



# 57

# AIX

*July 2000*

---

## **In this issue**

- 3 Web-based ping from AIX
- 15 Check mail utility – part 2
- 21 Mailto - an AIX Web server extension
- 41 Building freeware and shareware
- 45 New features of RS/6000 hardware and software
- 52 AIX news

---

© Xephon plc 2000

# update

# AIX Update

---

## Published by

Xephon  
27-35 London Road  
Newbury  
Berkshire RG14 1JL  
England  
Telephone: 01635 550955  
From USA: 01144 1635 33823  
E-mail: [harryl@xephon.com](mailto:harryl@xephon.com)

## North American office

Xephon/QNA  
Post Office Box 350100, Westminster CO  
80035-0100, USA  
Telephone: (303) 410 9344  
Fax: (303) 438 0290

## Contributions

If you have anything original to say about AIX, or any interesting experience to recount, why not spend an hour or two putting it on paper? The article need not be very long – two or three paragraphs could be sufficient. Not only will you actively be helping the free exchange of information, which benefits all AIX users, but you will also gain professional recognition for your expertise and that of your colleagues, as well as being paid a publication fee – Xephon pays at the rate of £170 (\$250) per 1000 words for original material published in AIX Update.

To find out more about contributing an article, see *Notes for contributors* on Xephon's Web site, where you can download *Notes for contributors* in either text form or as an Adobe Acrobat file.

## Editor

Harold Lewis

## Disclaimer

Readers are cautioned that, although the information in this journal is presented in good faith, neither Xephon nor the organizations or individuals that supplied information in this journal give any warranty or make any representations as to the accuracy of the material it contains. Neither Xephon nor the contributing organizations or individuals accept any liability of any kind howsoever arising out of the use of such material. Readers should satisfy themselves as to the correctness and relevance to their circumstances of all advice, information, code, JCL, scripts, and other contents of this journal before making any use of it.

## Subscriptions and back-issues

A year's subscription to *AIX Update*, comprising twelve monthly issues, costs £180.00 in the UK; \$275.00 in the USA and Canada; £186.00 in Europe; £192.00 in Australasia and Japan; and £190.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the November 1995 issue, are available separately to subscribers for £16.00 (\$23.00) each including postage.

## AIX Update on-line

Code from *AIX Update* is available from Xephon's Web page at [www.xephon.com/aixupdate.html](http://www.xephon.com/aixupdate.html) (you'll need the user-id shown on your address label to access it).

---

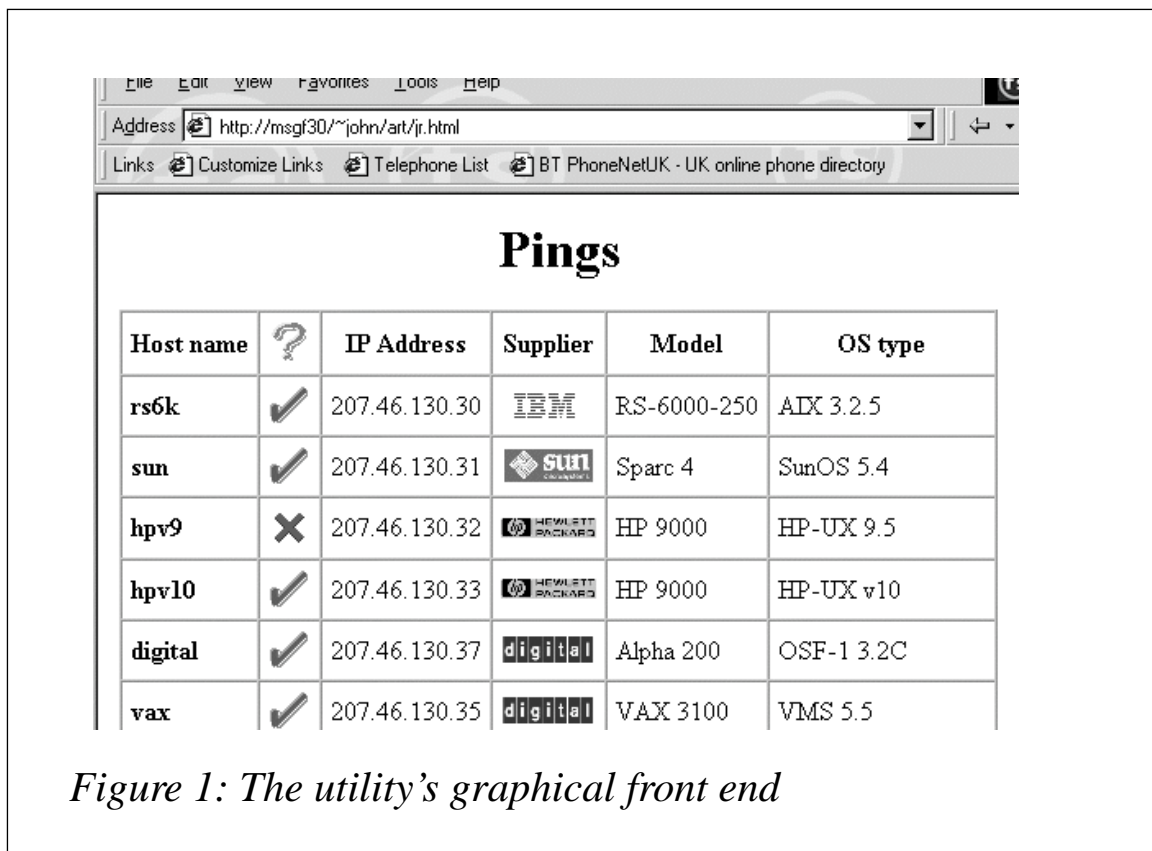
© Xephon plc 2000. All rights reserved. None of the text in this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior permission of the copyright owner. Subscribers are free to copy any code reproduced in this publication for use in their own installations, but may not sell such code or incorporate it in any commercial product. No part of this publication may be used for any form of advertising, sales promotion, or publicity without the written permission of the publisher. Copying permits are available from Xephon in the form of pressure-sensitive labels, for application to individual copies. A pack of 240 labels costs \$36 (£24), giving a cost per copy of 15 cents (10 pence). To order, contact Xephon at any of the addresses above.

*Printed in England.*

## Web-based ping from AIX

In this article I explore the subject of implementing a Web-based 'ping' utility and take a look at the thinking and tricks and techniques behind it. By dissecting a script for such a tool, I hope to share with you some tips that may be of use to you for writing your own scripts (Web-based or otherwise).

This is an AIX-based tool that can assist in the management of multiple machines on an intranet or the Internet running either AIX or other operating systems. From time to time, machines go off-line for maintenance and other reasons. When this happens it's normal to ping the machines to check whether they've come back on-line. The actual response time of the ping is rarely noted, what most users want to know is whether the machine is alive or dead. This tool enhances ping by giving it a graphical Web-based front-end that reports on the status of all machines of interest at a glance. This is best illustrated by a screenshot of the result as it appears on a browser (see Figure 1).



## HOW IT WORKS

The tool consists of a single shell script, *ping.cgi*, that generates HTML dynamically (the script conforms with the CGI standard) plus a collection of small graphics in GIF format, which make the result easier on the eye.

The script pings each machine you specify using a timeout of one second. An HTML table is then generated by the script using the results.

A full listing of the script and the HTML is included later in the article. In 'pseudo code', this is what the script does:

- 1 Create an HTML header.
- 2 Create HTML to define the start of a table.
- 3 Carry out the following procedure for each machine whose status is required:
  - Ping the machine.
  - If the machine is alive, set the result to a picture of a green tick.
  - If the machine is dead, set the result to a picture of a red cross.
  - Create an HTML table row entry for this machine.
- 4 End the table.
- 5 End the HTML.

This pseudo-code translates directly into the function *DO\_main* that appears at the top of *ping.cgi*:

```
DO_main ()
{
    DO_HTML_Header
    DO_HTML_Table_Start

    for PLATFORM in rs6k sun hpv9 hpv10 digital vax alpha as400
    do
        DO_Timed_Ping $PLATFORM
        DO_Set_Graphic_According_to_Ping_Result $?
        DO_HTML_Table_Row
```

```

done

DO_HTML_Table_End
DO_HTML_Footer
}

```

I find that functions are a convenient way of giving a script both structure and readability. However, after implementing *DO\_main*, remember to call it at the end of the script!

I'll now examine each function called by *DO\_main* in more detail, looking at them in the order in which they appear:

- *DO\_HTML\_Header*
- *DO\_HTML\_Table\_Start*
- *DO\_Timed\_Ping*
- *DO\_Set\_Graphic\_According\_to\_Ping\_Result*
- *DO\_HTML\_Table\_Row*
- *DO\_HTML\_Table\_End*
- *DO\_HTML\_Footer*.

Many of the functions use the *echo* command. By echoing HTML directly to *stdout* and running the script from a browser via CGI, the HTML generated is displayed on the user's browser.

#### DO\_HTML\_HEADER

```

DO_HTML_Header ()
{
    echo "<html>"
    echo "<head>"
    echo "  <title>Pings</title>"
    echo "</head>"
    echo ""
    echo "<body>"
    echo "<h1 align=center>Pings</h1>"
    echo "<p>"
}

```

This function generates the tags required at the top of a page of HTML and aligns the title of the page on the centre.

## DO\_HTML\_TABLE\_START

```
DO_HTML_Table_Start ()
{
    echo "<table border align=center cellpadding=5>"
    echo "  <caption align=bottom> ping results</caption>"
    echo "    <tr>"
    echo "      <th align=center>Host name</th>"
    echo "      <th align=center><IMG SRC=images/question.gif></th>"
    echo "      <th align=center>IP Address</th>"
    echo "      <th align=center>Supplier</th>"
    echo "      <th align=center>Model</th>"
    echo "      <th align=center>OS type</th>"
    echo "    </tr>"
}
```

As the name suggests, this function generates the tags required to begin a table definition in HTML in preparation for the table rows that follow.

## DO\_TIMED\_PING

```
DO_Timed_Ping ()
{
    DO_Timeit 1 "ping ${1} 56 1" >/dev/null 2>&1
    return $?
}
```

This function takes a machine domain name (or IP address) as a parameter and issues a **ping** command with a timeout of one second. This timeout is important to ensure that the script executes in a timely manner, even when some or all the machines on the list are down. On most networks, a ping will return in a matter of 10 milliseconds, so a timeout of one second is plenty. If you intend to use this script to monitor machines on a WAN, you may want to use a bigger timeout. The output of **ping** is ignored except for the return status indicating either success ('0') or failure ('1'). The return status is passed back in *\$?* to the calling function, *DO\_Timed\_Ping*, which in turn calls *DO\_Timeit* (which is covered later in this article).

## DO\_SET\_GRAPHIC\_ACCORDING\_TO\_PING\_RESULT

```
DO_Set_Graphic_According_to_Ping_Result ()
{
    if [ $1 -ne 0 ]
```

```

then
    GRAPHIC="images/no.gif"
else
    GRAPHIC="images/yes.gif"
fi
}

```

This function simply takes the return value of the **ping**, which is passed to the function as parameter *\$1*, and sets the environment variable *GRAPHIC* to the name of a file containing either a picture of a green tick or a red cross (*yes.gif* or *no.gif* respectively), depending on the value of *\$1*. (Note the use of the continuation character, ‘>’, which indicates a formatting line break that’s not present in the original source code.)

## DO\_HTML\_TABLE\_ROW

```

DO_HTML_Table_Row ()
{
    case ${PLATFORM} in
        rs6k)
            HOST_TYPE="RS-6000-250"
            OSYS_NAME="AIX 3.2.5"
            TEST_TARG=UNIX
            PICCY=images/logos/ibmsmall.gif
            ;;

```

(The code is truncated; see the listing at the end for the full source.)

```

        *)
            HOST_TYPE="Unknown"
            OSYS_NAME="Unknown"
            TEST_TARG=Unsupported
            PICCY=
            ;;

        esac

        IP_ADDR=`nslookup ${PLATFORM} | grep -v Alias | tail -2 | head -1 |
        > cut -c10-`

        echo "    <tr>"
        echo "        <td align=left><b>${PLATFORM}</b></td>"
        echo "        <td align=center><IMG SRC=${GRAPHIC}></td>"
        echo "        <td align=left>${IP_ADDR}</td>"
        echo "        <td align=center><IMG SRC=${PICCY}></td>"
        echo "        <td align=left>${HOST_TYPE}</td>"
        echo "        <td align=left>${OSYS_NAME}</td>"

```

```

    echo "    </tr>"
}

```

As the name suggests, this function generates the tags for a table row. The first part of the script sets up the environment variables that are used by the second part, which generates the actual HTML. The line beginning *IP\_ADDR* finds the IP address of the domain name passed in the environment variable, *PLATFORM*.

## DO\_TIMEIT

```

DO_Timeit()
{
    interval=$1 export interval
    shift
    $* &
    pid=$! export pid
    (
        sleep $interval
        kill -HUP $pid >/dev/null 2>&1
    ) &
    wait $pid
}

```

*DO\_Timeit* is a generic function that can be used in many scripts. It allows any command to be executed, along with a full set of parameters and a timeout. **ping**, for example, should return a result quickly on a fast network if the machine to be ‘pung’ is in good health, but will take more than a few seconds to return if the machine is dead. Just as importantly, if the command executes quickly, we don’t want to wait for the timeout to expire.

The first parameter (passed in *\$1*) is the timeout value in seconds. This is copied to the environment variable *INTERVAL* for use later. The *shift* command moves all parameters that were in *\$2*, *\$3*, and *\$4* left by one to *\$1*, *\$2*, and *\$3*. This is a way of getting rid of *\$1*, leaving the actual command to be executed in *\$1*, with parameters in *\$2*, *\$3*, etc. The strange looking command:

```
$* &
```

executes the command itself – the ampersand (‘&’) is there to request the command to be executed in the background as another AIX process. At this point, *\$!* contains the process ID. This is copied to the environment variable *pid*.



Now comes the important part – the timeout. To kill or not to kill – that is the question! As mentioned earlier, we have to code the function so it doesn't wait for the timeout to expire if the command itself returns quickly. This is achieved by executing **sleep** followed by the **kill** command as a separate process, so we end up with three processes running simultaneously – the script itself, the command being executed (in this case **ping**), and the **sleep** command.

So, the job of the 'main' function is to start the two background processes, then wait for the 'command process' (**ping**) to finish – hence the command **wait \$pid**. If the command completes before the timeout value, the *DO\_Timeit* function completes and returns to the calling function. This has the effect of killing the other process, which is likely to be in the middle of executing the **sleep** command. On the other hand, if the command doesn't complete before the timeout value, the process executing the **sleep** command will finish first and kill the command process. This **kill** command kills the command that was executed earlier (**ping**, in our case), and then the whole function completes and returns to the caller. The parenthesis followed by ampersand ensure that the **sleep** command followed by the **kill** command execute one after the other in the same background process.











## HTML OUTPUT FROM PING.CGI

This is the HTML generated by the script that resulted in the screenshot shown in Figure 1.

```
<html>
<head>
<title>Pings</title>
</head>

<body>
<h1 align=center>Pings</h1>
<p>

<table border align=center cellpadding=5>
  <caption align=bottom> ping results</caption>
  <tr>
    <th align=center>Host name</th>
    <th align=center><IMG SRC=images/question.gif></th>
    <th align=center>IP Address</th>
    <th align=center>Supplier</th>
```

Model	OS type
<b>rs6k</b>	
207.46.130.30	
RS-6000-250	AIX 3.2.5
<b>sun</b>	
207.46.130.31	
Sparc 4	SunOS 5.4
<b>hpv9</b>	
207.46.130.32	
HP 9000	HP-UX 9.5
<b>hpv10</b>	
207.46.130.33	
HP 9000	HP-UX v10
<b>digital</b>	
207.46.130.37	
Alpha 200	OSF-1 3.2C

```
<td align=left><b>vax</b></td>
<td align=center><IMG SRC=images/yes.gif></td>
<td align=left> 207.46.130.35</td>
<td align=center><IMG SRC=images/logos/decsmall.gif></td>
<td align=left>VAX 3100</td>
<td align=left>VMS 5.5</td>
</tr>
```

```
<tr>
<td align=left><b>alpha</b></td>
<td align=center><IMG SRC=images/yes.gif></td>
<td align=left> 207.46.130.34</td>
<td align=center><IMG SRC=images/logos/decsmall.gif></td>
<td align=left>Alpha 200</td>
<td align=left>Open VMS 6</td>
</tr>
```

```
<tr>
<td align=left><b>as400</b></td>
<td align=center><IMG SRC=images/no.gif></td>
<td align=left> 207.46.130.36</td>
<td align=center><IMG SRC=images/logos/ibmsmall.gif></td>
<td align=left>AS/400</td>
<td align=left>OS-400</td>
</tr>
```

```
<tr>
<td align=left><b>scrumpy</b></td>
<td align=center><IMG SRC=images/yes.gif></td>
<td align=left> 207.46.130.21</td>
<td align=center><IMG SRC=images/logos/intel3.gif></td>
<td align=left>Pentium PC</td>
<td align=left>Windows NT Server</td>
</tr>
```

```
</TABLE>
</body>
</html>
```

## PING.CGI

```
#!/bin/ksh
```

```
set -x
```

```
DO_main ()
{
    DO_HTML_Header
    DO_HTML_Table_Start
```

```

for PLATFORM in rs6k sun hpv9 hpv10 digital vax alpha as400 scrumpy
do
    DO_Timed_Ping $PLATFORM
    DO_Set_Graphic_According_to_Ping_Result $?
    DO_HTML_Table_Row
done

DO_HTML_Table_End
DO_HTML_Footer
}

DO_Timed_Ping ()
{
    DO_Timeit 1 "ping ${1} 56 1" >/dev/null 2>&1
    return $?
}

DO_Set_Graphic_According_to_Ping_Result ()
{
    if [ $1 -ne 0 ]
    then
        GRAPHIC="images/no.gif"
    else
        GRAPHIC="images/yes.gif"
    fi
}

DO_Timeit()
{
    interval=$1 export interval
    shift
    $* &
    pid=$! export pid
    (
        sleep $interval
        kill -HUP $pid >/dev/null 2>&1
    ) &
    wait $pid
}

DO_HTML_Header ()
{
    # echo "Content-type: Text/HTML"
    # echo
    echo "<html>"
    echo "<head>"
    echo "<title>Pings</title>"
    echo "</head>"
    echo ""
    echo "<body>"
}

```

```

    echo "<h1 align=center>Pings</h1>"

    echo "<p>"
}

DO_HTML_Table_Start ()
{
    echo "<table border align=center cellpadding=5>"
    echo "  <caption align=bottom> ping results</caption>"
    echo "    <tr>"
    echo "      <th align=center>Host name</th>"
    echo "      <th align=center><IMG SRC=images/question.gif></th>"
    echo "      <th align=center>IP Address</th>"
    echo "      <th align=center>Supplier</th>"
    echo "      <th align=center>Model</th>"
    echo "      <th align=center>OS type</th>"
    echo "    </tr>"
}

DO_HTML_Table_End ()
{
    echo "</TABLE>"
}

DO_HTML_Footer ()
{
    echo "</body>"
    echo "</html>"
}

DO_HTML_Table_Row ()
{
    case ${PLATFORM} in
        alpha)
            HOST_TYPE="Alpha 200"
            OSYS_NAME="Open VMS 6"
            TEST_TARG=VMS
            PICCY=images/logos/decsmall.gif
            ;;
        as400)
            HOST_TYPE="AS/400"
            OSYS_NAME="OS-400"
            TEST_TARG=OS400
            PICCY=images/logos/ibmsmall.gif
            ;;
        digital)
            HOST_TYPE="Alpha 200"
            OSYS_NAME="OSF-1 3.2C"
            TEST_TARG=UNIX
    esac
}

```

```
PICCY=images/logos/decsmall.gif
;;

hpv9)
HOST_TYPE="HP 9000"
OSYS_NAME="HP-UX 9.5"
TEST_TARG=UNIX
PICCY=images/logos/hpsmall.gif
;;

hpv10)
HOST_TYPE="HP 9000"
OSYS_NAME="HP-UX v10"
TEST_TARG=UNIX
PICCY=images/logos/hpsmall.gif
;;

scrumpy)
HOST_TYPE="Pentium PC"
OSYS_NAME="Windows NT Server"
TEST_TARG=WNTS
PICCY=images/logos/intel3.gif
;;

rs6k)
HOST_TYPE="RS-6000-250"
OSYS_NAME="AIX 3.2.5"
TEST_TARG=UNIX
PICCY=images/logos/ibmsmall.gif
;;

rslab)
HOST_TYPE="RS-6000-250"
OSYS_NAME="AIX 3.2.5"
TEST_TARG=UNIX
PICCY=images/logos/ibmsmall.gif
;;

sun)
HOST_TYPE="Sparc 4"
OSYS_NAME="SunOS 5.4"
TEST_TARG=UNIX
PICCY=images/logos/sunsmall.gif
;;

vax)
HOST_TYPE="VAX 3100"
OSYS_NAME="VMS 5.5"
TEST_TARG=VMS
PICCY=images/logos/decsmall.gif
;;
```

```

*)
HOST_TYPE="Unknown"
OSYS_NAME="Unknown"
TEST_TARG=Unsupported
PICCY=
;;

esac

IP_ADDR=`nslookup ${PLATFORM} | grep -v Alias | tail -2 | head -1 |
➤ cut -c10-`

echo "      <tr>"
echo "      <td align=left><b>${PLATFORM}</b></td>"
echo "      <td align=center><IMG SRC=${GRAPHIC}></td>"
echo "      <td align=left>${IP_ADDR}</td>"
echo "      <td align=center><IMG SRC=${PICCY}></td>"
echo "      <td align=left>${HOST_TYPE}</td>"
echo "      <td align=left>${OSYS_NAME}</td>"
echo "      </tr>"
}

# DO_main $* >/dev/null 2>&1
DO_main $*

exit 0

```

## GIF FILES

Note that this script doesn't provide the *.gif* files that are used in the final presentation. One source for such files is the various vendor Web sites – *www.ibm.com*, *www.sun.com*, etc. There you should find appropriate graphics, though permission may be required to use them.

---

*John Rainford (UK)*

© Xephon 2000

---

## Check mail utility – part 2

This month's instalment concludes this article on the check mail utility (the first and only other instalment appeared in last month's issue).

## LISTING KCHKMD.SH

```
#####  
# Name      : kchkmd.sh ( kill check mail daemon )  
#  
# Description: Kills the daemon process chkmd.sh that checks mail.  
#  
# Notes     : 1. The script contains the following functions:  
#             o main  
#             o InitializeVariables  
#             o InstanceCheck  
#             o RootUser  
#             o DisplayMessage  
#             o MoveCursor  
#             o ProcessExit  
#  
#           2. The script must be run from the root account.  
#  
#           3. It may take a while to kill the daemon as the  
#             daemon receives signal 15 and waits for all its  
#             child proceses to complete.  
#####  
# Name      : InitializeVariables  
#  
# Description: Initializes all the required variables  
#####  
InitializeVariables ()  
{  
#  
# define message prefixes  
#  
ERROR="kchkmd.sh:ERROR:"  
INFO="kchkmd.sh:INFO:"  
#  
# define escape sequences  
#  
ESC="\0033["  
RVON=_[7m           # reverse video on  
RVOFF=_[27m         # reverse video off  
BOLDON=_[1m         # bold on  
BOLDOFF=_[22m       # bold off  
BON=_[5m            # blinking on  
BOFF=_[25m          # blinking off  
#  
# define return codes  
#  
TRUE=0  
FALSE=1  
#  
# define exit codes
```



```

#
FEC=1
SEC=0
#
# define messages
#
NOT_ROOT_USER="Must execute the script from root account${RVOFF}"
DAEMON_NOT_RUNNING="Deamon process is not running${RVOFF}"
KILLING_DAEMON="Killing daemon process${RVOFF}"
DAEMON_KILLED="Successfully killed daemon process${RVOFF}"
DAEMON_NOT_KILLED="Failed to kill daemon process${RVOFF}"
OS_ERROR="\${ERR_MSG}${RVOFF}"
}
#####
# Name      : MoveCursor
#
# Input     : Y and X coordinates
#
# Returns   : None
#
# Description: Moves the cursor to the required location (Y, X).
#
# Notes     : 1. The function must be run in ksh for print to work.
#####
MoveCursor ( )
{
YCOR=$1
XCOR=$2
echo "${ESC}${YCOR};${XCOR}H"
}
#####
# Name      : DisplayMessage
#
# Description: Displays a message.
#
# Input     : 1. Message type (E = Error, I = Informative)
#           : 2. Error Code as defined in DefineMessages ().
#           : 3. Message to be acknowledged flag
#
# Notes     : 1. The user must acknowledge the message if the
#           : acknowledgement flag is set to Y.
#####
DisplayMessage ( )
{
MESSAGE_TYPE=$1
MESSAGE_TEXT=`eval echo $2`
ACK_MESSAGE="$3"
if [ "${ACK_MESSAGE}" = "" ]
then
    ACK_MESSAGE="Y"

```

```

fi
clear
MoveCursor 24 1
if [ "${MESSAGE_TYPE}" = "E" ]
then
    echo "`eval echo ${RVON}${ERROR}`${MESSAGE_TEXT}\c"
else
    echo "`eval echo ${RVON}${INFO}`${MESSAGE_TEXT}\c"
fi
#
# let the user acknowledge the message
#
if [ "${ACK_MESSAGE}" = "Y" ]
then
    read DUMMY
fi
return ${TRUE}
}
#####
# Name      : ProcessExit
#
# Description: The function processes a graceful exit.
#
# Input     : 1. Exit Code
#####
ProcessExit ()
{
#
# assign parameters
#
EXIT_CODE="$1"
exit $EXIT_CODE
}
#####
# Name      : InstanceCheck
#
# Description: Checks to see if the mail check daemon is running.
#
# Returns   : $TRUE if daemon process is running
#            $FALSE otherwise.
#####
InstanceCheck ()
{
if ps -eaf | grep "chkmd.sh" | grep -v "grep" > /dev/null 2>&1
then
#
# an instance is running
#
return $TRUE
else

```

```

        return $FALSE
    fi
}
#####
# Name      : RootUser
#
# Description: Checks to see if the user is root.
#
# Returns   : TRUE if user is root
#             FALSE otherwise.
#####
RootUser ()
{
USER=`id | cut -d '(' -f2 | cut -d ')' -f1`
if [ "${USER}" = "root" ]
then
    return $TRUE
else
    return $FALSE
fi
}
#####
# Name      : KillDaemonProcess
#
# Description: Kills the daemon process.
#
# Returns   : $TRUE or
#             $FALSE.
#####
KillDaemonProcess ()
{
PID=`ps -eaf | grep "chkmd.sh" | grep -v "grep" | grep -v "kchkmd" |
➤ awk {'print $2'}`
kill -15 $PID
while true
do
    if ps -eaf | grep "chkmd.sh" | grep -v "grep" | grep -v "kchkmd"
➤ > /dev/null 2>&1
    then
        #
        # still waiting for child process to complete
        #
        :
    else
        return $TRUE
    fi
done
}

```

```

#####
# Name      : main
#
# Description: Invokes all other functions.
#
# Notes     : The function calls the following functions:
#             o InitializeVariables
#             o InstanceCheck
#             o RootUser
#             o KillDaemonProcess
#             o DisplayMessage
#             o ProcessExit
#####
main ()
{
InitializeVariables
if ! RootUser
then
    DisplayMessage E "${NOT_ROOT_USER}"
    ProcessExit $FEC
fi
if ! InstanceCheck
then
    DisplayMessage E "${DAEMON_NOT_RUNNING}"
    ProcessExit $FEC
fi
DisplayMessage I "${KILLING_DAEMON}"
if ! KillDaemonProcess
then
    DisplayMessage E "${DAEMON_NOT_KILLED}"
    ProcessExit $FEC
else
    DisplayMessage I "${DAEMON_KILLED}"
fi
ProcessExit $SEC
}
#
# invoke main
#
main

```

---

*Arif Zaman*  
*DBA/Administrator*  
*High-Tech Software (UK)*

© Xephon 2000

---

# Mailto - an AIX Web server extension

## INTRODUCTION

Mailto is a CGI Web server extension written in C. We run this program under AIX 4.2 with Netscape Enterprise Server 3.62, but it should work with any other AIX-compatible Web server.

Mailto's purpose is to gather, format, and send information from a standard Web form to one or more e-mail addresses. The e-mail is formatted either automatically or using a sophisticated command-driven process. Mailto's rich instruction set means you don't have to write a custom program to handle every form that you post on the Web.

Mailto can be invoked simply by assigning its name and path to a *FORM*'s *ACTION* attribute, specifying the e-mail address of the recipient using a hidden *INPUT* tag. An optional hidden *INPUT* tag can be used to set the subject of the e-mail. An example of these tags appears below. While Mailto's preferred *METHOD* is *POST*, it also works with *GET*.

```
<FORM ACTION="http://www.server.name/CgiDirPath/Mailto" METHOD=POST>  
<INPUT TYPE=HIDDEN NAME=_TO VALUE="Receiver_Email_address">  
<INPUT TYPE=HIDDEN NAME=_SUBJECT VALUE="Message subject matter">
```

The body of the message is built using the name and content of all *INPUT*, *SELECT*, and *TEXTAREA* tags on the form other than those whose name starts with an underscore character ('\_'). The latter are used by Mailto to control processing, as we shall see later.

Name-value pairs are listed in the same order as their associated tags appear on the form. By default, the name and value are separated by an equal sign ('=') preceded and followed by a space. Each name-value pair is separated by a line break. This may be changed using *\_SUFFIX\_NAME* and *\_SUFFIX\_VALUE*, which contain character strings to be inserted respectively after the name and value of all name-value pairs.

Additional text may be inserted before or after any name-value pair using one or more *\_WRITE* or *\_WRITE\_COND* instructions before or

after the associated tag. To omit a name-value pair from a message, add the `_INVISIBLE` attribute to the tag. To omit a value only if the user has left the field blank, use `_INVISIBLE_COND`.

Automatic processing of name-value pairs is disabled when the attribute `_INVISIBLE` is applied to user input tags. In such circumstances, a more natural style of prose can be generated in the message body using the `_WRITE` and `WRITE_COND` instructions. Output text encoded with these statements may contain references to form fields; each reference is replaced by the value of the named `<INPUT>`, `<SELECT>`, or `<TEXTAREA>`.

Six field attributes may be applied to input tags to force special processing, such as conversion of the user's input to uppercase or lowercase, conditional or unconditional exclusion of a field from the message body, restriction of a field to numeric data, and enforcement of mandatory fields (fields the user must complete). Another subset of instructions enables full control over the mail message's headers, such as *From:*, *To:*, *Subject:*, *Cc:*, *Bcc:*, and *Reply-To:*.

Mailto also implements a sophisticated 'IF block' conditional structure to test the content of input fields. This feature allows selective printing and message formatting based on user input data. For example, an IF block may be used to check input data in order to select the most appropriate URL to which to redirect the user when the program terminates.

There are no restrictions on the number of user input tags a form may contain. Mailto-extended data validation and error reporting also has the benefit of easing the burden of testing and debugging – on-screen error messages always offer a direct link to on-line help files, and the program also implements a debug mode in which messages are not mailed but displayed on the browser screen, along with extensive debugging information.

Mailto is very safe to use on the 'Net, as **sendmail** is invoked internally through a pipe, not a system call, and no static buffers are ever used—everything is managed through dynamic memory allocation. A log file also traces all calls. Use of the program may be restricted to forms from known domain names by using a validation list.

## PROGRAM INSTALLATION AND CUSTOMIZATION

Installing Mailto is fairly straightforward. Begin with program customization, enabling optional features, and setting path and file names according to your site-specific standards. Customization guidelines are given below. When you have finished this part, compile the source code using the command:

```
cc -o Mailto Mailto.c
```

and place the executable file in the Web server's CGI directory. The last task is to copy the on-line HTML help file into a convenient Web directory.

Customizable statements are located near the top of the program, right after 'include' statements. Default settings are shown below.

```
#define MAILPGM "/usr/sbin/sendmail -t -oi -oem" /* Invoke sendmail */
#define TMPFILE "/tmp/Mailto.tmp" /* Temporary work file */
#define LOGFILE "/usr/log/Mailto.log" /* Log file absolute path */
#define NSQFILE "/usr/log/Mailto.noseq" /* Save file: _$NOSEQ fct */
#define ONLNOTE 0 /* On-line help file */
#define HLPHTTP "http://www.server.name" /* URL-serv to help file */
#define HLPHTML "/FullWebPath/Mailto.htm" /* URL-file to help file */
#define DFLTSND "Web@Some.Net" /* Default sender e-mail */
#define DFLTSUB "No title." /* Default e-mail subject */
#define DOMCHCK 0 /* Referrer domain check */
#define DOMLIST \
{\
    ".domain.name.suffix",\
} /* List authorized domains */
```

*MAILPGM* is the absolute path to **sendmail**'s executable file, appended with appropriate options (**-t -oi -oem** are mandatory). *TMPFILE* is the full path name of a work file that's created by Mailto whenever necessary (the person responsible for running the Web server needs permission to create files in the directory in which *TMPFILE* resides).

*LOGFILE* is the full path name of the log file and *NSQFILE* is the full path name of the file used to save the current *\_\$NOSEQ* settings. Both files, which are initially empty, should exist and the person who runs the Web server should be granted read and write permissions to them using either base Unix permissions or AIX ACL extended permissions.

If you do not intend to install the on-line help file, assign the value '0' to *ONLNOTE* and ignore both *HLPHTTP* and *HLPHTML*. Otherwise,

set *ONLNOTE* to '1' and assign the URL of the server hosting Mailto's on-line HTML help file to *HLPHTTP* and the path to the help file to *HLPHTML*.

*DFLTSND* is the fully-qualified e-mail address of the default sender, which is used on the *From:* message header when one is not explicitly set. *DFLTSUB* is the default string assigned to the *Subject:* header when one is not set explicitly using *\_SUBJECT*.

Assign the value '1' to *DOMCHCK* if you wish to enable referrer domain checking. This setting restricts Mailto usage to pages on domains, sub-domains, or machines explicitly listed in *DOMLIST*. When *DOMCHCK* is '0', Mailto can be called from anywhere on the 'Net.

## ON-LINE HELP AND DEBUG MODE

Users don't need to memorize URLs either to execute the program or view on-line documentation. While Mailto is invoked using the URL *http://www.server.name/CgiDirPath/Mailto*, the URL to access to the on-line help is obtained by adding *PATHINFO/help* to the server's URL, as shown below. Error reporting screens always offer a direct link to this help file.

```
http://www.server.name/CgiDirPath/Mailto/help
```

*FORM* tags that invoke Mailto using the following *ACTION* attribute:

```
ACTION="http://www.server.name/CgiDirPath/Mailto"
```

result in normal processing, while adding either */debug* or */debug/* to the end of the URL invokes the program in debug mode (see the examples below). In this mode, the formatted message is not mailed to the user but is displayed on the user's screen, along with extensive debugging information. Note that the two forms of the command are not identical – */debug/* provides more extensive information than */debug*.

```
ACTION="http://www.server.name/CgiDirPath/Mailto/debug"
```

```
ACTION="http://www.server.name/CgiDirPath/Mailto/debug/"
```



## MAILTO INSTRUCTION SET

Mailto processing is controlled by instructions embedded in Web forms using special *INPUT* tags. These tags must conform to the following rules:

- The *INPUT* tag must have the attribute *TYPE="HIDDEN"* to ensure the field is not rendered on screen.
- The value of the *NAME* attribute must be a string containing only uppercase characters preceded by an underscore (‘\_’) – the string is the name of the instruction to be performed on the server.
- The attribute *VALUE* is assigned a string that’s used by the targeted instruction.

Below is an example of an embedded instruction (the continuation character, ‘>’, indicates a formatting line break):

```
<INPUT TYPE="HIDDEN" NAME="_SUBJECT" VALUE="Reservation  
> acknowledgement">
```

Consequently, avoid using an underscore as the first character of the names of other *<INPUT>*, *<SELECT>*, and *<TEXTAREA>* fields, as these will be mistaken for Mailto statements.

Instructions are grouped into three sets: mailing instructions, message formatting instructions, and control instructions. The table in Figure 1 opposite lists all available instruction codes by category along with their default value and a short description, where applicable.

Every form processed by Mailto must contain at least one *\_TO* or *\_TO\_IXI* instruction that identifies the message receiver. Apart from this, no other instruction is mandatory. When applicable, the default values listed in the table are used.

All instructions may occur more than once in a form. However, including multiple instances of the instructions *\_FROM*, *\_LINE*, *\_MARGIN*, *\_MAX\_GAP*, *\_SUFFIX\_NAME*, *\_SUFFIX\_VALUE* and *\_XFR\_URL* outside of a conditional construct is simply redundant. When Mailto comes across more than one instance of any of the above instructions, it checks all instances for correct syntax and discards all but the last one.

<b>Instruction</b>	<b>Default</b>	<b>Argument (mailing instruction)</b>
_TO	None	E-mail address of one receiver for public group mailing
_TO_1X1	None	E-mail address of one receiver for private separate mailing
_FROM	'Web@Some.Net'	Sender's e-mail address
_SUBJECT	'No title.'	Mail subject matter
_CC	None	E-mail address to forward a manifest carbon copy
_BCC	None	E-mail address to forward a dissimulated carbon copy
_REPLY_TO	None	E-mail address where a reply is expected
<b>Instruction</b>	<b>Default</b>	<b>Argument (formatting instruction)</b>
_LINE	80	Maximum message line width in characters
_MARGIN	0	Left margin size in characters
_MAX_GAP	Value of _LINE	Maximum gap from right margin when folding a long line
_SUFFIX_NAME	' = '	String printed following field name
_SUFFIX_VALUE	'\n'	String printed following field content
_WRITE	Null string	Text to be printed at current position
_WRITE_COND	Null string	Ditto, if all referenced fields are non-blank
<b>Instruction</b>	<b>Default</b>	<b>Argument (control instruction)</b>
_XFR_URL	None	URL of page to transfer to when Mailto terminates
_MANDATORY	None	Name of a field that cannot be left blank
_NUMERICAL	None	Name of a field that should be numerical only
_UPPERCASE	None	Name of a field to be translated into uppercase
_LOWERCASE	None	Name of a field to be translated into lowercase
_INVISIBLE	None	Name of a field to be excluded from message
_INVISIBLE_COND	None	Name of a field to be excluded if left blank
_IF	None	{Field-Name}.OP.String
_AND	None	{Field-Name}.OP.String
_OR	None	{Field-Name}.OP.String
_ELSE_IF	None	{Field-Name}.OP.String
_ELSE	None	Optional comment
_END_IF	None	Optional comment

*Figure 1: Summary of commands*

Except for instructions that are executed in an IF block and ones that are context-sensitive (*\_WRITE* and *\_WRITE\_COND*), all other instructions are order- and position-independent. However, to increase legibility and ease of debugging, I consider it good programming practice to group all Mailto commands at the start of a form immediately after the *<FORM>* tag.

When coding tags, remember that strings assigned to keywords are case-sensitive. Any character with an ISO-8859-1 code may appear both in strings assigned to *NAME* and *VALUE* attributes. However, Mailto will scan all such strings and replace all non-printable characters (codes 0x00 to 0x1f and 0x80 to 0xa0) with blanks, with the exception of line breaks (code 0x0a), which are preserved.

## REFERENCES TO USER INPUT AND CONTEXT VARIABLES

Data entered in a particular field can be referenced elsewhere on the form by enclosing the field name between a pair of braces. For example, *{Street-Address}* refers to the information typed in the field named 'Street-Address'.

Some instructions (*\_TO*, *\_TO\_IX1*, *\_FROM*, *\_CC*, *\_BCC*, and *\_REPLY\_TO*) accept only a single reference without any surrounding text. For example, the tag:

```
<INPUT TYPE="HIDDEN" NAME="_FROM" VALUE="{Sender-Email}">
```

tells Mailto that the e-mail address to use on the *From:* header is the one typed or selected by user in the *<INPUT>*, *<SELECT>*, or *<TEXTAREA>* field named 'Sender-Email'.

A single reference may also appear in the argument of *\_IF*, *\_AND*, *\_OR*, and *\_ELSE\_IF* conditional instructions. Consider the following tag:

```
<INPUT TYPE="HIDDEN" NAME="_IF" VALUE="{City}.EQ.Leeds">
```

This evaluates to *TRUE* if the value of the field named 'City' is 'Leeds' ('EQ' is the operator for 'equals'). The syntax and use of IF blocks is explained later.

Other instructions (*\_SUBJECT*, *\_WRITE*, and *\_WRITE\_COND*) may

include any number of references embedded in text or otherwise. For example:

```
VALUE="You will be enrolled in group {No-Grp} of course  
    > {No-Crs};\n"
```

is a valid argument for the instruction *NAME="\_WRITE"*. Under Mailto processing, the references *{No-Grp}* and *{No-Crs}* are replaced by the value of the fields named *No-Grp* and *No-Crs*.

A reference is said to be ‘objectless’ when no input field has the same name as the reference. In such cases, no substitution is possible and the objectless reference is left in place and appears ‘as is’ in the message body.

A reference is said to be ‘multi-object’ when many input fields have the same name as the reference. A multi-object reference is replaced by the string obtained by concatenating the contents of all non-empty occurrences of the named field. The string is prefixed with an opening

<b>Variable</b>	<b>Description</b>	<b>Example</b>
_\$DATE	Current date	2000/03/11
_\$YEAR	Year	2000
_\$YY	Year of the century	00
_\$MM	Month of the year as a decimal	03
_\$DD	Day of the month as a decimal	11
_\$MONTH	Month of the year as a literal	March
_\$YRDAY	Day of the year as a decimal	070
_\$WKDAY	Day of the week as a literal	Saturday
_\$TIME	Current time	16:17:47
_\$HRS	Hours decimal figure	16
_\$MIN	Minutes decimal figure	17
_\$SEC	Seconds decimal figure	47
_\$REFERER	URL of calling page	http://www.srv. nam/Path/Ex.htm
_\$NOSEQ	Dynamic form sequence number	37

*Figure 2: Context variables*

brace (‘{’) and suffixed with a closing brace (‘}’). The field values are concatenated in the order in which they appear in the form, and are separated from each other by the string ‘ & ’ (notice the space on either side of the ampersand).

Mailto supports a variety of context variables, which are referenced in the same way as user input fields. Mailto replaces a reference to a context variable (such as *{\_ \$DATE}*) that occurs in the text part of a *\_SUBJECT*, *\_WRITE*, or *\_WRITE\_COND* statement with the current value of the named variable.

You may use this feature to include in the message body the current date and time, the day of the week, the URL of the calling page, or even a sequence number that is incremented by one each time Mailto is invoked from a particular page. Figure 2 on page 35 lists all context variables implemented in Mailto.

You may include literal left and right braces in a *\_SUBJECT*, *\_WRITE*, and *\_WRITE\_COND* statement by ‘escaping’ them (*\{* and *\}*); this stops them being interpreted as reference initiators and terminators. Similarly, a literal ‘\’ should be escaped as ‘*\\*’. Note that, of all Mailto instructions, only *\_FROM* can be the target of a reference (as in *{\_FROM}*).

## MAILING INSTRUCTIONS

There are seven mailing instructions: *\_TO*, *\_TO\_1X1*, *\_FROM*, *\_SUBJECT*, *\_CC*, *\_BCC*, and *\_REPLY\_TO*. They are used to set the corresponding mail headers of the e-mail message under construction.

Except for *\_SUBJECT*, whose argument is a string of characters (with or without embedded references), all other mailing instructions have either an e-mail address or a reference to a field where the user has typed or selected such an address as an argument. Note that only the first occurrence of the named field is considered when a reference is used in this context.

Only one address or reference is allowed per statement. To specify more, just use additional statements. Always use fully qualified addresses, including both the user name and domain name – if you omit the domain name, your mail server may append a default one

(usually the domain name of the server) to the user name. Mailto checks the syntax of all addresses, but cannot verify that an address with a valid form really is associated with a user.

```
<INPUT TYPE=HIDDEN NAME=_TO VALUE="Email">  
<INPUT TYPE=HIDDEN NAME=_TO_1X1 VALUE="Email">
```

At least one *\_TO* or *\_TO\_1X1* instruction is mandatory in all Web forms. Use as many of these statements as are required to list all addressees. The argument is either an e-mail address or a reference to a field containing such an address

Addresses from all *\_TO* statements are concatenated and the resulting string is assigned to the envelope's *To:* header. Copies of the message, wrapped in identical envelopes, are shipped to all the addressees, who are all aware of each other. Copies of the message are also sent to all addressees listed in *\_TO\_1X1* statements, but in this case the envelope's *To:* header lists only the recipient, without mentioning any other *\_TO* or *\_TO\_1X1* addressees. Examples are given below.

```
<INPUT TYPE=HIDDEN NAME=_TO VALUE="Mary_Poppins@Magik.land.uk">  
<INPUT TYPE=HIDDEN NAME=_TO VALUE="{Your-email-addr}">  
<INPUT TYPE=HIDDEN NAME=_TO_1X1 VALUE="Ali.Baba@hotmail.com">
```

The *\_FROM* instruction is optional; its argument, which is either an e-mail address or a reference to a field bearing such an address, identifies the sender of the message. This information sets the value of the message envelope's *From:* header. When the form does not include a *\_FROM* statement, Mailto uses the default sender address defined at installation time. If multiple *\_FROM* statements are encountered, only the last one is retained for processing.

```
<INPUT TYPE=HIDDEN NAME=_FROM VALUE="Email">
```

Using *{\_FROM}* on a *\_CC*, *\_BCC*, *\_REPLY\_TO*, *\_SUBJECT*, and *\_WRITE* statements recalls the sender's e-mail address, which may be taken from a *\_FROM* statement, a reference to a field, or the default. You may also use *{\_FROM}* in a *WRITE\_COND* statement, though the default value is never used in this context and this instruction is discarded if the sender's e-mail is not explicitly defined.

```
<INPUT TYPE=HIDDEN NAME=_FROM VALUE="Very.clean@mister.net">  
<INPUT TYPE=HIDDEN NAME=_FROM VALUE="{User_Email_Address}">
```

Use one or more *\_SUBJECT* statements to define the exact wording of the message envelope's *Subject:* header. This text is obtained by concatenating the *VALUE* of all *\_SUBJECT* statements in the form in the order in which they appear and without adding any demarcation character. Use the string '\n' to add line breaks to the text. You may also embed references to *<INPUT>*, *<SELECT>*, and *<TEXTAREA>* fields to recall their value. References to context variables are also allowed.

```
<INPUT TYPE=HIDDEN NAME=_SUBJECT VALUE="Message title">
```

A line may be wrapped if it exceeds either the default maximum line width or the width explicitly set by a *\_LINE* statement. Refer to the *\_MAX\_GAP* instruction for an explanation of where forced line breaks occur within lines exceeding the limit. Under debug mode, a line that is broken up is right-padded with a backslash ('\') to indicate that the line break results from the maximum line width. Note that a line width of ten characters is used when the value of a *\_LINE* statement is less than this.

Always maintain a one-blank character margin after the initial *Subject:* line and continuation lines, if required by mail utilities. When either no *\_SUBJECT* statement is encoded or all references lead to empty strings, Mailto uses the subject 'No title.' (or another default subject defined at installation time). Here are some examples.

```
<INPUT TYPE=HIDDEN NAME=_SUBJECT VALUE="Information request">  
<INPUT TYPE=HIDDEN NAME=_SUBJECT VALUE="on {Topic} - {$_DATE}.">
```

When necessary, use *\_CC* and *\_BCC* instructions to identify where to ship carbon copies of the message. The argument of these statements is either a single explicit e-mail address or a reference to a field containing such an address. If the sender (either the default sender or an explicit one) is also to receive a copy, use the construct: *VALUE={\_FROM}*.

```
<INPUT TYPE=HIDDEN NAME=_CC VALUE="Email">  
<INPUT TYPE=HIDDEN NAME=_BCC VALUE="Email">
```

Use as many *\_CC* and *\_BCC* statements as are necessary to list all recipients. The message's envelope contains no information about *\_BCC* recipients, though the *Cc:* header lists all addresses from *\_CC* instructions. Some examples are:

```
<INPUT TYPE=HIDDEN NAME=_BCC VALUE="Abc.Xyz@cdc.com">
<INPUT TYPE=HIDDEN NAME=_CC VALUE="{Supervisor}">
<INPUT TYPE=HIDDEN NAME=_CC VALUE="{_FROM}">
```

The optional *\_REPLY\_TO* instruction is used to include all return addresses that should be available to a recipient replying to the message. The argument of this statement is either a single explicit e-mail address or a reference to a field bearing such an address. If the sender's address, either the default address or an explicit one, is also one of the return addresses, you can use a *VALUE={\_FROM}* construct to list it. Use as many *\_REPLY\_TO* statements as are necessary to list all return addresses. Some examples follow.

```
<INPUT TYPE=HIDDEN NAME=_REPLY_TO VALUE="Adam@01d.paradise.com">
<INPUT TYPE=HIDDEN NAME=_REPLY_TO VALUE="{Selected-Email}">
<INPUT TYPE=HIDDEN NAME=_REPLY_TO VALUE="{_FROM}">
```

## MESSAGE FORMATTING INSTRUCTIONS

Formatting instructions belong in one of two groups. The first includes the following instruction:

- *\_LINE*
- *\_MARGIN*
- *\_MAX\_GAP*
- *\_SUFFIX\_NAME*
- *\_SUFFIX\_VALUE*.

These statements set the value of parameters used to build the message overall; they include line width, left margin, demarcation characters, and how far from the right margin to look for a suitable place to split an oversized line.

Default values apply to statements that are not included in the form. When multiple occurrences of the same statement are encountered, only the last one is used, though all are checked for syntax errors.

The second group of formatting instructions has only two members: *\_WRITE* and *\_WRITE\_COND*. These statements either complement or supersede the name-content pairs from user input fields.



The message body is built according to the following rules:

- All *<INPUT>*, *<SELECT>*, and *<TEXTAREA>* fields are reported other than those explicitly excluded by the *\_INVISIBLE* attribute.
- The field name is printed, followed either by the string ‘ = ’ (including the leading and trailing spaces) or the string specified in a *\_SUFFIX\_NAME* statement, then the field content, and lastly either a line break character (‘\n’) or the string defined in a *\_SUFFIX\_VALUE* statement.

Name-value pairs are reported in the same order as their tags appear in the form. To omit a field when it’s left blank, use the *\_INVISIBLE\_COND* attribute. Additional text may be printed before or after any name-value pair by including one or more *\_WRITE* or *\_WRITE\_COND* instructions before or after the tag.

Text in write statements may embed any number of references to user input fields or context variables. A *\_WRITE\_COND* statement is not processed if one or more fields referenced in its argument are blank. Applying the *\_INVISIBLE* attribute to all user fields and using *\_WRITE* and *\_WRITE\_COND* statements allows a message to be written in natural prose, something that’s not possible using automatic processing of name-value pairs.

A long string does not have to be encoded in a single instruction – it may be more convenient to distribute the text among a number of consecutive statements. Line breaks are never implied – they must be set explicitly with a ‘\n’ in a *\_WRITE*, *\_WRITE\_COND*, *\_SUFFIX\_NAME*, or *\_SUFFIX\_VALUE* statement.

You may also embed a tab character (‘\t’) in *\_SUFFIX\_NAME*, *\_WRITE*, and *\_WRITE\_COND* statements when the rest of the line is to be aligned on the right margin, with the ‘leading character’ or ‘tab leader’ being the first character following the tab sequence (often a dot or a space character). In any line, there should be no more than one tab (Mailto silently ignores any other tabs on the same line).

The use of tabs makes sense only if the message is to be displayed or printed with a non-proportional fixed-width font. For example, let’s

assume a form contains three user fields named 'Yellow', 'Orange', and 'Red' that have values 'Light', 'Dark', and 'Medium' respectively. The output below is produced if the user fields are processed according to the following Mailto instructions (which appear before the form elements):

```
<INPUT TYPE=HIDDEN NAME=_SUFFIX_NAME VALUE=" \t. ">
<INPUT TYPE=HIDDEN NAME=_SUFFIX_VALUE VALUE="\n">
<INPUT TYPE=HIDDEN NAME=_LINE VALUE=40>
<INPUT TYPE=HIDDEN NAME=_WRITE VALUE="Parm Name \t Parm Value\n">
<INPUT TYPE=HIDDEN NAME=_WRITE VALUE="\t-\n\n">
```

This produces the output:

```
Parm Name                Parm Value
-----
Yellow ..... Light
Orange ..... Dark
Red ..... Medium
```

Mailto ensures that lines never exceed the default or explicit maximum line width. Oversized lines are split, and such lines are marked with a trailing backslash character in debug mode. Splits occur at word boundaries whenever possible; however, when it is impossible to locate a word boundary within a reasonable distance from the end of the line, lines are cut right after the last allowable position, even if this means splitting a word. The acceptable distance is set using a MAX\_GAP statement.

```
<INPUT TYPE=HIDDEN NAME=_LINE VALUE="Nbr-of-characters">
```

Use the LINE statement to set the maximum line width of the message body in characters. This figure must be a non-zero positive integer not exceeding 512, and must not include a sign, comma, or decimal point. The line width also applies to the text of the *Subject*: header, which is further restricted to being between 10 and 512.

Note that the line width excludes the left margin. When the form does not include a LINE statement, Mailto uses a default line width of 80. If multiple LINE statements are encountered, only the last one is used ('132' in the example below).

```
<INPUT TYPE=HIDDEN NAME=_LINE VALUE="80">
<INPUT TYPE=HIDDEN NAME=_LINE VALUE="132">
```

The *\_MARGIN* statement defines the left margin in characters. Its argument must be a positive integer not exceeding 512, and excluding a sign, comma, or decimal point. When there is no *\_MARGIN* statement in a form, a default margin size of zero applies and printing starts at column one. If more than one *\_MARGIN* statement occurs in a form, all but the last one are ignored. When a non-zero left margin is defined, each line of text is preceded by *\_MARGIN* space characters. Remember that the margin size is not included in the line width. Here are examples:

```
<INPUT TYPE=HIDDEN NAME=_MARGIN VALUE="6">
<INPUT TYPE=HIDDEN NAME=_MARGIN VALUE="0">
```

As with other formatting instructions, the value of *\_MAX\_GAP* must be a positive integer, without sign, comma, or decimal point, and not exceeding 512. This figure is used when splitting a line exceeding the maximum size. When *\_MAX\_GAP* is zero, the line is split right after the last character, even if this breaks the last word of the line.

When *\_MAX\_GAP* is non-zero, Mailto breaks an oversized line at a word boundary, if one occurs within *\_MAX\_GAP* characters of the end of the line. In this case, the line break occurs right after a blank character or the punctuation mark; otherwise it's after the last character on the line.

When a form does not include a *\_MAX\_GAP* instruction, Mailto uses a default value equal to the line size (either the default value or one set by a *\_LINE* statement).

```
<INPUT TYPE=HIDDEN NAME=_MAX_GAP VALUE="20">
<INPUT TYPE=HIDDEN NAME=_MAX_GAP VALUE="0">
```

The *\_SUFFIX\_NAME* instruction defines the demarcation string to print between the field name and its content when Mailto reports user fields (the default is '='). The demarcation string may include a new line ('\n') or a tab ('\t') sequence.

For example, consider a form that contains three user fields, 'Alpha', 'Beta', and 'Pi', that have the values 'aa', 'bbbb', and 'ccccc' respectively. When *\_SUFFIX\_NAME*, *\_SUFFIX\_VALUE*, and *\_LINE* are set as follows:

```
<INPUT TYPE=HIDDEN NAME=_SUFFIX_NAME VALUE=" \t- ">
```

```
<INPUT TYPE=HIDDEN NAME=_SUFFIX_VALUE VALUE=";\n">
<INPUT TYPE=HIDDEN NAME=_LINE VALUE=27>
```

The name-content pairs reported are:

```
Alpha ----- aa;
Beta ----- bbbb;
Pi ----- ccccc;
```

Other examples of *\_SUFFIX\_NAME* statements are shown below.

```
<INPUT TYPE=HIDDEN NAME=_SUFFIX_NAME VALUE=" : ">
<INPUT TYPE=HIDDEN NAME=_SUFFIX_NAME VALUE=" is ">
<INPUT TYPE=HIDDEN NAME=_SUFFIX_NAME VALUE=" \t. ">
```

The *\_SUFFIX\_VALUE* statement defines what should be printed right after the field content to demarcate successive name-content pairs. If this instruction is omitted, the default value is ‘\n’. Within the demarcation string, use ‘\n’ to force a line break and ‘\t’ to right-align the remainder of the current line. Some examples are shown below.

```
<INPUT TYPE=HIDDEN NAME=_SUFFIX_VALUE VALUE=".\n">
<INPUT TYPE=HIDDEN NAME=_SUFFIX_VALUE VALUE="\n\n">
```

The context-sensitive instructions *\_WRITE* and *\_WRITE\_COND* are used to insert text into the message body. Place them before or after a user input tag respectively to precede or follow the name-value pair with the text encoded as argument. In addition to references to user-input fields and context variables, the string argument may also include ‘\n’ and ‘\t’.

## CONTROL INSTRUCTIONS

There are thirteen control instructions available as follows:

- One redirection instruction
- Six attribute enabling statements
- Six ‘IF block’ constructors.

*\_XFR\_URL* is the redirection instruction; it defines the page to which the user is redirected when Mailto finishes processing the current form. All types of URL are allowed, including static HTML page addresses and CGI program calls (note, however, that the value of the *\_XFR\_URL* statement is not syntax checked). When a form does not

include an *\_XFR\_URL* statement, no redirection occurs and the user is left with a default termination message on screen.

Even if relative URLs work, I strongly recommend that you always use absolute URLs. This is mandatory when the target page is password-protected. Examples of this type of statement are shown below.

```
<INPUT TYPE=HIDDEN NAME=_XFR_URL VALUE="http://www.Eden.com/  
    > Abel/msg.htm ">  
<INPUT TYPE=HIDDEN NAME=_XFR_URL VALUE="/Sales/Ok.html">  
<INPUT TYPE=HIDDEN NAME=_XFR_URL VALUE="/cgi-bin/dbu?key=179&  
    > n=x ">
```

The control statements that are used to enable attributes are:

- *\_MANDATORY*
- *\_NUMERICAL*
- *\_LOWERCASE*
- *\_UPPERCASE*
- *\_INVISIBLE*
- *\_INVISIBLE\_COND.*

A control statement invokes special processing on the field named in the *VALUE* attribute. Through this mechanism, you may request user input to be converted to uppercase or lowercase, a field to be excluded from message body either unconditionally or if it's blank, that the user is required to complete a field, and that a field is restricted to numerical data. It is a good practice to group control statements near the top of the form before all user input fields.

If a particular control statement applies to many fields, use separate statements for each. When a statement applies to all fields, you may indicate this with the shortcut *VALUE=\_ALL*. This, however, cannot be used with the statement *\_MANDATORY*.

*\_MANDATORY* requires the user to complete the target field, while *\_NUMERICAL* restricts it to numeric data. Appropriate error messages are issued when a user fails to comply with these requirements. A field is considered to be complete when one of its occurrences (if more than

one field in the form has the same name) contains at least one non-blank character. A numerical field may contain only digits, blanks (which are discarded), a plus or minus sign at the very start or end of the field, and a decimal point or comma.

*\_UPPERCASE* and *\_LOWERCASE* convert data in the named fields in the way their names suggest. If both are applied to the same field, the last one prevails. Mailto omits fields that are the target of an *\_INVISIBLE* statement from the message body; it also omits a field that is the target of an *\_INVISIBLE\_COND* attribute if all occurrences of the field are blank. *\_INVISIBLE* supersedes *\_INVISIBLE\_COND* when both are applied to the same field.

Note that a user field may be the target of many different attributes and that an attribute applies to all occurrences of the target field, if more than one field on the form has the target name. I recommend that you group control statements near the top of the form before user input fields, especially before Mailto instructions embedding a reference to a user field that is the target of a control statement. This ensures that all special processing is performed on user fields before they are referenced elsewhere.

If the target of a control statement is an empty or non-existent field, Mailto silently ignores the statement, as long as it is not *\_MANDATORY*. Note that, among all Mailto instructions, only *\_FROM* can be the target of a control statement, as long as the statement is not *\_NUMERICAL*.

```
<INPUT TYPE=HIDDEN NAME=_MANDATORY VALUE="_FROM">
<INPUT TYPE=HIDDEN NAME=_MANDATORY VALUE="AGE">
<INPUT TYPE=HIDDEN NAME=_NUMERICAL VALUE="AGE">
<INPUT TYPE=HIDDEN NAME=_INVISIBLE_COND VALUE="_ALL">
<INPUT TYPE=HIDDEN NAME=_INVISIBLE VALUE="Submit">
<INPUT TYPE=HIDDEN NAME=_UPPERCASE VALUE="State">
<INPUT TYPE=HIDDEN NAME=_LOWERCASE VALUE="code">
```

IF blocks comprise the last group of control instructions. Their operation codes are:

- *\_IF*
- *\_AND*

- *\_OR*
- *\_ELSE\_IF*
- *\_ELSE*
- *\_END\_IF*.

These statements are used to frame other Mailto instructions that should be processed only when certain conditions are met.

Use IF blocks to select, for example, a different addressee, subject title, or redirection URL, depending on the content of a field. IF blocks are also frequently used to frame *\_WRITE* statements when some text or the content of some fields should be printed only when some fields have certain values.

IF block constructors are used to frame other Mailto instructions whose processing should be conditional. A conditional block is always initiated by an *\_IF* statement and terminated by an *\_END\_IF* statement. Between those two statements only other Mailto instructions may appear – *<INPUT>*, *<SELECT>*, and *<TEXTAREA>* fields would be reported as errors.

In its simplest form, a conditional block consists of a single logical section where a group of Mailto statements is preceded by an *\_IF* and followed by an *\_END\_IF*. If the condition evaluates to *True*, the statements are processed, otherwise they are bypassed after the usual syntax check.

In its most general form, a conditional block comprises any number of logical sections that are initiated by an *\_IF* or *\_ELSE\_IF* statement (or an *\_ELSE*, if it's the last statement). A logical section finishes where a new one starts or the block is terminated by an *\_END\_IF* statement.

If the condition of the *\_IF* statement evaluates to *True*, statements in the enclosed section are processed and those in all other sections are bypassed. If it evaluates to *False*, the *\_IF* section is bypassed, and the next *\_ELSE\_IF* section is evaluated. As soon as one of the *\_ELSE\_IF* statements evaluates to *True*, instructions in it are executed and all other *\_ELSE\_IF* sections to the closing *\_END\_IF* statement are

bypassed. If all conditional proposals evaluate to *False*, statements in the optional *\_ELSE* section are processed.

Conditional proposals are either simple or compound. A simple condition is encoded using a single *\_IF* or *\_ELSE\_IF* statement. Compound proposals require an initial *\_IF* or *\_ELSE\_IF* constructor followed by any number of *\_AND* and *\_OR* statements. As usual, *\_AND* has precedence over *\_OR*; hence:

w OR x AND y OR z

is interpreted as:

w OR (x AND y) OR z.

Parentheses are not allowed.

Conditional proposals encoded on *\_IF*, *\_AND*, *\_OR*, and *\_ELSE\_IF* statements are expressed as *VALUE="{Field-Name}.OP.String"*. 'Field-Name' is the name of an *<INPUT>*, *<SELECT>*, or *<TEXTAREA>* field whose content is to be compared, character by character, with the literal 'String'. The logical operator '.OP.', which is inserted tightly between its arguments (in other words, without spaces), defines what relationship must exist between 'Field-Name' and 'String' for the condition to be true. The operator codes available are listed below.

- .LT. Less than
- .LE. Less than or equal
- .EQ. Equal
- .GE. Greater than or equal
- .GT. Greater than
- .NE. Not equal.

When multiple occurrences of 'Field-Name' exist, the condition is true if and only if all instances match the condition. In the following example, an IF block structure is used to generate the most appropriate letter heading for each type of user:

```
<INPUT TYPE=HIDDEN NAME=_IF VALUE="{sex}.EQ.male">  
<INPUT TYPE=HIDDEN NAME=_AND VALUE="{Status}.EQ.single">
```



```
<INPUT TYPE=HIDDEN NAME=_WRITE VALUE="Dear Mr {Name},\n\n">
<INPUT TYPE=HIDDEN NAME=_ELSE_IF VALUE="{sex}.EQ.female">
<INPUT TYPE=HIDDEN NAME=_AND VALUE="{Status}.EQ.single">
<INPUT TYPE=HIDDEN NAME=_WRITE VALUE="Dear Mrs {Name},\n\n">
<INPUT TYPE=HIDDEN NAME=_ELSE_IF VALUE="{Status}.EQ.married">
<INPUT TYPE=HIDDEN NAME=_WRITE VALUE="Dear Mr and Mrs {Name},\n\n">
<INPUT TYPE=HIDDEN NAME=_ELSE>
<INPUT TYPE=HIDDEN NAME=_WRITE VALUE="To whom it may concern,\n\n">
<INPUT TYPE=HIDDEN NAME=_END_IF>
```

The code for this utility appears (in its entirety) in next month's issue of *AIX Update*.

---

*Pierre Croisetiére*  
*System Analyst (Canada)*

© Xephon 2000

---

## Building freeware and shareware

Not all freeware and shareware, such as that found on Groupe Bull's archive (<http://www-frec.bull.com/docs/download.htm>), is **smit**-installable. Often, you are required to download the software, unpack it, read the documentation, compile the source code, install the executable, and then ... hopefully ... be able to use it!

### DOWNLOADING

Often the software needs to be downloaded from an FTP site, such as UCLA's download site (<ftp://aixpdslib.seas.ucla.edu/>). In days gone by, you used the **ftp** command to initiate an anonymous FTP connection. Even if you didn't have an account with the freeware's host, it would let you download files. This meant that you used the user name *anonymous* and your e-mail address in place of the password. The file to be downloaded would probably be a binary file, so binary mode would need to be set, as in the following example:

```
$ cd /tmp
$ ftp aixpdslib.seas.ucla.edu
Name (aixpdslib.seas.ucla.edu:klauser): anonymous
Password: <your e-mail address>
```

```
ftp> cd pub
ftp> binary
ftp> get <file to be downloaded>
ftp> quit
$ ls
<downloaded file>
$
```

Today things are much more straightforward – you just use your favourite browser and the full FTP address of the file, including the protocol identifier ‘ftp://’.

In either case, you need to have enough disk space for the downloaded file(s). For that matter, you need to take into account that uncompressed files will need even more space.

## UNPACKING

The downloaded file will often be in a compressed format. This not only uses less disk space on the freeware’s host system, it also means faster downloading, as fewer bytes are transferred from their system to yours. The ending of the file name tells you which compression tool (or tools) was (or were) used.

‘.Z’ means the standard Unix **compress** command was used. Use **uncompress** to uncompress the file. More likely you will see ‘.gz’, ‘.tgz’, or ‘.tar.gz’, which means that the **gzip** program was used. Regrettably, **gzip** (and its companion **gunzip**) do not belong to AIX’s command suite. This GNU tool is found at most FTP sites, including the above-mentioned <ftp://aixpdslib.seas.ucla.edu/>.

To uncompress a ‘gzipped’ file, simply use **gunzip**. For example:

```
$ gunzip filename.gz
```

In this example, this creates a new file with the base name as the file name. Typically, the base file is a bundle that was assembled with the **tar** utility, so its name ends in ‘.tar’. In fact, the ‘.tgz’ extension is a shorthand for ‘.tar.gz’. The ‘.tar’ extension is not necessary, but is commonly used.

To extract all the files stored in a **tar** archive, use the **tar** command:

```
$ tar -xvf filename.tar
```

This is likely to create many files as it unravels the bundle. You will usually see a subdirectory with the software name and version number. Change to this subdirectory and look for a *README*, *README.TXT*, or *README.FIRST* file.

## PRELIMINARY DOCUMENTATION

The *README* file should tell you a bit about the package, what other packages it requires, and how to build it. Checking the *README* files can help you learn about the techniques used by others in putting together their packages, techniques you may find handy in your own work.

Some packages include a file named *INSTALL*. This file, as you would expect, contains information on how to build and install the package. Follow the instructions.

## COMPILING

Free software has been around for years, so there are some standards for building packages. The key problem is that software written for a variety of systems needs to handle platform differences. For example, the threading libraries on Sun's Solaris differ from those on AIX. The 'Holy Grail' of the free software movement has been to find a way to automatically detect and handle these platform differences. Of course, if everyone just adopted AIX or Linux, the issue would go away! (Alas, this argument is also used by Microsoft....)

One of the most common ways to deal with platform differences is to use a 'configure' script. This checks for system dependencies and outputs a *makefile* that is used by the **make** program to build the software. With a *configure* file, the basic commands for building the program are:

```
$ ./configure
$ make
$ make install
```

The **make** command should build the program code, compile, link, and perform other necessary steps. The last step should be to copy the

program files that were built to their proper locations. You will typically need to be a super-user to install files.

Older software that uses the X Window system for graphics often comes with a file named *Imakefile*. Like the *configure* script, this file holds a set of rules for building a platform-specific *makefile* that can be used by **make**. If you see an *Imakefile*, the basic commands to perform a build are:

```
$ xmkmf
$ make
$ make install
```

In this case, the **xmkmf** script runs the **imake** command that generates a *makefile* that's used by **make**.

With these commands, you can build most software packages. For more on this, check the on-line documentation on **make**, **imake**, and the GNU 'configure' system.

## INSTALLATION

The command **make install** will usually install freeware and shareware in the directory */usr/local* and its subdirectories. As I regard */usr* as an AIX operating system directory belonging to *rootvg*, I have it soft-linked to my data volume group:

```
$ ln -s /data/UNIX/usr/local/AIX/usr/local
```

## USE IT

That's it! Use your new software and enjoy it!

---

*Werner Klauser*  
*Klauser Informatik (Switzerland)*

© Xephon 2000

---

## **New features of RS/6000 hardware and software**

This article describes the various additions and enhancements to RS/6000 hardware and related software announced by IBM on 9 May, 2000. This is a very significant announcement, containing systems that replace current low to mid-range RS/6000 servers. The servers announced, which have an '80' in their model names, are based on the same RS64-III copper-based microprocessor that's used in the current high-end Model 80 server. The new models significantly improve both the performance and 'RAS' (Reliability, Availability, Serviceability) of the models they replace.

### **RS/6000 ENTERPRISE SERVER F80**

The desktide Enterprise Server Model F80 replaces the Model F50, a 32-bit server that is similar in design to the F80. It can be configured with one to six RS64-III processors mounted on one to three processor cards. The 1-, 2-, and 4-way machines use the 450 MHz processor, while the 6-way system uses the 500 MHz version of the CPU. The processor is configured with a 128 KB L1 data cache and 128 KB L1 instruction cache, plus a 2 MB L2 cache for the 1-way version and a 4 MB L2 cache for the 2-, 4-, and 6-way versions.

The processors are the same or faster versions of the 450 MHz processors used in the S80. The system can address up to 16 GB of SDRAM using two memory cards that take 128 MB, 256 MB, and 512 MB memory DIMMs. The DIMMs must be installed in units of four. If the maximum 16 GB of memory is expected to be needed it makes sense to configure the system with 512 MB DIMMs so as to avoid discarding the original ones in future. For 1-way systems, eight DIMM positions are available on the single processor card that can accommodate up to 4 GB of RAM. Two 128 MB DIMMs are required for a 256 MB entry-level system.

A special memory feature is 'chip-kill' protection, which is common to all F, H, M, and S80 models. Memory that supports this feature scatters bits across four 72-bit ECC words, enabling the identification

and isolation of a failed memory chip until the customer decides to replace it. Without this function, multiple bit errors in a single ECC word would cause a system 'check stop'. The combined system bandwidth of this model, which uses two independent system buses, is 4 GB per second.

The F80 has 12 media bays organized in two 'six-pack' enclosures similar to the ones used in Models F50, H50, H70, and S80. The six-packs can house SCSI or SSA hot-swappable disks with maximum capacity of 18.2 GB, providing a total internal storage capacity of 218.4 GB. Two additional bays are available for optional boot disks. A CD-ROM drive and a 1.44 MB 3.5-inch diskette drive are included, as is an additional bay that can be used to install tape drive. The computer has an integrated 10/100 Ethernet adapter as well as two integrated Ultra2 SCSI adapters (internal and external). Ten hot-plug PCI slots are available for 32-bit and 64-bit I/O, graphics, and communication cards. Six PCI slots operate at 66 MHz with 3.3 volt power and a transfer rate of 528 MB per second and the remaining four operate at 33 MHz with 5 volt power and a transfer rate of 132 MB per second. The system has four serial and one parallel port, as well as a keyboard and mouse ports.

The power supply features 'autoranging' of either 110-127 volts AC or 200-240 volts AC. The system can be fitted with optional redundant power supplies and cooling fans. A service processor, which is included as a standard, provides the following functions:

- Environmental monitoring and alerting of AC/DC voltage, fan speed, and temperature.
- Early power off warning and error log analysis and alert.
- Automatic dial-out calls for system operators or IBM personnel.
- Automatic reboot after power loss, hardware checkstop, machine check interrupts, or operating system failure.

The system also supports 'Dynamic CPU De-allocation', a feature that enables AIX to isolate a faulty processor and automatically migrate jobs that are using it, thus allowing the system to take the processor off-line for maintenance.

A field upgrade from Model F50 is available that allows the re-use of the following components:

- Memory DIMMs (including 32 MB and 64 MB DIMMs in units of four).
- Hard disks (with carrier change).
- Most PCI adapters.

The F80 does not support ISA slots, which means that ISA adapters installed in the F50 cannot be moved to the upgraded machine. The upgraded F80 chassis has the same serial number as the replaced F50 machine, allowing the re-use of existing software licenses and continued financial depreciation of the system.

The following table shows the performance of the F80 compared with that of the F50 using the relative OLTP performance estimate for commercial processing.

	F50	F80	Improvement
1-way	10.0	23	130%
2-way	17.9	50	160%
4-way	32.8	87.7	167%
6-way	N/A	111.9	241% (compared with 4-way F50)

#### RS/6000 ENTERPRISE SERVER H80

The Enterprise Server Model H80 is a rack-mounted system that replaces (and is similar to) both the 32-bit H50 and 64-bit H70 servers. The CPU and memory of this machine is identical to that of the F80 and is packaged in a 'CEC drawer' five EIA units (5U) high. The H80 CEC drawer has two redundant hot-swappable power supplies and two redundant hot-swappable cooling fans.

The machine's I/O subsystem is also packaged in a primary I/O drawer 5U high. It contains a CD-ROM drive and a 1.44 MB 3.5-inch diskette drive, plus an additional bay that can be used for a tape drive. The drawer also contains an integrated 10/100 Ethernet adapter and two integrated Ultra2 SCSI adapters (internal and external). Fourteen

hot-pluggable PCI slots are available for 32- and 64-bit I/O, graphics, and comms cards comprising ten 3.3 V, 66 MHz, 528 MB per second slots and four 33MHz, 5 V, 132 MB per second slots. The drawer has four serial and one parallel port, plus keyboard and mouse ports. The power supply supports 110-127 volts AC or -48 volts DC. Both CEC and I/O drawer can be fitted with optional redundant power supplies if an AC power supply is selected. Redundant power supplies are included when the DC power option is chosen. The system supports both a service processor and Dynamic CPU De-allocation. A second I/O drawer is available as a PRPQ option. The CEC and I/O drawers are connected using the same Remote IO cables as the Model S80.

Field upgrades from both Model H50 and H70 are available, allowing the re-use of memory DIMMs (including 32 MB and 64 MB DIMMs in units of four), internal hard disks (only two SCSI to Primary I/O drawer), external disk drawers, system rack, and most PCI adapters. Like the F80, the H80 doesn't support ISA cards. As with the F80 upgrade, the new H80 chassis has the same serial number as the replaced machine.

The following table shows the performance of the F80 compared with that of the H70 using the relative OLTP performance estimate for commercial processing.

	H70	H80	Improvement
1-way	16.7	23	37%
2-way	31.9	50	57%
4-way	57.1	87.7	53%
6-way	N/A	111.9	96% (compared with 4-way H70)

#### RS/6000 ENTERPRISE SERVER M80

The rack-mounted Enterprise Server Model M80 replaces the current 64-bit Model S7A. It can be configured with two to eight RS64-III processors mounted on one to four processor cards. All configurations use the 500 MHz version of the RS64-III. The processor is configured with 128 KB L1 data cache and 128 KB L1 instruction cache plus 4 MB L2 cache. The system can be configured with up to 32 GB of SDRAM using two memory cards that accept 128 MB, 256 MB, and



512 MB memory DIMMs. The DIMMs must be installed in units of eight. If a maximum memory of 32 GB is eventually needed, it's a good idea to configure the system with 512 MB DIMMs in order to avoid discarding original DIMMs in future.

The CPU and memory is packaged in an 8U CEC drawer. Both the CEC and primary I/O drawer are fitted with two redundant hot-swappable power supplies and cooling fans. The system features an integrated system switch connecting the processors, memory, and I/O that has an aggregated transfer bandwidth of no less than 18 GB per second.

Up to four I/O drawers, identical to one used on the H80, can be attached to the system, providing a maximum of 56 PCI slots (16 32-bit and 40 64-bit). Bootable SCSI disks can replace two slots in the first I/O drawer.

The following table shows the performance of the M80 compared with that of the S7A using the relative OLTP performance estimate for commercial processing.

	S7A	M80	Improvement
4-way	46	100	117%
8-way	N/A	193.3	70% (compared with 12-way S7A)

## 19" RACK OPTIONS

Two new system racks, used to consolidate installation of RS/6000 systems and peripheral equipment, were also announced. The model T00 has a height of 36U (1.8 metres) and the model T42 has a height of 42U (2 metres). The previously available S00 rack, with a height of 32U, is still available. Up to three Model H80 units can be installed in a T00 Rack, while the Model T42 rack supports the installation of up to three Model M80s.

Both models include optional front doors, removable side panels, and 'ruggedized' components for earthquake protection. It is possible to join multiple racks into 'suites'. The Model T00 supports both AC and DC configurations, while the Model T42 supports DC only.

## RS/6000 HA-H80 AND HA-M80 CLUSTER SERVER SOLUTIONS

The HA-F80, HA-H80, and HA-M80 Cluster Server solutions are pre-packaged high availability systems that contain two servers plus a Model 7133 Serial Disk (two for HA-M80), which is installed in a single rack in the case of the H80 and M80. The systems are configured with redundant power supplies, dual boot disks, dual advanced serial RAID PCI SSA disk adapters, and dual LAN adapters. The disk subsystem contains four 9.1 GB, 18.2 GB, or 36 GB disks and dual data paths as well as a redundant power supply. The system runs under AIX 4.3.1 with HACMP providing high availability for applications by monitoring and detecting system hardware and software failures, including ones affecting the processor, network adapter, power supply, and applications. In case of a failure, HACMP starts pre-defined recovery actions to a designated back-up component or server. HACMP also facilitates the graceful recovery and reintegration of a failed server with an operational cluster.

The system is expandable and is designed for easy growth. It is also possible to add disks to the 7133 subsystem supplied. Pre-configured HACMP scripts for popular databases and applications, such as DB2, Oracle, Informix, BAAN, and SAP, are to be provided by IBM at its Web site.

### AIX 4.3.3 MAINTENANCE LEVEL 04/2000

All the newly announced models run only AIX 4.3.3 or later. A special maintenance level, announced in April this year, is now available to support the new models and fix numerous existing bugs. This maintenance level can be downloaded from the Internet using IBM's **fixdist** utility.

### AIX 4.3 BONUS PACK ENHANCEMENTS

The contents of AIX Bonus Pack CD, which ships with every copy of AIX 4.3, have been refreshed. The following products were updated to their latest releases:

- IBM AIX Developer Kit Java 2 Technology Edition Version 1.2.2, based on Sun's Java SDK 1.2.2 reference port, and including IBM enhancements, such as a just-in-time compiler and mixed mode interpreter, and a re-engineered JVM that passed all tests in the Java Compatibility Kit (JCK).
- Geodesic's Great Circle V4.0.6.1 (30-day 'try-and-buy'), which is a collection of C and C++ debugging libraries that link to object code and use garbage collection technology to find and eliminate memory leaks and errors in an application's code automatically. The new version includes enhanced components to test and analyse C and C++ applications for memory problems during development and run-time.
- SecureWay-SSL Version 3.3, which now uses the same certificate support services as AIX IP Security Version 4.3.3 and is 'Euro-ready'.
- Netscape Communicator 4.7, which provides support for Hebrew and Arabic HTML.

## REFERENCES

- 1 IBM announcement letters at <http://www.ibm.link.ibm.com>.

---

*Alex Polak*  
*System Engineer*  
*APS (Israel)*

© Xephon 2000

---

# AIX news

---

Bull has announced new Escala mid-range AIX servers. The EPC610 and EPC810 are rack-mounted six- and eight-way clusterable nodes. Each clustered system supports up to 32 nodes and can be mixed and matched with other node types within the EPC family. Both use 64-bit RS64-III copper chip technology.

The minimum configuration is two 450 MHz processors for the EPC610 and two 500 MHz processors for the EPC810. Other features include hot-pluggable PCI adapters and support for CPU de-allocation.

Out now, prices are available on request from the vendor.

*For further information contact:*  
Bull Information Systems, 300 Concord Road, Billerica, MA 01821, USA  
Tel: +1 978 294 6000  
Fax: +1 978 294 4908  
Web: <http://www.bull.com>

Bull Information Systems Ltd, Computer House, Great West Road, Brentford, Middlesex TW8 9DH, UK  
Tel: +44 181 568 9191  
Fax: +44 181 568 1581

\* \* \*

ParaSoft has announced CodeWizard 3.1, the latest version of its code analysis tool for C++, which now has additional rules for C developers and enhanced customization capabilities for the RuleWizard module. Originally developed for C++, the analysis tool enforces standards, providing

information about the size and complexity of code. It's designed to help assess and then correct the coding style, which is said to make code easier to debug.

Out now on AIX 3.4 and 4 (and a number of other versions of Unix), prices start at US\$1,000.

*For further information contact:*  
ParaSoft, 2031 South Myrtle Avenue, Monrovia, CA 91016, USA  
Tel: +1 888 305 0041  
Fax: +1 626 305 3036  
Web: <http://www.parasoft.com>

ParaSoft Ltd, Suite 144, 52 Upper Street, Islington Green, London N1 0QH, United Kingdom  
Tel: +44 171 288 6600  
Fax: +44 171 288 6602

\* \* \*

Veritas has announced plans to port and optimize its storage management software, including Volume Manager and File System, to AIX Monterey for PowerPC and Intel IA-64 systems. This is the first time the company has ported its product to AIX – it has supported most other Unix systems for some time.

*For further information contact:*  
Vision Software Tools Inc, 2101 Webster Street, 8th Floor, Oakland, CA 94612, USA  
Tel: +1 510 238 4100  
Fax: +1 510 238 4118  
Web: <http://www.vision-soft.com>

