT

SYNTAX: flush [(mpg|pg)] (chip_name | broadcast | bc | mem | l2) [SSname]

DESCRIPTION: If the argument **broadcast**, or **bc**, is passed then ESP does the following:

Any cached memory is flushed out to the machine under test (MUT)

Any cached scan strings for all chips are flushed to the MUT

If the argument passed is a CHIP NAME then only the chip's scan strings are flushed, and ONLY IF the chip's strings have been altered! If the optional 'mpg|pg' argument is used then the specified multiprocessor group or processor group will be flushed. By default, the current processor group is flushed.

NOTE: Flush always flushes 'real' scan strings (that need it) first. If there are any sub-strings left then they are flushed to the MUT using their individual scan commands.

If optional **SSname** is passed describing a scan string name, then only that scan string is flushed to the MUT (if it needs it).

NOTE: The SSname argument only works when a chip_name is specified.

EXAMPLES

If a chip has long, boundary, exmem, and counter scan strings, and if long contains both boundary and exmem (that is, boundary and exmem are sub-strings of the long scan string), then the following will happen:

If the long and boundary scan strings have been modified then only the long scan string will be flushed to the chip.

If only the boundary scan string has been modified, then only the boundary scan string will be flushed to the chip.

If the boundary and exmem scan strings, both sub-strings of the long scan string, have been modified then the boundary and exmem scan strings will be flushed to the chip in two seperate operations.

flush 604 - flush 604 scan strings out to the chip

flush mem - flushes any cached memory from ESP

flush l2 - flushes any cached L2 memory from ESP

flush 604 boundary - flushes only boundary data to 604 chip

RELATED INFORMATION:

See ESP command *dirty*, used in single chip testing also.

get

PURPOSE: Get a binary scan table file

SYNTAX: get filename

DESCRIPTION: ESP gets the scan table information in one of two ways. The first way is to read the chip scan string tables which are contained in the .x files (e.g. comboST.x, dcacheST.x, fpuST.x fxptST.x, icacheST.x, scuST.x). This must be done at least once after a new scan string table is received, but it is relatively slow. The second method is to read in a binary image of the information that was saved on a previous run.

Normally, when a new level of scan string information is received, or when a new level of the ESP program is created, the chip scan string definitions will be read in once, and then will be saved. Subsequent runs of ESP will get the binary scan table file.

You can't "get" more than one file without trouble. Also, you can't read ".x" files that define scan strings or arrays, and after that use the "get" command. "get" overwrites whatever was in ESP's data area. (i.e., "get" expects the ESP scan string definition area to be empty.)

RELATED INFORMATION:

save

gexpect

PURPOSE: Support testing for values from a Graphics Read operation

SYNTAX: gexpect address radix'value'

DESCRIPTION:

This command performs just as gread, but then compares the value read from the MUT with the 'value' specified. If the values fail to match then a message is printed specifying the address, the expected value, and the value actually returned.

RELATED INFORMATION:

gread and gwrite

gread

PURPOSE: Support Graphics Read operations

SYNTAX: gread address [radix]

DESCRIPTION:

This command implements a version of the architected instruction eciwx. The EAR register should be set appropriately before using this command, and the 'address' specified in this command is set on the 604 bus without translation.

A (4 byte) word is returned formatted in whatever radix is specified, or hexadecimal if no radix is specified.

Addresses may be specified in any ESP supported radix and, unlike the architected instruction, may specify addresses on byte boundaries, e.g. 0x101 or0x20000305. No matter what address is specified at least one double word bus access will occur.

The following table shows which four bytes are returned from a gread command based on the address specified.

The characters ABCD represent the four bytes displayed by the gread command.

User specified address low 3 bits	1st data bus xfer (8 byte lanes)	2nd data bus xfer (8 byte lanes)
000	ABCDxxxx	(no 2nd xfer)
001	xABCDxxx	(no 2nd xfer)
010	xxABCDxx	(no 2nd xfer)
011	xxxABCDx	(no 2nd xfer)
100	xxxxABCD	(no 2nd xfer)
101	xxxxxABC	Dxxxxxxx
110	xxxxxxAB	CDxxxxxx
111	xxxxxxxA	BCDxxxxx

RELATED INFORMATION:

gwrite and gexpect

gwrite

PURPOSE: Support Graphics Write operations

SYNTAX: gwrite address radix'value'

DESCRIPTION:

This command implements a version of the architected instruction ecowx. The EAR register should be set appropriately before using this command, and the 'address' specified in this command is set on the 604 bus without translation.

Addresses may be specified in any ESP supported radix and, unlike the architected instruction, may specify addresses on byte boundaries, e.g. 0x101 or0x20000305. No matter what address is specified at least one double word bus access will occur.

The following table shows which four bytes are written on a gwrite command based on the address specified.

The characters ABCD represent the four bytes modified by the gwrite command.

User specified address low 3 bits	1st data bus xfer (8 byte lanes)	2nd data bus xfer (8 byte lanes)
000	ABCDxxxx	(no 2nd xfer)
001	xABCDxxx	(no 2nd xfer)
010	xxABCDxx	(no 2nd xfer)
011	xxxABCDx	(no 2nd xfer)
100	xxxxABCD	(no 2nd xfer)
101	xxxxxABC	Dxxxxxxx
110	xxxxxxAB	CDxxxxxx
111	xxxxxxxA	BCDxxxxx

RELATED INFORMATION:

gread and gexpect

help

PURPOSE: Display help for ESP commands

SYNTAX: help [cmd1 cmd2 ... cmdN]

DESCRIPTION:

This command, without arguments, will list the names of all built-in ESP commands to the TTY window.

With arguments, each command listed will have its manual page presented on the screen, by default using GHOSTVIEW, a public domain PostScript viewing program.

The AIX environment variable VIEWER may be set (and exported) to specify the PostScript viewer of choice if GHOSTVIEW is unavailable or not desired.

EXAMPLES:

To use xpreview instead of GHOSTVIEW:

export VIEWER=/bin/xpreview

To put up seperate windows with manual pages for the alter, display and expect commands:

help alter display expect

To list all ESP commands to the TTY window:

help

RELATED INFORMATION:

The 'Help' button on the Menu Bar

hreset

PURPOSE: Toggle Or Set The HRESET Line

SYNTAX: hreset [(0|1)]

DESCRIPTION: iThis low level ESP command allows the hardware HRESET line to be toggled, set to zero, or set to one. This command is useful for testing the integrity of the ESP to MUT connection but is not usually used to reset the processor.

Using this command in any form will force ESP to consider all its buffered chip and memory information to be invalidated.

RELATED INFORMATION:

treset

ipl

in

PURPOSE: Perform a 1 byte memory mapped io read

SYNTAX: in address [radix]

DESCRIPTION:

This command is a convenience function that allows one byte to be read from memory mapped IO at address 0x80000000 and above.

The **address** specified is bitwise ored with 0x80000000 to produce the bus address. Eight bit **data** is then read in as a MMIO cycle and displayed in hexadecimal by default, or in whatever **radix** was specified.

Under the covers, ESP is producing an *mmior* command to implement the *in* command.

RELATED INFORMATION:

The ESP commands *in*, *in16*, *in32*, *out*, *out16*, and *out32* are all related.

in16

PURPOSE: Perform a 2 byte memory mapped io read

SYNTAX: in address [radix]

DESCRIPTION:

This command is a convenience function that allows two bytes to be read from memory mapped IO at address 0x80000000 and above.

The **address** specified is bitwise ored with 0x80000000 to produce the bus address. Sixteen bit **data** is then read in as a MMIO cycle and displayed in hexadecimal by default, or in whatever **radix** was specified.

Under the covers, ESP is producing an *mmior* command to implement the *in16* command.

RELATED INFORMATION:

The ESP commands *in*, *in16*, *in32*, *out*, *out16*, and *out32* are all related.

in32

PURPOSE: Perform a 4 byte memory mapped io read

SYNTAX: in address [radix]

DESCRIPTION:

This command is a convenience function that allows four bytes to be read from memory mapped IO at address 0x80000000 and above.

The **address** specified is bitwise ored with 0x80000000 to produce the bus address. Thirty-two bit **data** is then read in as a MMIO cycle and displayed in hexadecimal by default, or in whatever **radix** was specified.

Under the covers, ESP is producing an *mmior* command to implement the *in32* command.

RELATED INFORMATION:

The ESP commands *in*, *in16*, *in32*, *out*, *out16*, and *out32* are all related.

ioflag

PURPOSE: Tell ESP about run/stop state of an IO type of chip

SYNTAX: ioflag [(mpg | pg)] (chip_name | broadcast | bc) [(running | stopped)]

DESCRIPTION:

This command is used to inform ESP that an IO type of chip is either running or stopped. ESP uses this knowledge when accessing an IO chip's scan strings. If the IO chip is running, ESP will not scan the IO chip, even if a demand for data is made, like using the alter or display command or having a screen up with IO chip data present.

If optional argument (**running|stopped**) is omitted then *ioflag* will report the ioflag status of the chip or chips selected. The status will be either RUNNING or STOPPED.

Specifying (**running|stopped**) means that you are setting the IO chip ioflag status.

If you wish to set or view ioflag status of a particular chip then specify the **PG** and **chip_name** of the chip. To manipulate multiple chips at the same time specify an **MPG** and/or use the **broadcast** or **bc** chip pseudo names. The use of **broadcast** or **bc** means "all IO chips".

EXAMPLES:

ioflag **bc** Lists all equipped IO type chips flag status in current PG

ioflag pgc cobra Lists pgc cobra chip ioflag status only

ioflag pgc cobra **stopped**Sets cobra ioflag status to STOPPED in PG pgc only.

ioflag mpg1 **bc running**Sets ioflag status to RUNNING in all IO type chips in all PG that are in MPG mpg1.

RELATED INFORMATION:

IO type chips are created with the *cat* command.

ipl

PURPOSE: Toggle Or Set The SRESET Line

SYNTAX: ipl [(0|1)]

DESCRIPTION: This low level ESP command allows the hardware SRESET line to be toggled, set to zero, or set to one.

ESP performs no other action as a result of using this command. That is, buffered data and screen updates will NOT be performed because 'sreset' has been issued.

RELATED INFORMATION:

iplrun

treset

hreset

iplrun

PURPOSE: To force the machine under test (MUT) to run and take a soft reset interrupt.

SYNTAX: iplrun [(mpg|pg)] [time] [-v] [-bc]

DESCRIPTION:

NOTE: This is a "high level" command which means that any cached scan strings or memory will be flushed to the MUT before the MUT is set running. If the MUT stops then ESP screens will be updated.

If the *iplrun* command is issued without any arguments, then the default action is to toggle the reset interrupt hardware line, send a RESUME command (sequentially) to all the equipped chips in the current PG, print a message indicating the *iplrun* has been issued, and wait (forever) for the 604 status to indicate that it has stopped. Iplrun would then do a 'soft stop' procedure, print a message indicating a NORMAL stop, and return.

The **-v** option will keep *iplrun* from issuing the two messages just mentioned.

The **-bc** flag will force a single broadcast RESUME to be sent to all chips on the current port instead of sequentially sending RESUME to all chips in the specified PG or MPG.

If a **time** argument of zero is issued, then *iplrun* is issued to the MUT, as it would otherwise, but the *iplrun* command will not wait around for the MUT to stop. Instead, *iplrun* will quietly return to the caller. It is then up to the caller to look for the MUT to stop, and take appropriate action (i.e. issue *stop* command.)

If a **time** argument greater than zero is issued, then that number is the longest time (in seconds) that *iplrun* will wait for the MUT to stop. If the MUT stops before the time indicated then *iplrun* will return normally. If the MUT has not stopped by the number of seconds indicated, then *iplrun* will print an error message about timing out, and return to the caller. It is up to the caller to then take appropriate action (i.e. issue *stop* command.)

When *iplrun* is waiting for the MUT to stop, it looks at three status bits to determine the condition of the MUT: the hardware lines -CHECK-STOP and +POWERGOOD as well as the 604 chip status indicating run/stop status. If any of these status change then a stop condition is

understood. If checkstop or powergood indicate an error condition, then such conditions are reported, and iplrun returns to the caller without having performed a 'soft stop'.

If, while *iplrun* is waiting for the MUT to stop, the user types a program interrupt,[1] then *iplrun* will print a message indicating that it is aborting and returns to the caller. It is up to the caller to take appropriate action to control the MUT after such an abort (i.e. issue *stop* command.)

If the user wants the MUT to stop, a breakpoint must be set prior to issuing this command. If desired, the MUT can be run without a breakpoint, and after some amount of time a *stop* may be issued. (For example, if the program terminates in an infinite loop.) If the user just wants to let the MUT run without any breakpoints set, and if the MUT is currently stopped due to encountering a breakpoint, ESP will clear the current breakpoint, cause the MUT to execute one instruction, replace the breakpoint, and then tell the MUT to run. If the MUT returns to the same address, it will again stop.

Except for a very few early machines that do not have an IPLROS installed, this instruction will have the following affect. ESP will cause the MUT to vector to its cold or warm IPL entry point (depending on the previous code that was run.) The program at the cold IPL entry point will set up some required configuration registers and then will search for a boot program. This may be on a diskette or a hard file. The program will cause this boot program to be loaded and will branch to the boot program entry point.

Because the hardware breakpoint is not in memory, the breakpoint can be set prior to loading the program (and must be set prior to loading the program if the user desires to have a breakpoint set.) When the MUT detects the breakpoint address, the system will stop.

IMPLEMENTATION NOTE:

- All ESP cached memory is flushed to the MUT via the assigned PG.
- For each PG in the MPG specified, or the PG specified, or the CURRENT PG otherwise, all chips in each PG are flushed from ESP, ESP chip buffers are invalidated, and the RS_HIZ command is issued to the 604 chip.
- If an MPG was specified, then the enable for the MPG is executed
- The reset interrupt hardware line is toggled to all PG
- The RESUME command is sent sequentially to all equipped chips in the specified PG or MPG or a broadcast RESUME command is sent if -bc was specified
- If a time argument > 0 was specified then ESP waits up to that number of seconds for all PG 604 chips to become stopped.

---

1. See the section "Break Signal" for more information on the program interrupt.

EXAMPLES:

IPLRUN Start the MUT, then wait for it to stop.

IPLRUN pgb 0 Start the PG "pgb" MUT and return.

IPLRUN 12 Start the MUT, then wait up to 12 seconds for it to stop.

IPLRUN -V Start the MUT, then wait for it to stop. Do this without saying anything.

IPLRUN mpg3 5Start all PG in MPG "mpg3" and wait up to 5 seconds for all the PG to stop.

RETURN CODES:

0 All's well.

1 While waiting for MUT to stop, iplrun timed-out.

2 While waiting for MUT to stop, the MUT power failed.

3 The user typed a program interrupt.

4 While waiting for MUT to stop, the MUT checkstopped.

RELATED INFORMATION:

por, ipl, stop, run, bp

is

PURPOSE: Instruction step(s)

SYNTAX: is [(mpg|pg)] n [(soft|int)]

DESCRIPTION:

NOTE: The 604DD1.0 and 604DD1.1 do not have instruction step capability.

NOTE: This is a "high level" command which means that any cached scan strings or memory will be flushed to the MUT before the MUT is single stepped. When the MUT stops ESP screens will be updated.

This command sets a bit in the machine under test (MUT) which causes one instructions to be executed. ESP will repeat the required sequences to step as many times as requested. The first time this command is entered, the 'n' parameter is required. Subsequent commands can enter 'is' without the 'n' parameter and the previous value for 'n' will be used. If 'is' is entered the first time without the 'n' parameter, results are not specified.

A soft stop means that the processor is 'asked' to stop, and when it has stopped, ESP will take control. The processor is supposed to have quiesced the system it is in and then halted in such a way that it can be re-started.

The 'int' parm sets the 604 such that it will branch to interrupt vector 0x0d00 after the current instruction has been completed.

Because program execution stops prior to executing the instruction at the breakpoint address, a run command or an instruction step command issued after the processor has stopped at a breakpoint would cause the MUT to stop execution prior to executing the instruction at the breakpoint address. In other words, n instructions would be executed. When a run command or an instruction step command is executed, ESP will check to see if the breakpoint address is the same as the IAR value. If they are the same, ESP will clear the breakpoint for one instruction, and then will reset it to what it was. This allows run through a loop without continually setting and clearing a breakpoint, and instruction stepping from a breakpoint address without first clearing the breakpoint address.

NOTE: Breakpoint, branch trace, and instruction step all share (and set) the 604 stop mode.

NOTE: You cannot single step through a rfi or mtmsr instruction, or any other instruction that will modify the

MSR, since the instruction step mode is enabled by a bit in the MSR.

RELATED INFORMATION:

bp - break point

sais - screen add instruction step command

bt - branch trace

is - instruction step

isd

PURPOSE: Start the ESP Interactive Screen Designer

SYNTAX: isd

DESCRIPTION:

The Interactive Screen Designer is a windows based tool developed to allow users to create custom ESP screens visually.

Although ISD is a stand-alone program, it is invoked from ESP as if it were a built-in ESP command. ESP responds by creating a temporary file containing names and sizes of all devices created with the ESP commands *cad* and *caa*[1]. ESP then invokes ISD passing the name of the file as a command line argument to ISD. With this, ESP is finished with the *isd* command and is ready to process other ESP commands.

ISD, once running, is its own process and is in no way tied to the running ESP. ESP can, in fact, be terminated without effect on ISD.

Multiple ISD programs may be started from ESP and be running at the same time, if desired. Each ISD program started from ESP is given its own copy of a temporary file with names and sizes of known devices.

ISD creates/modifies ESP screens. ESP screens are simply ESP command files filled with ESP commands that, taken together, create an ESP screen. ESP command files are ASCII-text files, named with a .x filename extension, and filled with ESP commands. The ISD program can read and write the ESP command files used to create ESP screens.

For information on how to use the ISD tool see the ***ISD User's Reference Manual*** in /afs/awd/public/esp/userdoc/ISDmanual.ps. For information about ESP screens, see the ***ESP Training Manual*** in /afs/awd/public/esp/userdoc/espTrainingManual.ps.

RELATED INFORMATION:

1. Only device and array names of equipped chips are generated.

jtag

PURPOSE: Send JTAG data and protocol

SYNTAX: jtag TDIvalue TMSvalue

DESCRIPTION: This command supports the ability to send any JTAG data and TMS pattern on the current JTAG port.

TDIvalue and **TMSvalue** can be expressed as binary or hex. ESP determines the number of TCK clocks to send from the number of bits expressed in the data.

For each bit position in the data (**TDIvalue** and **TMSvalue**), ESP sets TDI to the **TDIvalue** bit, TMS to the **TMSvalue** bit, and then issues a TCK. This process is repeated for the number of bits in the data.

The first bit sent to the JTAG port is the right-hand side of the **TDIvalue** or **TMSvalue**.

NOTE: Data expressed should not exceed roughly 17,500 bits.

This command is not available for COP protocol chips.

EXAMPLE:

To send 32 bits into the current JTAG port where the data is x'feedbeef' and the TMS line is to be held high:

jtag x'feedbeef' x'ffffffff'

RELATED INFORMATION:

The *cop* and *copcmd* commands.

layout

PURPOSE: locate or load a screen, move it to xy and set its origin and base addresses

SYNTAX: layout screen x y org_adr base_adr [PG=pg] [CHIP=chip]

DESCRIPTION: This loads the named 'screen' from file and then moves the screen to the 'x','y' values specified. Layout also sets the screens ORIGIN address to 'org_adr' and the screens BASE address to 'base_adr'.

Optional argument 'PG=pg' allows you to set the Processor Group variable, PG, before an attempt is made to load the screen from disk. After the attempt is made, PG is reset to its old value.

Optional argument 'CHIP=chip' allows you to set the Chip variable, CHIP, before an attempt is made to load the screen from disk. After the attempt is made, CHIP is reset to its old value.

This command might be used in a ESP Script file with other layout commands in order to force a particular configuration of screens.

ESP uses this capability by storing layout commands in the users HOME directory in a file called .esplayout.x whenever the Option menu button "Save Layout" is pressed.

Normally the 'screen' parameter would be the nickname of a screen. However, a special screen name, _ESP_CONSOLE, is understood by layout to mean the ESP main window.

RETURNS:

EXAMPLES: The following commands would place the ESP Console and cause two screens to come up:

```
layout _ESP_CONSOLE 24 511 0 0
layout mpmrs 638 426 0 0 PG=pgc
layout mpmrs 638 232 0 0 PG=pgb
```

RELATED INFORMATION:

See ESP commands "saveLayout" and "move".

list

PURPOSE: List details about internal ESP objects

SYNTAX: list [n [options]]

list - displays a list of the command options

List will allow visibility of the following:

0 CHIP_REC [chip_name]
1 CHIPTYP_REC [chip_type]
2 ARRAY_REC [chip_type]
3 Array DEVICE_REC's [chip_type]
4 Device DEVICE_REC's [chip_type [device_name]]
5 MPGROUP's
6 PGROUP's
7 Array Names and Sizes

DESCRIPTION: Not recommended for everyday use. This command is more of a debug tool than a user resource.

EXAMPLE:

list 0 icu -- will cause the ESP program to display a list of the structure definition for the ICU chip.

RELATED INFORMATION:

listall

listall

PURPOSE: List all defined devices.

SYNTAX: listall

listall chip_type

DESCRIPTION: This command lists all devices and arrays to a list window. The list may be scrolled to view the contents. This command is more of a debug tool than a user resource.

RELATED INFORMATION:

list

load

PURPOSE: Download a file from ESP Host into MUT memory.

SYNTAX: load [[-d] [-r] [-vN] [-z] filename]

FLAGS

-r Do not report on down load progress. Default for load is to display down load progress.

-vN Verbatim load the entire file into memory at address N. The address, N, can be specified in hex, binary, octal, or decimal. (eg. -vx'10025' or -v64 or -vo'377')

-d Don't Zero the common area after file load. (The common area is also known as the uninitialized data area.)

-z Zero out the common area of a previously loaded file (to re-run it)

DESCRIPTION: The purpose of this command is to load a file residing on the ESP Host machine into MUT memory, with or without including the header, starting at the load address specified in the header, or optionally, at some other address.

If no load options are used then the file mentioned will be loaded into MUT memory at whatever address that is found in the files header. TOC64, TOC96 and XCOFF files are understood by ESP. Separate text, data, and uninitialized data areas specified by the XCOFF file format are supported by ESP. TOC files are expected to have contiguous data areas.

If the -vN option is specified then the load command will ignore the files header and just get the entire files size using the AIX stat command. The file will then be read by ESP and loaded into MUT memory at address 'N' as specified by the user. Using this option means that it does not matter what the file type is: ascii text, TOCxx, XCOFF, binary data or whatever. The file is simply loaded, verbatim, into MUT memory at address 'N'.

We have tried to make changes to the load command options backwards compatible with old versions of ESP. Older versions of load supported the -t, -6, and -a options which are now not used. These old option flags will be ignored by the newer ESP so that users will not have to change previously written programs that include them.

The source file name can be any valid AIX file name. It can be located in any directory as specified in ESP AVPBIN environment variable.

Here are some rules about the source file presented to load:

- The load address (from HEADER or from -v option) must be on a word boundary. (no remainder when address anded with 0x3)
- No file name convention is enforced on the source file.

EXAMPLES: load avp1

load -r avp1

load -vx'200f' avp1

load -z avp1

RELATED INFORMATION:

dump

load_cntr

PURPOSE: Make loading the run-for-N counter easier

SYNTAX: load_cntr chip_name count

DESCRIPTION:

This is a low level command used to load the run-for-N counter. No other operation is performed.

NOTE: The scan string loaded by ESP is assumed to be known as the "counter" scan string, and is also assumed to be 32 bits wide.

NOTE: The least significant counter bit will always be set to one by this routine.

Load_cntr will take a count, 0 to 2^32, and perform the necessary conversion for a incrementing counter (counts= 0x7fffffff - requested_count). Load_cntr will then send the necessary COP commands to send out the "counter" scan string.

If chip_name is "604" then the "counter" device is expected to exist and be 32 bits wide (to accept the count).

If chip_name is "mamba" or "x5" then the "tap" device is expected to exist and be 10 bits wide.

Any other chip_name will not have its counter loaded.

EXAMPLES: Load the serial counter to do 50 cycles.

load_cntr 604 50

RELATED INFORMATION:

See the 'cop' command to issue the RUNN command.

Refer to the 'CS' command for system wide run-for-n.

log

PURPOSE: Start logging messages to a file

SYNTAX:

log	closes current log file (if any)
log filename	open log file (truncate, close current as needed)
log -append filename	open log file (append, close current as needed)
log -clear	empty current log file (if any)
log -print	print current log file (if any)
log -edit	edit current log file (if any)

DESCRIPTION: A log file captures any text sent to the TTY window (even if the TTY window is turned off). Once a log file is opened it can be closed with the simple 'log' command.

The log file can be opened so that it first erases any existing file of the same name, or the -append option can be used to append text to an existing log file.

The -clear option clears the currently open log file. Logging continues.

The -print option prints the currently open log file to the AIX systems default printer. Logging continues unchanged.

The -edit option edits the currently open log file. A 'snapshot' copy of the current log is made and your editor is used (using the AIX environment variable EDITOR) to edit that copy. Once editing is over the copy of the log file is deleted. Normal logging remains unchanged during and after the edit session.

RELATED INFORMATION:

none

ls

PURPOSE: Display the contents of a directory.

SYNTAX: ls

DESCRIPTION: See AIX Commands Reference for details. This ESP command is passed through to the AIX shell to be executed. It is included in ESP for user convenience.

The files found are listed on the TTY window.

RELATED INFORMATION:

xlist

which

mal

PURPOSE: Adds a line to a menu.

SYNTAX: mal name label command

DESCRIPTION: See "Menus" in Part 1 for details.

Custom menus can be created by the user and accessed from the ESP command line or from the ESP console menu bar "Custom Menus" button.

The mal command supports creation of a single line on a menu. The menu label is created with the 'label' argument, and the single ESP command that is to be executed when the menu line is selected is created with the 'command' argument.

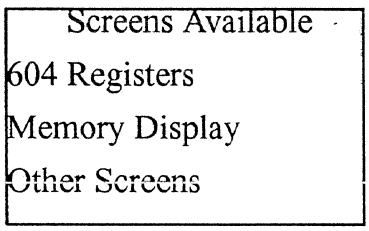
Since there can be many named menus, the 'name' argument must specify which menu a line is being added to.

Chaining from one menu to another is supported by allowing the ESP command executed by selection of a menu item to call another menu.

EXAMPLES:

To create a menu and add lines to it:

```
mat screens "Screens Available"
mal screens "604 Registers" "screen 604reg"
mal screens "Memory Display" "screen mema"
mal screens "Other Screens" "menu other"
```



RELATED INFORMATION:

mat

menu

mat

PURPOSE: Creates a new menu.

SYNTAX: mat nickname title

DESCRIPTION: See "Menus" in Part 1 for details.

Custom menus can be created by the user and accessed from the ESP command line or from the ESP console menu bar "Custom Menus" button.

The mat command is the first step in creating a custom menu since it both names the menu with 'nickname' and gives it a title, 'title', when displayed.

Either the 'nickname' or 'title' can be used to reference the menu with either the 'menu' command or the 'mat' command.

RELATED INFORMATION:

mal

menu

meminit

PURPOSE: read in information required for accessing MUT external memory

SYNTAX: meminit [pg] [tc60x]

DESCRIPTION:

'meminit' is an ESP command that forces ESP to initialize its memory handling sub-routines. Usually, 'meminit' is called only once when ESP is first started.

This command must be executed after the scan string definitions and array definitions have been read in.

Because ESP may be used in situations where partial scantables or no scantables are in place, it is sometimes useful to *not* perform ESP memory access initialization.

Typically, meminit is placed in the profile.x file right after the binary scantables have been read.

Optional argument 'pg' is used to support ESP memory access through different Processor Groups in a Multiple Processor environment. Memory access is always made from the PG specified in 'meminit', even if the Current PG is another PG.

Optional argument 'tc60x' supports the Motorola Test Card used to bring up the 604 chip. If 'meminit tc60x' is specified, then ESP will access memory via the Motorola Test Card device driver instead of through the scan strings.

RELATED INFORMATION:

none

memread

PURPOSE: Generate a list of memory data.

SYNTAX: memread startAddress numWords [radix]

DESCRIPTION: This command is used to read a list of (4 byte) words from memory, 'numWords' long, beginning at startAddress. The data can be displayed in any of the usually supported radices.

memread is intended to support the ESP sockets programming interface and the programming interface. Normal user interactive memory display is better performed using the display command.

RELATED INFORMATION:

memwrite

display

alter

memwrite

PURPOSE: To write a list of words to memory.

SYNTAX: memwrite startAddress numWords\n data\n data\n ... data\n

DESCRIPTION: This command is used to write a list of (4 byte) words to memory, 'numWords' long, beginning at 'startAddress'. The data can be written in any of the usually supported radices, and unlike the alter command, memwrite flushes memory automatically instead of waiting for it to be done manually by the user, or until the ESP's cache buffer is filled.

memwrite is intended to support the ESP sockets programming interface and programming interface. Normal user interactive memory modification is better performed using the alter command.

RELATED INFORMATION:

memread

alter

display

menu

PURPOSE: Displays a particular Custom Menu if one is specified, or the first Custom Menu if no menu name is specified.

SYNTAX: menu [name]

DESCRIPTION: Custom Menus may be built by the user using the mat and mal commands. The menu command "pops" one up for use. See "Menus" in Part 1 for details.

Custom menus can be created by the user, using the mat and mal commands, and may then be accessed from the ESP command line.

RELATED INFORMATION:

mat

mal

mmior

PURPOSE: Support reading from memory mapped devices.

SYNTAX: mmior startAddress numBytes radix

DESCRIPTION: Devices other than memory may be connected to the memory bus of the processor. This command supports reading values from these devices in multiples of byte wide accesses.

From 1 to 8 bytes at a time may be read from a memory mapped device starting from any address. The output will be formatted in whatever 'radix' is specified. If no radix is specified then the output will be presented in hexadecimal. Radix specification is expected to be a single character, upper or lower case, in the set a=ascii, b=binary, d=decimal, e=ebcdic, f=float, i=instruction, o=octal, x=hex. 'startAddress' and 'numBytes' may be expressed in binary, decimal, octal, or hex.

RELATED INFORMATION:

mmiow

mmiow

PURPOSE: Support writing to memory mapped devices.

SYNTAX: mmiow startAddress numBytes data

DESCRIPTION: Devices other than memory may be connected to the memory bus of the processor. This command supports writing values to these devices in byte multiples.

From 1 to 8 bytes at a time may be written to a memory mapped device starting at any address.

'startAddress' and 'numBytes' may be expressed in binary, decimal, octal, or hex. The 'data' may be expressed in any ESP supported radix.

RELATED INFORMATION:

mmior

move

PURPOSE: To reposition screens on the display

SYNTAX: move screen_name y x

DESCRIPTION: By issuing this command with the name of a specific screen you want to move and a set of x and y coordinates, you can rearrange the layout of the screens on your display.

Coordinates x and y are expected to be decimal integers greater than or equal to zero. X is the horizontal coordinate and Y is the vertical coordinate.

Normally the 'screen_name' parameter would be the nickname of a screen. However, a special screen name, _ESP_CONSOLE, is understood by move to mean the ESP main window.

EXAMPLE: move mema 0 0

This would move the mema screen to the top left corner of the display area

move _ESP_CONSOLE 500 100

This would move the ESP main window.

RELATED INFORMATION:

layout

number

PURPOSE: Control displaying of message numbers.

SYNTAX: number (on|off) [all rexx socket pi tty log]

DESCRIPTION: Each message that ESP produces has a message I.D. number that goes with it. This command allows you to select whether the message I.D. number will be displayed with a message or not. Selection is based on where the message is being sent. If the rexx option is chosen to be turned on, only those messages that the ESP sends to rexx will contain their associated I.D. number. Similarly, the socket option selects messages being sent to a socket, the pi option selects messages being sent to the programming interface, the tty option selects messages being displayed in the tty window, and the log option selects messages being sent to a log file.

RELATED INFORMATION:

none

ocs

PURPOSE: Manipulate the OCSOVERIDE_ line

SYNTAX: ocs (on|off)

DESCRIPTION: The 604 processor is a JTAG processor and has no ESP enable line or an On-Chip-Sequencer to manipulate. This means that the *ocs* command is not useful for the 604 processor.

However, some systems incorporating the 604 processor might find the OCS line useful.

NOTE: Currently ESP for 604 DD2 and DD3 has the OCS command activated. The OCS pin has been assigned as pin 13 of the ESP connector.

The *ocs* command might be used to activate the MUT line that tells the MUT that ESP is connected.

ocs **on** forces ESP line OCSOVERIDE_ inactive, or high, while *ocs* **off** forces the line active, or low.

The default value for the OCSOVERIDE_ line is inactive, high, set by ESP when it is first started.

RELATED INFORMATION:

none

out

PURPOSE: Perform a 1 byte memory mapped io write

SYNTAX: out address data

DESCRIPTION:

This command is a convenience function that allows a single byte to be written to memory mapped IO at address 0x80000000 and above.

The **address** specified is bitwise ored with 0x80000000 to produce the bus address. Single byte **data** is then written as a MMIO cycle.

Under the covers, ESP is producing an *mmiow* command to implement the *out* command.

RELATED INFORMATION:

The ESP commands *in*, *in16*, *in32*, *out*, *out16*, and *out32* are all related.

out16

PURPOSE: Perform a 2 byte memory mapped io write

SYNTAX: out address data

DESCRIPTION:

This command is a convenience function that allows two bytes to be written to memory mapped IO at address 0x80000000 and above.

The **address** specified is bitwise ored with 0x80000000 to produce the bus address. Sixteen bit **data** is then written as a MMIO cycle.

Under the covers, ESP is producing an *mmiow* command to implement the *out16* command.

RELATED INFORMATION:

The ESP commands *in*, *in16*, *in32*, *out*, *out16*, and *out32* are all related.

out32

PURPOSE: Perform a 4 byte memory mapped io write

SYNTAX: out address data

DESCRIPTION:

This command is a convenience function that allows four bytes to be written to memory mapped IO at address 0x80000000 and above.

The **address** specified is bitwise ored with 0x80000000 to produce the bus address. Thirty-two bit **data** is then written as a MMIO cycle.

Under the covers, ESP is producing an *mmiow* command to implement the *out32* command.

RELATED INFORMATION:

The ESP commands *in*, *in16*, *in32*, *out*, *out16*, and *out32* are all related.

pages

PURPOSE: Lists the screens that are up or in memory.

SYNTAX: pages

DESCRIPTION: This command generates a list of pages that are in memory. A page is put in memory every time a screen is called up.

This command is most useful from a program that needs to know what screens are currently up on the console.

RELATED INFORMATION:

none

pause

PURPOSE: Halt execution momentarily and display a message

SYNTAX: pause "message"

DESCRIPTION:

The pause command may be used to temporarily halt ESP execution while at the same time displaying a message for the user. The pause command is meant to support interactive use of ESP between a command file or REXX program and the user.

If ESP is running in the interactive mode with X windows, then a window will be presented with the "message" indicated. ESP will stop executing instructions until the user has pressed the OK button.

If ESP is running in the interactive mode with out X windows, then the message will be printed on the standard output, and the user must press some key to restart ESP.

RELATED INFORMATION:

none

pg

PURPOSE: Select a Processor Group as the default or current Processor Group

SYNTAX: pg [PGname]

DESCRIPTION: Specifying pg without other arguements will cause ESP to list what the current Processor Group is set to.

If 'PGname' is specified then ESP will remember that Processor Group name, and any ESP commands not specifically stating the PG to use will default to this PGname.

In a single processor environment this command does not need to be used. ESP defaults to a "DEFAULTMUT" Processor Group in which all chips are members. (MUT means Machine Under Test, and is reported by pg only if no Processor Groups have been created by the user.)

RELATED INFORMATION:

none

pinread

PURPOSE: Read the value of a pin on an ESP buffer card

SYNTAX: pinread bufnum pinName

DESCRIPTION:.

With this command the value of a (named) pin on a particular ESP buffer card can be obtained. If the pin is an ESP output, then the value that ESP is currently driving will be read.

ESP can support up to sixteen ESP buffer cards, each with a 16 pin connector that plugs into a MUT. **bufnum** allows specification of any one of the sixteen buffers.

There are sixteen pins on the ESP connector, numbered 1 through 16. Some or all of the pins might be named from the *cbuf* command. To read the value of a pin, you must specify **.pinName** with a mnemonic known to ESP. The mnemonics known to ESP are listed in Table 4

Table 4: Pin Names Known To ESP

Mnemonic	Meaning
trst	JTAG reset
tms	JTAG mode select
tck	JTAG clock (ESP output)
tdi	JTAG serial data in (ESP output)
tdo	JTAG serial data out (ESP input)
ctl	COP control
clk	COP clock
si	COP serial in
so	COP serial out
cs	checkstop
rs	run/stop
pg	power good
hr	hard reset
sr	soft reset
ocs	ocs override

Table 4: Pin Names Known To ESP

Mnemonic	Meaning
auxout1	Auxiliary output #1
auxout2	Auxiliary output #2
auxout3	Auxiliary output #3
auxout4	Auxiliary output #4
auxin1	Auxiliary input #1
auxin2	Auxiliary input #2
auxin3	Auxiliary input #3
auxin4	Auxiliary input #4

RELATED INFORMATION:

pinwrite, cbuf, readcon

pinwrite

PURPOSE: Set the value of a pin on an ESP buffer card

SYNTAX: pinwrite bufnum pinName pinValue

DESCRIPTION:.

With this command the value of a (named) pin on a particular ESP buffer card can be set[1].

ESP can support up to sixteen ESP buffer cards, each with a 16 pin connector that plugs into a MUT. **bufnum** allows specification of any one of the sixteen buffers.

There are sixteen pins on the ESP connector, numbered 1 through 16. Some or all of the pins might be named from the *cbuf* command. To set the value of a pin, you must specify **pinName** with a mnemonic known to ESP. The mnemonics known to ESP are listed in Table. 4.

1. Read on. Some pins may NOT be set.

Table 5: Pin Names Known To ESP

Mnemonic	Meaning
trst	JTAG reset
tms	JTAG mode select
tck	JTAG clock (ESP output)
tdi	JTAG serial data in (ESP output)
tdo	JTAG serial data out (ESP input)
ctl	COP control
clk	COP clock
si	COP serial in
so	COP serial out
cs	checkstop
rs	run/stop
pg	power good
hr	hard reset
sr	soft reset
ocs	ocs override
auxout1	Auxiliary output #1
auxout2	Auxiliary output #2
auxout3	Auxiliary output #3
auxout4	Auxiliary output #4
auxin1	Auxiliary input #1
auxin2	Auxiliary input #2
auxin3	Auxiliary input #3
auxin4	Auxiliary input #4

RELATED INFORMATION:

pinread, cbuf, readcon

pioe

PURPOSE: Extended Transfer Protocol or Programmable I/O -- EXPECT

SYNTAX: pioe address value

DESCRIPTION: Devices other than memory may be connected to the memory bus of the processor. This command supports reading values from direct store (I/O) storage space and testing that the value read matches a value expected.

This ESP command only supports a 4 byte single beat transfer.

'address' and 'value' may be expressed in binary, decimal, octal, or hex.

RELATED INFORMATION:

piow and pior

pior

PURPOSE: Extended Transfer Protocol or Programmable I/O -- READ

SYNTAX: pior address [radix]

DESCRIPTION: Devices other than memory may be connected to the memory bus of the processor. This command supports reading values from direct store (I/O) storage space.

This ESP command only supports a 4 byte single beat transfer.

The output word will be formatted in whatever 'radix' is specified. If no radix is specified then the output will be presented in hexadecimal. Radix specification is expected to be a single character, upper or lower case, in the set a=ascii, b=binary, d=decimal, e=ebcdic, f=float, i=instruction, o=octal, x=hex.

'address' may be expressed in binary, decimal, octal, or hex.

RELATED INFORMATION:

piow and pioe

piow

PURPOSE: Extended Transfer Protocol or Programmable I/O -- WRITE

SYNTAX: piow address data

DESCRIPTION: Devices other than memory may be connected to the memory bus of the processor. This command supports writing values to direct store (I/O) storage space.

This ESP command only supports a 4 byte single beat transfer.

'address' and 'data' may be expressed in binary, decimal, octal, or hex.

RELATED INFORMATION:

pior and pioe

por

PURPOSE: Perform the simplest MUT reset possible.

SYNTAX: por [(mpg|pg)]

DESCRIPTION: The por command is intended to initialize the processor. The JTAG TRST line is toggled to reset the TAP, then the hardware reset line is asserted (low) and held while a FFRZ command is issued to the chip. Finally the hardware reset line is released (high). After these actions are performed, if a screen is up, it will be updated.

On ESP for 604 the 'por' command performs exactly the same function as the 'reset' command. Both command names have been retained for backwards compatibility.

RELATED INFORMATION:

reset

prs

PURPOSE: Prints a screen as it appears on the display.

SYNTAX: prs screen [filename]

DESCRIPTION: The prs screen command with no file argument uses the AIX program 'qprt' to print the ESP screen on the default printer for the system. (The actual AIX command generated is: '(cd /tmp;qprt -r -T "title" file)'.)

If a filename is specified, the prs command copies the ESP screen to the named file. If the file already exists it will be overwritten.

In either case, the ESP screen is rendered into ASCII text.

The 'screen' argument can merely specify the screen name, like "mema" or "604reg". However, if more than one of the same screen is up at the same time, then the screen sequence number must also be specified, like "mema(0)" or "604reg(1)". The full screen name, including sequence number, is in the title of the screen.

RELATED INFORMATION:

none

pwd

PURPOSE: Display ESP current working directory

SYNTAX: pwd

DESCRIPTION:

This command allows the user to determine ESP's notion of the current working directory. The current working directory will be printed in the ESP TTY window.

RELATED INFORMATION:

ls, cd

quit

PURPOSE: Terminate the ESP program.

SYNTAX: quit

DESCRIPTION: The quit command, executed from a ESP script file, from REXX, or from the ESP command line will cause ESP to stop execution.

Use the 'exit' command to terminate a ESP script file early.

RELATED INFORMATION:

exit

readcon

PURPOSE: Read an ESP buffer card connector

SYNTAX: readcon [bufnum]

DESCRIPTION:.

The *readcon* command provides a means to see what values are on the pins of an ESP connector that is plugged into a MUT. ESP can support up to sixteen ESP buffer cards, each with a 16 pin connector that plugs into a MUT. **bufnum** allows specification of any one of the sixteen buffers. If **bufnum** is not specified, the first buffer card will be accessed.

There are sixteen pins on the ESP connector, numbered 1 through 16. Some or all of the pins might be named from the *cbuf* command.

The output of this command is 17 lines long with a width of 12 characters (in case you wish to put the output on a screen using the *saespco* command.) The first line reports the buffer being accessed, and the next sixteen lines report the names and values of the connector pins.

RELATED INFORMATION:

pinread, cbuf

reducedpinmode

PURPOSE: Match the ESP to the hardware configuration

SYNTAX: reducedpinmode [(on|off)]

DESCRIPTION:

This command is used to force ESP to match the 604 hardware configuration. When the 604 was hard-reset, pin qack_ was sampled by the 604 and the pin state forced the 604 to be in the reduced-pin-mode or not. If the qack_ line was high, then the reduced-pin-mode was turned on in the 604. ESP must then have its reduced-pin-mode turned on using this command.

Typing *reducedpinmode* without arguments will cause ESP to report the mode it is currently in. Typing *reducedpinmode* **on** will turn on the ESP reduced-pin-mode (but not affect the 604 chip) and typing *reducedpinmode* **off** will turn off the ESP reduced-pin-mode (but not affect the 604 chip).

NOTE: If it were possible ESP would match the 604 reduced-pin-mode automatically.

The effect of the reduced-pin-mode on ESP is to have ESP expect only 32 bit operations on the 604 bus during memory access or memory mapped IO.

RELATED INFORMATION:

ESP commands *drtrymode* and *32bitmode*.

refresh

PURPOSE: Update one, many, or all currently active screens

SYNTAX: refresh [screenList]

DESCRIPTION:

This command can be used to force updates of visible screens even if the screen autoupdate is turned off.

A **screenList** is a list of one or more screen names seperated by spaces or tabs.

EXAMPLES:

To update all screens:

refresh

To update only screens 604reg and mema:

refresh 604reg mema

If two 604 reg screens are up, 604reg(0) and 604reg(1), and you want to refresh only the 604reg(1) screen:

refresh 604reg(1)

RELATED INFORMATION:

none

reset

PURPOSE: Goes through POR sequence

SYNTAX: reset

DESCRIPTION: .The reset command is intended to initialize the processor. The JTAG TRST line is toggled to reset the TAP, then the hardware reset line is asserted (low) and held while a FFRZ command is issued to the chip. Finally the hardware reset line is released (high). After these actions are performed, if a screen is up, it will be updated.

On ESP for 604 the 'por' command performs exactly the same function as the 'reset' command. Both command names have been retained for backwards compatibility.

RELATED INFORMATION:

por

resetint

PURPOSE: Will set the reset interrupt line to state indicated if the icache is not "equip"ed.

SYNTAX: resetint [(mpg | pg)] 1 - set reset interrupt line to a 1

resetint [(mpg | pg)] 0 - set reset interrupt line to a 0

DESCRIPTION: Normally, the icache controls the +RUN/-BREAKPOINT line. This line must be low for ESP to attempt to scan any chips, and must be high for any of the chips to run any functions, including self test and run for N cycles. When a chip or group of chips is being tested and there is no icache present, connecting the reset interrupt line (an input to the icache only) to the +RUN/-BREAKPOINT line (an output from the icache only) allows ESP to control the +RUN/-BREAKPOINT line directly. This is required for single chip testing and testing any chips without an icache present.

RELATED INFORMATION:

none

run

PURPOSE: To run the machine under test (MUT) until it stops, and then do a 'soft stop' procedure.

SYNTAX: run [(mpg | pg)] [time] [-v] [-bc]

DESCRIPTION:

NOTE: This is a "high level" command which means that any cached scan strings or memory will be flushed to the MUT before the MUT is set running. If the MUT stops then ESP screens will be updated.

If the run command is issued without any arguments, then the default reaction is to run the MUT, print a message indicating the run has been issued, and wait (forever) for the MUT status to indicate it is stopped. Run would then do a 'soft stop' procedure, print a message indicating a NORMAL stop, and return

The 'dash v' option (-v) would keep run from issuing the two messages just mentioned.

The **-bc** flag will force a single broadcast RESUME to be sent to all chips on the current port instead of sequentially sending RESUME to all chips in the specified PG or MPG.

If a time argument of zero is issued, then the run is issued to the MUT, as it would otherwise, but the run command will not wait around for the MUT to stop. Instead, run will quietly return to the caller. It is then up to the caller to look for the MUT to stop, and take appropriate action (ie. issue 'stop' command.)

If a time argument greater than zero is issued, then that number is the longest time (in seconds) that run will wait for the MUT to stop. If the MUT stops before the time indicated then run will return normally. If the MUT has not stopped by the number of seconds indicated, then run will print an error message about timing out, and return to the caller. It is up to the caller to then take appropriate action (ie. issue 'stop' command.)

When run is waiting for the MUT to stop, it looks at three status bits to determine the condition of the MUT. -CHECKSTOP, RUN/BREAKPOINT, and +POWERGOOD status are monitored. If any of these status change then a stop condition is understood. If checkstop or powergood indicate an error condition, then such conditions are reported, and run returns to the caller without having performed a 'soft stop'.

If, while run is waiting for the MUT to stop, the user types a program interrupt,[1] then run will print a message indicating that it is aborting and then return to the caller. It is up to the caller to take appropriate action to control the MUT after such an abort (i.e. issue 'stop' command.)

If the user wants the MUT to stop, a breakpoint must be set prior to issuing this command. The user can run without a breakpoint, and after some amount of time issue a 'stop' command. If the user just wants to let the MUT run without any breakpoints set, and if the MUT is currently stopped due to encountering a breakpoint, that breakpoint must be cleared prior to issuing the run command. If the breakpoint is not cleared, the MUT will test for a breakpoint prior to executing the first instruction and stop without ever having started.

EXAMPLES:

RUN Start the MUT, then wait for it to stop.

RUN 0 Start the MUT and return.

RUN 12 Start the MUT, then wait up to 12 seconds for it to stop.

RUN -v Start the MUT, then wait for it to stop. Do this without saying anything.

RETURN CODES:

0 All's well.

1 While waiting for MUT to stop, run timed-out.

2 While waiting for MUT to stop, the MUT power failed.

3 The user typed a program interrupt.(4)

4 While waiting for MUT to stop, the MUT checkstopped.

RELATED INFORMATION:

por

stop

iplrun

1. See the section "Break Signal" for more information on the program interrupt.

saa

PURPOSE: Prepare a screen to display array data with page up/down

SYNTAX: saa (title|nickname) chip array increment

DESCRIPTION:.

This command prepares a screen to display one or more arrays (of the same height) in a paged format. The base and origin address of the screen is set to zero, and the height of the array(s) to be displayed is recorded (so that PageUp and PageDown will know where the end of the array(s) is.) The number of hex digits required to hold the maximum address of the array(s) is calculated and recorded.

The 'increment' value is recorded for use with PageUp and PageDown. When the PageUp button is pressed, the screen page origin will be decremented by 'increment' value, and if the PageDown button is pressed, the screen page origin will be incremented by 'increment'.

This command should only be performed once per screen as it is a kind of "initialization" for the screen to prepare it for array display.

RELATED INFORMATION:

sao, sav, sad

sabreak

PURPOSE: Screen Add Breakpoint Handling Field

SYNTAX: sabreak screen_name row col format [pg]

DESCRIPTION: This will cause a data field to be added to a screen which contains the current breakpoint. The breakpoint can be altered by altering this field. This is used as an example in the mrs.x screen.

The screen field will have the following characteristics:

aaaaaaaa bbbbbbbbbbb

where: a = 8 hex digits that are the breakpoint address

b = either "soft" or "hard" or "int"

If optional argument 'pg' is specified then this field will reflect the breakpoint settings for only that Processor Group.

RELATED INFORMATION:

saco

PURPOSE: Screen Add Captured Output

SYNTAX: saco screen_name row col width "AIX command" [bg=] [fg=]

DESCRIPTION: This ESP command will cause a label to be added to a screen which will contain a single line output from whatever 'AIX command' that was executed. Each time the screen is refreshed, the 'AIX command' will be executed with its output captured and presented on the screen in the label area.

The screen label cannot be edited by the user.

RELATED INFORMATION:

none

sacs

PURPOSE: Screen add cycle step

SYNTAX: sacs screen_name row col format [pg]

DESCRIPTION: This will cause a data field to be added to a screen which contains the current number of cycle steps (machine cycles) to be executed. This value can be altered by altering this field. This is used in the mrs.x screen.

RELATED INFORMATION:

none

sad

PURPOSE: Adds an editable data field to a screen.

SYNTAX: sad name row col radix [pg] chip device(dx:dy) or
sad name row col radix [pg] chip array[offset](dx:dy)or
sad name row col radix mem_offset or
sad name row col radix l2[offset]

DESCRIPTION:

This ESP command adds a field to a screen that not only displays the value of the device, array, or memory, but also allows the user to modify the data on the screen.

The *sad* command has several forms of syntax to allow specification of scan string devices, array names, external memory, or other memory such as L2 cache memory.

name informs ESP which screen this *sad* command is associated with.

row and **col** specify where the screen field leftmost character should be positioned on a screen with origin 0,0.

radix specifies the format of the data in the screen field and may be any of: x= hex, d= decimal, b= binary, o= octal, a= ASCII, f= float, e= EBCDIC, or i= instruction decode.

In MP systems, the optional **pg** parameter may be specified to pinpoint the Processor Group the command is to work in. If the **pg** is not specified, then the CURRENTPG[1] is implied. In MP systems it is frequently convenient to refer to "?PG" to specify the Processor Group. This is a reference to an ESP variable "PG", which is usually set (by ESP) to the CURRENTPG name, but that may be temporarily overridded during screen creation from the *screen* command (PG=pg).

If a scan string device or an array is to be placed on the screen, then the next field must specify which **chip** the scan string or array is in.

If a scan string device is being placed on the screen then the **device** name is mentioned next, and may optionally specify bit ranges using parenthesis. If a single bit is being specified, then only the bit number need be mentioned. If a range is being specified, then **dx**[2] must specify the first bit of the device, followed by a colon, followed by **dy**, the last bit of the range.

---

1. See the *pg* command.
2. Bit range values, dx and dy, must be specified as positive decimal integer whole numbers.

If an array element is being placed on the screen then **array** specifies the name of the array, and must be followed by square brackets in which the **offset** into the array must be specified. This may optionally be followed by a bit range expression

An **offset**, wether for an array or memory, may be an absolute index into the array or memory, or may be a relative offset into the array or memory based on the screen base address. If the offset is absolute then the offset expression must be some unsigned whole number. (e.g. 14) If the offset is relative, then the offset must be expressed as a positively signed whole number. (e.g. +14)

The character width of the data field is determined by ESP. The width, in bits, of the device, array, or memory is looked up and considered with the **radix** specified.

For more information about screens and the sad command see the ESP User's Training Manual about "ESP Screens" and the ESP User's Reference Manual about "Screens". Both documents may be found in / afs/awd/public/esp/userdoc.

EXAMPLES:

Present an element of the TLB array, offset 0 from the screen base address.

sad tlb 0 10 x 603 tlb[+0](23:35)

Present system memory, screen base address + 12.

sad vrm1 11 10 x +12

When the 604reg screen is created, the HID0 register in the 604 chip of the Processor Group specified in ?PG will be accessed.

sad 604reg 5 3 x ?PG 604 HIDO

RELATED INFORMATION:

screen, sat, sal

saespco

PURPOSE: Screen Add ESP Captured Output

SYNTAX: saespco screen_name row col width height "ESP command"

DESCRIPTION:

This ESP command will cause a label to be added to a screen, **screen_name**, which will contain the output from whatever **'ESP command'** that was executed. Each time the screen is refreshed, the **'ESP command'** will be executed with its output captured and presented on the screen in the label area.

The screen label cannot be edited by the user.

Both the **width** and **height** of the label may be specified meaning that a multi-line label may be generated, **width** columns wide.

EXAMPLES:

To display the current Processor Group on a screen by executing the pg command:

saespco screen 5 3 15 1 pg

To display the current Processor Group on a screen by echoing the value of the PG variable:

saespco screen 5 3 15 1 'echo ?PG'

NOTE: If you wish for ESP to evaluate a variable each time the screen is refreshed be sure to place the ESP command in grave accents (a.k.a. backwards single quotes) instead of double quotes.

To catch the output of a user function that was dynamically loaded in a box 40 characters by 5 lines:

(Where the function was named ufunc1)

saespco screen 6 3 40 5 "ufunc1 alpha"

RELATED INFORMATION:

saco and other Screen-Add commands

saespvar

PURPOSE: Screen Add ESP Variable

SYNTAX: saespvar screen_name row col width height variable

DESCRIPTION:

This ESP command allows an ESP variable to be displayed and modified on an ESP screen. Each time the screen is updated, the latest value of the ESP variable will be presented.

The **width** as well as the **height** of the text field must be specified, allowing multiple lines for the display and modification of ESP variables with long lines.

ESP variables are usually created with the *set* command and deleted with the *unset* command. Reference to an ESP variable is usually performed with the *echo* command, as *echo ?variable*. The *saespvar* command augments these commands by allowing variables to be seen and modified from a screen as well as from the command line.

EXAMPLES:

To see and/or set the current ESPPATH:

saespvar screen 5 3 20 4 ESPPATH

RELATED INFORMATION:

saco and other Screen-Add commands, *set*, and *unset*.

sais

PURPOSE: Screen add instruction step

SYNTAX: sais screen_name row col format [pg]

DESCRIPTION: This will cause a data field to be added to a screen which contains the current number of instructions steps to be executed. This value can be changed by altering this field. This is used in the mrs.x screen.

RELATED INFORMATION:

is

sal

PURPOSE: Adds a label to a screen.

SYNTAX: sal name row col label

DESCRIPTION:

This ESP command adds a label field to a screen. The 'label' will be placed at the 'row' and 'col'umn specified, (0,0 origin), on the screen named 'name'.

See "Screens" in Part 1 for details.

EXAMPLES:

sal demo 5 67 "Extra Latches"

RELATED INFORMATION:

sat, sad

sam

PURPOSE: Screen add memory

SYNTAX: sam title|nickname increment

DESCRIPTION: On a screen that displays memory, this value will determine how many addresses the screen will scroll up or down each time the PageUp or PageDown key is pressed.

RELATED INFORMATION:

sao

sav

sanai

PURPOSE: Screen add next assembler instruction

SYNTAX: sanai screen_name row col

DESCRIPTION: This command will cause the disassembled value of the next instruction (the one pointed to by the IAR) to be displayed.

RELATED INFORMATION:

none

sao

PURPOSE: Sets screen origin for memory or array display screen

SYNTAX: sao title | nickname row col offset

DESCRIPTION: For screens displaying memory or arrays, this command contains a variable origin which is the base address from which all addresses are offset. This value is updated by the amount specified by the sam or saa command every time the page up or page down key is pressed.

RELATED INFORMATION:

sam

sav

sapb

PURPOSE: Add a push button to a screen

SYNTAX: sapb screen row column "push button label" "ESP command to execute"

DESCRIPTION: Adds a push button to a screen. The label on the push button will be "push button label".

This push button can be "clicked" by the mouse and will perform the "ESP command to execute". Execution can be a REXX file, a .x file, or any ESP command.

EXAMPLES: This is a sample screen definition full of push buttons.

```
sat control
sapb control 2 5 STOP stop
sapb control 4 5 RUN "run 0"
sapb control 6 5 STEP "is 1"
sapb control 2 15 RESET reset
sapb control 4 15 AI ai
sapb control ACLST "selftest 601 aclst 12345678 0"
sapb control 2 25 "IPLRUN 0" "iplrun 0"
sapb control 4 25 "IPLRUN 1" "iplrun 1"
screen control
```

RELATED INFORMATION:

none

sapn

PURPOSE: Screen Add Page Number

SYNTAX: sapn screen_name row col [radix]

DESCRIPTION:

This ESP command is useful on ESP screens presenting blocks of memory or array data that is relatively addressed (+offset specified).

Optional **radix** may be specified as o, x, or d, meaning octal, hex, or decimal, respectively. If omitted, *sapn* will default to decimal numbers.

More specifically, this command was created to support cache array access and display on ESP screens where groups of array address are considered logical units (cache lines) and the user wishes to get to a logical (screen full) unit by entering a cache line number.

This command causes a 4 digit editable field to be placed on the **screen_name** screen at **row**, **col**. The current base address of the screen divided by the screen increment[1] will be displayed in this field. If the user enters a number in this field, then that page number mulitplied by the current screen increment will set the screens base address.

EXAMPLES:

RELATED INFORMATION:

saco and other Screen-Add commands

1. Screen increment is provided by ESP command *sam* or *saa*.

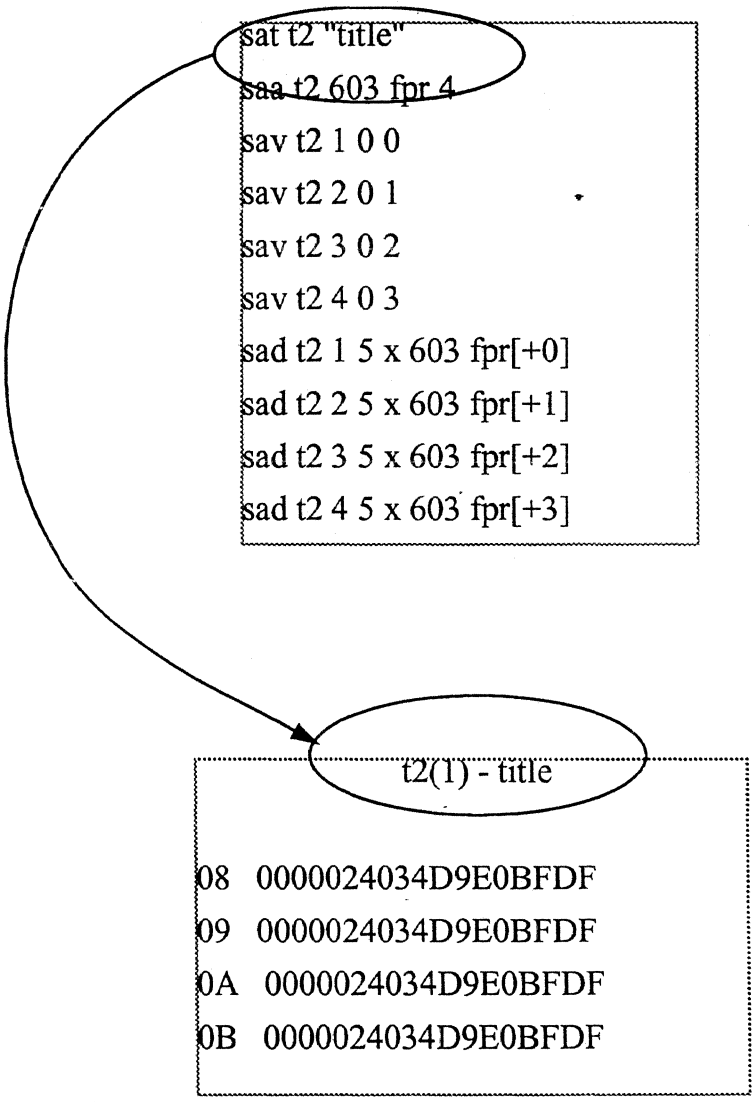
sat

PURPOSE: Creates a new screen.

SYNTAX: sat nickname title [bg=color] [fg=color] [geo=RxC]

DESCRIPTION: This ESP command creates a screen, with title 'title', that can then be refered to by other commands.

If optional argument **geo=** is specified then the screen will be created with approximately **R** rows and **C** cols inside a scrollable window.



RELATED INFORMATION:

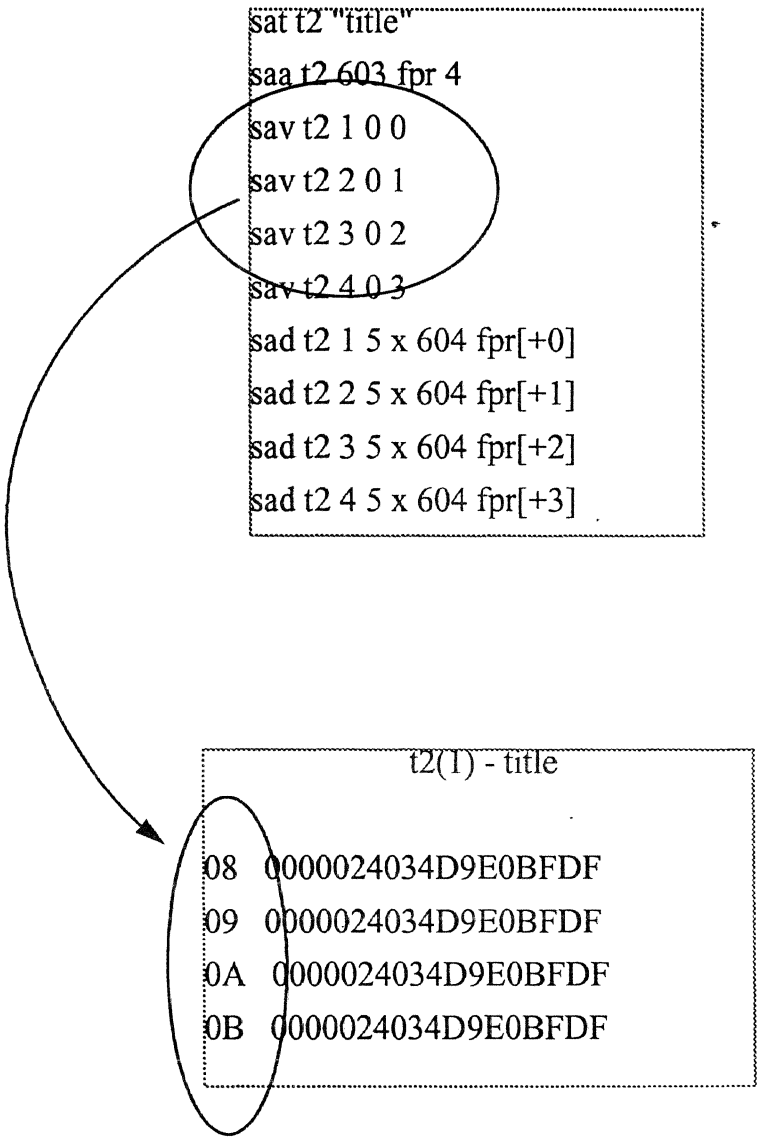
none

sav

PURPOSE: add a screen variable to a screen

SYNTAX: sav (title | nickname) row col offset

DESCRIPTION: This offset is added to the origin to determine the memory or array address that will be displayed at the location indicated.



RELATED INFORMATION:

sam

sao

save

PURPOSE: Save scan table in binary file

SYNTAX: save filename

DESCRIPTION: All data currently in ESP memory is saved in "binary" format to file 'filename'. This binary file can be read in by ESP using the 'get' command.

RELATED INFORMATION:

get

saveLayout

PURPOSE: Save current screen layout

SYNTAX: saveLayout

DESCRIPTION: Save the current ESP screen layout so that it will be restored the next time ESP is run.

ESP stores layout commands in the users HOME directory in a file called .esplayout.x.

RELATED INFORMATION:

See the ESP command "layout".

scanread

PURPOSE:

SYNTAX: scanread [(on|off)]

DESCRIPTION:

RELATED INFORMATION:

screen

PURPOSE: Put up an ESP screen

SYNTAX: for 604+ or 604e

screen name [PG=pg] [CHIP=chip] [(ERASE = | DUP=)]

for 604

screen name

DESCRIPTION:

This ESP command is used to cause an ESP screen, **name**, to be visible on the display.

If **DUP=** is not specified then the following paragraph explains how the *screen* command will behave. If the screen is already in memory, but not visible, then *screen* makes it visible. If the screen is already in memory and is also already visible, it is brought to the foreground, even if the screen window was iconified. If the screen is not found in memory, then *screen* tries to find and execute a command file with the screens **name** but with a ".x" extension.

If optional argument **DUP=** is specified then a new screen will be created even if one is already in memory and invisible or if such a screen is already up.

NOTE: Please note that only one of ERASE= and DUP= can be specified at the same time.

If optional argument **PG=pg** is applied, then while the screen is being constructed the ESP variable "PG", optionally used by the screen construction commands, will be the specified value **pg**. Once construction is finished, ESP variable "PG" is returned to its previous value.

If optional argument **CHIP=chip** is applied, then while the screen is being constructed the ESP variable "CHIP", optionally used by the screen construction commands, will be the specified value **chip**. Once construction is finished, ESP variable "CHIP" is returned to its previous value.

If optional argument ERASE= is specified then if ESP finds the named screen in memory (visible or not) the screen is deleted.

Many ESP "screen-add" commands contain an argument allowing specification of a Processor Group that the command is to apply to. If such a command refers to "?PG" as the Processor Group, then when the screen is created the value of the ESP variable "PG" will actually be specifying the Processor Group. ESP tries to keep "PG" set to the name of the CURRENTPG. If the *screen* command **PG=pg** argument

is specified, then, during the screens creation, "PG" will be set to **pg** instead of the CURRENTPG name. This trick allows the same screen definition to be used for many chips where each chip is of the same type and has the same name, but is located in a different Processor Group.

Many ESP "screen-add" commands contain an argument allowing specification of a chip that the command is to apply to. If such a command refers to "?CHIP" as the chip, then when the screen is created the value of the ESP variable "CHIP" will actually be specifying the chip. If the screen command **CHIP=chip** argument is specified, then, during the screens creation, "CHIP" will be set to **chip**. This trick allows the same screen definition to be used for many chips where each chip is of the same type, but has a different chip name.

When the *screen* command is invoked, optional parms **PG=pg** and **CHIP=chip** are parsed if present. If **PG=pg** was specified, then ESP remembers the old value of "PG" and sets "PG" to **pg**. If **CHIP=chip** was specified, then ESP remembers the old value of "CHIP" and sets "CHIP" to **chip**.

The screen name, **name**, is then used to create a command file name by having a ".x" appended to it. ESP searches for the command file **name**.x in the current directory, and then using the ESPPATH variable. If found, the command file is executed a line at a time. Normally, such a file would contain nothing but "sa", or "screen-add" type of commands, thus constructing a screen in ESP memory.

The last line of a screen file is an invocation of the *screen* command on the screen just created. This second invocation of *screen* finds the screen already in memory, and it is thus made visible.

Just before exiting, the *screen* command restores the "PG" and "CHIP" variables to their old values, if they were previously re-defined.

NOTE: A screen name is a valid command if the screen definition is in a ESP command file that has the same name as the screen.

For more information about screen creation and the *screen* command see the ESP User's Training Manual about "ESP Screens" and the ESP User's Reference Manual about "Screens". Both documents may be found in /afs/awd/public/esp/userdoc.

RELATED INFORMATION:

isd, sat, sal, sad, and other "sa" commands.

set

PURPOSE: Set a ESP variable to a value

SYNTAX: set verbose

set echo

set variable value

?variable - causes value to be executed

set - Set with no arguments will cause the program to display the current set status of all variables.

set variables

DESCRIPTION: unset turns echo and/or verbose off. It will also 'undefine' variables that have previously been set.

EXAMPLES:

SET BP1 BP X'3F00'

Sets variable 'bp1' to string bp x'3f00'.

NOTE: Note: This command, as all commands, may be included in a user's profile.

?BP1

If this was preceeded by the bp1 definition shown above, the command bp x'3f00', would be executed. The question mark preceeding the string bp1 causes ESP to look for a previously set value to substitute in its place. For this example, bp2 could be set to mean breakpoint 2, bp3 to mean breakpoint 3 and so on. However, because of the hardware implementation, ESP can never have more than one breakpoint set at a time.

SET

All set variables will be displayed

RELATED INFORMATION:

The following command variables are predefined and have special meaning. They are used to hold information for the command processor or control the operation of the command processor. Several of these variables control the processor by being set or unset. Their values are not important.

ECHO

If echo is set, then each command line will be printed on the screen after it has been tokenized and all variables have been substituted.

EXITONERR

If exitonerr is set and a command returns an exit status other than zero, then the command processor will stop reading commands from an command file and control will return to the user's command line.

NOEXECUTE

While noexecute is set, commands will be parsed but not executed. This may be useful for debugging command files. The verbose and/or echo variables could be set to trace execution without actually running commands. Note that there is no way to turn execution back on. The only way to exit the program is to press the break key. (The default AIX break key is Ctrl Backspace.)

STATUS

On exit, each command returns an exit status which tells whether or not an error occurred. The status of the last command will always be saved in the command variable, status. The set command with no arguments may be used to examine the status of the last command. If status is zero the last command completed successfully; otherwise, an error occurred.

VERBOSE

If verbose is set, then each command line will be printed exactly as it is typed.

CECHO

setclock

PURPOSE: Set COP/JTAG bus clock speed on ESP interface card

SYNTAX: setclock (0-13)

DESCRIPTION: This command will cause the interface card to choose a multiple of the scan clock period.

The clock speed being set is the CLK speed for COP bus protocol, or TCK for JTAG protocol.

The fastest speed the ESP hardware can run at is 6.25 Million Bits Per Second. This speed is set with 'setclock 0'. Table 3, below, presents the speeds at which ESP hardware can run.

Table 6: ESP Hardware Transfer Speeds

#	Bits/Sec
0	6,250,000
1	3,125,000
2	1,562,500
3	781,250
4	390,625
5	195,312
6	97,656
7	48,828
8	24,414
9	12,207
10	6,103
11	3,051
12	1,525
13	762

RELATED INFORMATION:

none

setvar

PURPOSE: Allows ESP to set a variable to an attained value.

SYNTAX: setvar var command

DESCRIPTION: The setvar command may be used with other ESP commands to set variables to the output of the ESP command.

Once a variable is set it may be used using the ESP macro expansion token: the question mark.

EXAMPLES:

```
setvar var cop data
setvar var cop data_eof
setvar var display
setvar var ver
setvar var wait

setvar alpha display 0 x
echo The value in memory at address 0 is ?alpha
```

RELATED INFORMATION:

set, echo

sjipl

PURPOSE: This command has been superseeded by "ipl"

SYNTAX:

DESCRIPTION:

RELATED INFORMATION:

ipl

sjreset

PURPOSE: This command has been superseeded by "reset".

SYNTAX:

DESCRIPTION:

RELATED INFORMATION:

reset

sleep

PURPOSE: Suspend execution for a specified time interval.

SYNTAX: sleep time

DESCRIPTION: This command is the same as the AIX sleep command. Time is in seconds.

RELATED INFORMATION:

none

sreset

PURPOSE: Toggle Or Set The SRESET Line

SYNTAX: sreset [(0|1)]

DESCRIPTION: This low level ESP command allows the hardware SRESET line to be toggled, set to zero, or set to one.

ESP performs no other action as a result of using this command. That is, buffered data and screen updates will NOT be performed because 'sreset' has been issued.

NOTE: This is the exact same command as *ipl*.

RELATED INFORMATION:

iplrun

treset

hreset

ipl

stat

PURPOSE: Display the status of all chips which are 'equipped'.

SYNTAX: stat

DESCRIPTION: For each chip which is 'equipped' 'stat' calls the 'wait' command with the -v option. Wait prints the status of each chip.

Figure 9: Examples of stat Command results

```
-----------------------------------------------------------------------
An example of a fully equipped RS6000 Release 1, 'stat' shows the fol-
lowing.
icu    ffrzOFF system{CHKSTP STOP PWROFF} chip{CHKSTP RUN  TSTCMP MAXCNT}
scu    ffrzOFF system{CHKSTP STOP PWROFF} chip{CHKSTP RUN  TSTCMP MAXCNT}
dcu0    ffrzOFF system{CHKSTP STOP PWROFF} chip{CHKSTP RUN  TSTCMP MAX-
CNT}
dcu1    ffrzOFF system{CHKSTP STOP PWROFF} chip{CHKSTP RUN  TSTCMP MAX-
CNT}
dcu2    ffrzOFF system{CHKSTP STOP PWROFF} chip{CHKSTP RUN  TSTCMP MAX-
CNT}
dcu3    ffrzOFF system{CHKSTP STOP PWROFF} chip{CHKSTP RUN  TSTCMP MAX-
CNT}
fxpt    ffrzOFF system{CHKSTP STOP PWROFF} chip{CHKSTP RUN  TSTCMP
MAXCN}T
combo1  ffrzOFF system{CHKSTP STOP PWROFF} chip{CHKSTP RUN  TSTCMP
MAXCN}T
combo2  ffrzOFF system{CHKSTP STOP PWROFF} chip{CHKSTP RUN  TSTCMP MAX-
CNT}
fpu    ffrzOFF system{CHKSTP STOP PWROFF} chip{CHKSTP RUN  TSTCMP MAXCNT}

An example of a 601 machine "stat" shows the following:
601    ffrzOFF system{CHKSTP STOP PWROFF} chip{CHKSTP RUN  TSTCMP MAXCNT}

An example of a 604 machine "stat" shows the following:
DEFAULTMUT 604   system{CKSTP PWRON}  chip{RUN }
-----------------------------------------------------------------------
```

RELATED INFORMATION:

wait

stop

PURPOSE: Stop the processor

SYNTAX: stop [-h]

DESCRIPTION:

This ESP command will stop the processor. The default is soft stop. Optional argument '-h' will have ESP perform a hard stop.

A soft stop means that the processor will stop fetching and allow all pending operations to complete before halting. When this is done it should be possible to re-start the processor where it left off.

A hard stop means that the processor clocks are shut off no matter what the processor is doing. This type of stop can not usually be recovered from.

RELATED INFORMATION:

run

iplrun

time

PURPOSE: time another command

SYNTAX: time command arg1 arg2 ...

DESCRIPTION: The time command remembers the current time, runs the specified command and computes the elapsed time. The command may be any legal ESP command, including ESP command files and REXX programs. The time is displayed in hours, minutes and seconds.

RELATED INFORMATION:

none

treset

PURPOSE: Toggle Or Set The TRST Line

SYNTAX: treset [(0|1)]

DESCRIPTION: iThis low level ESP command allows the hardware TRST line to be toggled, set to zero, or set to one.

ESP performs no other action as a result of using this command. That is, buffered data and screen updates will NOT be performed because 'treset' has been issued.

RELATED INFORMATION:

hreset

ipl

tty

PURPOSE: turns tty displaying on or off.

SYNTAX: tty [on | off]

DESCRIPTION: Normally, ESP will send messages to the TTY window. The tty command can be used to turn this displaying on or off. Without any arguments, the command will toggle between on and off depending on the previous setting. Otherwise, it can be set directly by giving the appropriate argument along with the command.

To keep output from being displayed on the TTY window type 'tty off'. To be able to see output in the TTY window type 'tty on'.

This command does not interfere with logging. Even though the TTY window might be off, any output that might have been visible is still transmitted to the log file if logging is turned on.

RELATED INFORMATION:

none

unset

PURPOSE: Un-set an ESP variable that was 'set'

SYNTAX: unset variablename

DESCRIPTION: This command unsets a variable that was previously set

unset echo - turns echo off

Commands that can be unset are: echo, esitonerr, noexecute, status, cecho and verbose.

RELATED INFORMATION:

set

update

PURPOSE: This command controls when the screen will be updated

SYNTAX: update - causes the screen to be updated

update auto - causes the screen to be completely updated whenever there is any change

update manual - the machine under test (MUT) hardware will not be updated to match what is on the screen until the "update" command is entered

DESCRIPTION: Normally, whenever any part of a screen is changed, the entire screen is updated. If all the data on the screen is data from a scan string, the time to update the entire screen is insignificant. However, if most of the screen is data from arrays, the time to update the entire screen is significant, and usually is not required. This command allows the user to control when the screen is updated.

Update with no arguments causes the screen to be updated immediately. If the argument, "auto", is passed to the update command then the screens will be updated any time a change is made. This is the default mode of operation. If the argument, "manual" is passed to the update command then the screens will be updated only when an explicit update command is issued. This is similar to automatic vs. manual recalculation options support by many spread sheet programs.

RELATED INFORMATION:

none

userbreak

PURPOSE: returns non-zero if user has pressed the break key

SYNTAX: userbreak

DESCRIPTION: ESP counts the number of times that the user presses the break key. Clearbreak resets the count to zero. Userbreak returns non-zero if the user has pressed the break key since the last time Clearbreak was called.

REXX programs which have long loops can use this function to poll for break from the user. An appropriate response to a break signal would be to return from REXX.

RELATED INFORMATION:

clearbreak

ver

PURPOSE: Get the version of this ESP program

SYNTAX: ver

DESCRIPTION:

RELATED INFORMATION:

none

verbose

PURPOSE: Echoes commands and messages to message line (or log if logging is turned on.)

SYNTAX: set verbose

unset verbose

DESCRIPTION: Echoes commands and messages to message line (or log if logging is turned on.)

If verbose is set on and log is on, any series of commands entered to the screen will also go to a log file. Later, that file can be run. This will have the same affect as re-entering exactly the same command sequence a second time. This allows for very complex initialization sequences to be saved and re-run simply by entering the name of the log file that the original commands were saved in.

RELATED INFORMATION:

none

wait

PURPOSE: The purpose of this command is to allow commands in a batch file to delay executing until a particular state is reached in the system under test. This allows a batch file to start the machine under test (MUT), and to not execute the subsequent commands until a breakpoint has been reached and DMAs and memory scrub operations have completed.

SYNTAX: wait [pg] chip seconds mask [value]

wait [pg] chip -v

DESCRIPTION: Wait up to the specified number of 'seconds' for the bits selected by the 'mask' to equal the 'value' specified. If no value is specified, the default value is 0.

If 'pg' is specified then the processor group specified will be operated on by the wait command. If 'pg' is omitted, then the current PG will be operated on.

The -v (visual) option allows a user to display the status of a chip instead of waiting for the status to be a value. 'wait' will also sense the check stop line and the power good line.

If wait with the -v option is called directly, an English statement will be displayed on the status line. For example, "wait 604 -v" would show the following:

Figure 10: Example of wait Command results

```
----------------------------------------------------------------------
 DEFAULTMUT 604 system{CHKSTP PWROFF} chip{STOP }
----------------------------------------------------------------------
```

If wait with the -v option is called from the setvar command, a hex number will be returned. For example: 3F

Time is time in seconds, from 1 to signed long int (from 1 second to 68 years).

Figure 11: Bit values for the wait command

```
----------------------------------------------------------------------
```

604 processor bit values are:

x'0001' (dec 1) Always zero (not used)

x'0002' (dec 2) Always zero (not used)

x'0004' (dec 4) Always zero (not used)

x'0008' (dec 8) Always zero (not used)

x'0010' (dec 16) Hardware status, +Power Good
x'0020' (dec 32) (not used)
x'0040' (dec 64) Hardware status, -Check Stop (0=checkstop condition)
x'0080' (dec 128) (not used)
x'0100' (dec 256) Always one (not used)
x'0200' (dec 512) Always zero (not used)
x'0400' (dec 1024) stop/run status, +stop/-run
x'0800' (dec 2048) bist status, +complete/-notcomplete
x'1000' (dec 4096) dtag bist status, +failed/-passed
x'2000' (dec 8192) dcache bist status, +failed/-passed
x'4000' (dec 32768) itag bist status, +failed/-passed
x'8000' (dec 16384) icache bist status, +failed/-passed

Sirocco bit values are:

x'0001' (dec 1) Always zero (not used)
x'0002' (dec 2) Always zero (not used)
x'0004' (dec 4) Always zero (not used)
x'0008' (dec 8) Always zero (not used)
x'0010' (dec 16) Hardware status, +Power Good
x'0020' (dec 32) (not used)
x'0040' (dec 64) Hardware status, -Check Stop (0=checkstop condition)
x'0080' (dec 128) (not used)
x'0100' (dec 256) Always one (not used)
x'0200' (dec 512) Always zero (not used)
x'0400' (dec 1024) stop/run status, +stop/-run
x'0800' (dec 2048) bist status, +complete/-notcomplete
x'1000' (dec 4096) other bist fails, +failed/-passed
x'2000' (dec 8192) dcache bist status, +failed/-passed
x'4000' (dec 32768) non-repairable cache, +failed/-passed
x'8000' (dec 16384) icache bist status, +failed/-passed

EXAMPLE: After starting BIST, to wait for BIST complete, use the 'wait' command. The mask is set for x'800' since the bist status bit is located there. The value is also set to x'800' since we wish to see the bist complete status at value one. We allow for up to 10 seconds for the BIST to complete. (Before timing out and giving up.)

```
wait 604 10 x'800' x'800'
```

RELATED INFORMATION:

stat

which

PURPOSE: Locates a ESP command file or REXX program.

SYNTAX: which "filename"

DESCRIPTION: This command is used to locate a ESP command file or REXX program.

An attempt to find the file using the ESPPATH is made, and if that fails then an attempt to find the file using the PATH variable is made (as if it were a REXX program.)

If the file is found then the full path and file name are printed in the tty window. If the file is not found then nothing is printed in the tty window.

RELATED INFORMATION:

xlist

filefinder

ls

xlist

PURPOSE: Get a listing of all ESP command files. (Those with the '.x' extension.)

SYNTAX: xlist

DESCRIPTION: This command brings up a window that lists all the available dot x files and their directories. This list may be paged up or down using the Page Up and Page Down keys, or using the built in scroll bar. A single file may be selected using a single mouse click, and the file may be executed with a double click.

The files found with 'xlist' are those located in the current directory as well as all those located in the ESPPATH directories. ESPPATH is an ESP variable set when ESP is invoked.

RELATED INFORMATION:

ls

which

filefinder

Symbols

! 47

A

Add
chip to chip data base 83
COP command 69
Port name 79
Processor Group name 77
scan string 80
scan string to chip data base 83
Add ESP Captured Output 199
Add ESP Variable To Screen 201
Add Page Number To Screen 209
aet 50
AIX
Running AIX commands/programs from ESP 47
alter 52
Array
add 192
add array definition 63
Array Definition
Example 64
Arrays
modify 52
Assembler instruction 205
autoupdate 57

B

beacon 59
bells 60
Binary scan table file 122
bp 61, 215
branch trace 62
Break key 233
Break signals
clear 89
Breakpoint
address 193
breakpoints 61, 215
bt 62

C

caa 63
cac - chip add chip 67
cacopcmd 69
cad 70
example 75
campg 76
capg 77
caport 79
cass 80
example 81
cat 83
cbuf 84
cd 87
cecho 88
Chip
add to chip data base 83
Clear
break signals 89
clearbreak 89
Clock speed
set 220
cls 90
Command
drtrymode 104
reducedpinmode 186
time 228
Command file
locates 239
COMMAND FILES 35
Command files

list 240
Commands
! 47
aet 50
alter 52
autoupdate 57
beacon 59
bells 60
bp 61, 215
bt 62
caa 63
cac 67
cacopcmd 69
cad 70
campg 76
capg 77
caport 79
cass 80
cat 83
cbuf 84
cd 87
cecho 88
clearbreak 89
cls 90
configure 91
cop 92
copcmd 97
coplog 98
copstub 99
cs 100
dirty 101
display 102
dump 105
dynload 107
echo 108
enable 109
equip 110
err 111
esp 112
exit 114
expect 115
faclist 117
filefinder 119
flush 120
get 122
gexpect 123
gread 124
gwrite 126
help 128
hreset 129
in 130
in16 132
in32 134
ioflag 135
ipl 136
iplrun 137
is 140
isd 142
jtag 143
layout 144
list 145
listall 146
load 147
load_cntr 149
log 150
ls 151
mal 152
mat 153
meminit 154
memread 155
memwrite 156
menu 157
mmior 158
mmiow 159
move 160
number 161
ocs 163
out 164
out16 166
out32 168
pages 169, 187
pause 170
pg 171
pinread 173
pinwrite 175
pioe 177
pior 178
piow 179
por 180
prs 181
pwd 182
quit 183

readcon 185
reset 188
resetint 189
run 190
saa 192
sabreak 193
sacs 194, 195
sad 196
saespco 199
saespvar 201
sais 202
sal 203
sam 204
sanai 205
sao 206
sapb 207
sapn 209
sat 210
sav 211
save 212
savelayout 213
screen 216
set 218
setclock 220
setvar 221
sleep 224
sreset 225
stat 226
stop 227
time 228
treset 229
tty 230
unset 231
update 232
userbreak 233
ver 234
verbose 235
wait 236
which 239
xlist 240
Compare scan string facilities, arrays, memory 115
configure 91
cop 92
COP Commands
examples 94
copcmd 97
coplog 98
copstub 99
cs 100
CURSOR MOTION 30
Cycle step 194, 195
Cycle step handler 100

D

Device
create 70
Devices
list defined 146
Directory
display 151
dirty 101
Display
status of chips 226
display 102
Display list of ESP commands 128
Download file from host to MUT memory 147
drtrymode 104
dump 105
dynload 107

E

echo 108
Echo command and command line arguments 88
Echo commands and messages 235
Edit
end 31
keys 31
start 31

enable 109
equip 110
Equiped chips 226
err 111
Error condition 111
esp 112
exit 114
expect 115
Extended Transfer Protocol 177, 179

F

faclist 117
filefinder 119
flush 120

G

get 122
gexpect 123
Graphics I/O read operation 124
Graphics I/O test operation 123
Graphics I/O write operation 126
gread 124
gwrite 126

H

Halt ESP execution 170
Hard stop 61, 62
help 128
hreset 129

I

in 130
in16 132
in32 134
Instruction step 140
  add 202
Interrupt line 189
ioflag 135
ipl 136
iplrun 137
is 140
isd 142

J

jtag 143

L

layout 144
list 145
List details about internal ESP objects 145
listall 146
load 147
load_cntr 149
LOADING MEMORY 39
Location of executed file 119
log 150
Log messages to file 150
ls 151

M

mal 152
mat 153
meminit 154
Memory
  modify 52
Memory access 53
Memory accessing 154
Memory data list 155

Memory display screen
set origin 206
Memory Images
of scan strings 53
Memory mapped devices 158, 159
memread 155
memwrite 156
Menu
add line to 152
create 153
display custom 157
menu 157
Menus
an example 32
calling up a 32
definition 31
Full Titles 33
making a 31
nicknames 32
title 32
Message numbers 161
mmior 158
mmiow 159
move 160
Multiprocessor
Add Multiprocessor group 76
Add processor group name 77

N

number 161

O

ocs 163
out 164
out16 166
out32 168

P

pages 169, 187
pause 170
pg 171
pinread 173
pinwrite 175
pioe 177
pior 178
piow 179
POR 188
por 180
Processor
stop 227
Processor group 171
Program debug 99
prs 181
Push button
add 207
pwd 182

Q

quit 183
Quit ESP 183

R

Read
memory mapped devices 158
pio 178
readcon 185
reducedpinmode 186
Reset
interrupt line 189
soft 137
reset 188
Reset MUT 180

resetint 189
REXX
  locates program 239
  location of executed file 119
Run
  MUT 190
run 190

S

saa 192
sabreak 193
sacs 194, 195
sad 196
saespco 199
saespvar 201
sais 202
sal 203
sam 204
sanai 205
sao 206
sapb 207
sapn 209
sat 210
sav 211
save 212
savelayout 213
Scan string
  add to chip data base 83
  distinguish between different types of scan string 80
Scan Strings
  modify 52
Scan Table
  Updating 40
Scan table
  save in binary file 212
Screen
  add a push button 207
  add a screen variable 211
  add an array 192
  add cycle step 194, 195
  add editable data field 196
  add ESP captured output 199
  add ESP variable 201
  add instruction step 202
  add label 203
  add memory 204
  add next assembler instruction 205
  add Page Number 209
  create 210
  list screen pages in memory 169, 187
  locate or load 144
  prints 181
  reposition 160
  save current layout 213
  set breakpoint address 193
  show 216
  update 232
screen 216
Screen Commands 27
set 218
Set origin 206
setclock 220
setvar 221
Simulate error condition 111
sjipl 222
sjreset 223
sleep 224
Soft stop 61, 62
Specifying chip types as COP or JTAG 83
Specifying JTAG Instruction Register Width 83
sreset 225
Start the Interactive Screen Designer 142

stat 226
Stop
  processor 227
  soft 190
stop 227
Suspend execution
  for specified time interval 224

T

time 228
trap 61, 62
treset 229
TTY
  display 230
tty 230
tty window
  clear 90

U

unset 231
update 232
userbreak 89, 233

V

Variable
  screen 211
  set to a value 218
  set to an attained value 221
  unset 231
ver 234
verbose 235
Version of ESP 234

W

wait 236
which 239
Write
  list of words to memory 156
  memory mapped devices 159

X

xlist 240