This is the second in the series of summary text files of my Windows Aspect (Wasp) course, lessons from 5 to 8 (lessons 1 to 4 may found in WASPA.EXE here)....

---------- Lesson # 5 ----------

SETMAIL.WAS

In this section of our "course", we will begin work on SETMAIL.WAS, a script to get user input of data which we will need when we use the PCBMAIL.WAS script (which will appear in a later lesson <G>), and to store that information in our INI file.

This time, we're going do things a little differently than we have done in earlier lessons. Rather than constantly repeating lines of code segments as we "build" our script, and since I already have a script written to perform the tasks which we need to store data in the INI file for our PCBMail script, what we'll do is post the script in it's entirety, with comments (indicated in the normal script fashion, by a ";" preceding each line) inserted where appropriate to explain what is being done. You may, if you wish, cut and paste from these lessons (with or without comments included) in order to "create" the complete SETMAIL.WAS script....

```
;SETMAIL.WAS v.5.2a - 04/13/93 011:40 AM
;Copyright (c) 1993, Gregg Hommel, All Rights Reserved

;SETMAIL.WAS is a Windows Aspect script for use with ProComm Plus for Windows 1.01. It
is ;used to set various parameters into an INI file. PCBMail can then read these
parameters ;to set options when it is run.

integer flag = 0, icondx = 5, icondx2 = 41, ndx = 0, rb1 = 1, source = 0
string board, defconf, dldir, door, iconvar = "UserID"
string iconvar2 = "Password", ini = "PCBMAIL.INI", maildir
string name = "(None)", pcblist = "(None)", pwfile = $PWTASKPATH, pword
string ren_def, user

;the above list of global variable declarations shows a personal preference on my
part... ;they are declared alphabetically. I have found that this makes it far easier to
add a
;variable later, without being concerned about duplicating a previously used one.
;Furthermore, and I have no proof of this, other than my opinion, but I have found that
;my scripts seem to run "better" with my variables declared in alphabetical order. It
;also points out a fact not clearly stated in the manuals... when a global variable
is ;declared, a default value can be assigned to it then. If the default value of
the ;variable is a PCP/Win internal system variable, this too can be assigned when
the ;variable is declared.

proc main
  integer button, charval, dlgstatus, lendx
  string del_set, pword2, save_set, user2, wrong

;In our dialog box later, we will need to have a global string which is the name and
path ;of the PCP/Win executable (which is where we will find the icons that we'll use
<G>). We ;already have the variable pwfile assigned the name of the default path for the
PCP/Win ;directory, i.e. $PWTASKPATH, so now we just use the ADDFILENAME command to add
PW.EXE to ;that path.

  addfilename pwfile "PW.EXE"

; In many cases, a script may be performing some function or task, but show
nothing ;visible on the screen to indicate this to a user of that script. The following
dialog box ;is placed here simply to tell a user that the script is indeed running. You
may find it ;necessary to insert dialogs of this type in your scripts, so as to remind
yourself that, ;although nothing appears to be happening on the screen, in reality, the
script is doing ;things.

  dialogbox 100 75 200 30 13
```

```
    text 10 10 180 10 center "PCBMAIL setup is scanning for boards now."
  enddialog

;It would be quite simple to create a string listing the names of the systems which
you ;call, to be used in our combobox later. However, once again, we are trying to create
a ;"generic" script which can be used by anyone. When doing so, we have to remember that
the ;user's dialing directory might change from time to time. To account for this, and to
make ;the script as "generic" as possible, we are going to read each entry in the
dialing ;directory, and if the user has set the script for that entry as PCBMAIL, we will
get the ;name of the entry, and add it to our string variable, pcblist, which will be used
in our ;combobox later. To do this, we use the internal "system" variables, $DIALCOUNT
(the ;number of entries in the currently active dialing directory), $D_SCRIPT (the entry
in the ;last accessed dialing directory entry which lists the script to use with it), and
$D_NAME ;(the name used in the dialing directory for the last accessed entry).

  for ndx = 1 upto $DIALCOUNT
    set dialdir access ndx
    if strcmpi $D_SCRIPT "pcbmail"
      strcat pcblist ","
      strcat pcblist $D_NAME
    endif
  endfor

;SETMAIL uses a section on the INI for board "(None)" to show sample settings in
the ;dialog box. However, if this is the first time that SETMAIL has been run, or if the
INI ;has been deleted, there will be no section, labelled [(None)]. The easiest way
to ;determine if a section exists in an INI file, is to read a description that definitely
;has an entry, if that section exists. If it is a null string (i.e. if there is no
entry ;there, then you can safely assume that the section itself doesn't exist. To this
end, we ;read a description which we know should exist in [(None)], and if we get a null
string, ;we assume that the section doesn't exist, and we "create" it.

;Note here a couple of points... if a named INI file does not exist, the first
PROFILEWR ;command issued to that INI name will create the INI file. Second, if a section
in a given ;INI file does not exist, the first PROFILEWR command issued which writes to
that section ;will also automatically add the appropriate section header. This makes using
INI files ;quite easy, as you do not need to create them, or open them for reading or
writing, or ;any of the other "normal" file manipulation needs.

  profilerd ini name "board_id" board
  if null_str(board)     ; if there is no (None) entry for here
    board = "chanone"    ; set default values for it, and write
    door = "e.g. 15"     ; them to the INI file
    maildir = "X:\MAILDIR"
    defconf = "e.g. 2"
    write_ini()
  else                   ; otherwise
    read_ini()           ; read the default values for (None)
  endif
  pause 1
  destroydlg

;Once we have scanned the dialing directory for a list of entries which are pre-set to use
;the PCBMAIL script, we then check to see if there were any! If the pcblist
variable ;contains nothing but the entry (None) (put there by default), then there are no
PCBMAIL ;systems in this directory, and we so inform the user, and then shut down this
script.

  if strcmpi pcblist "(None)"
    errormsg "There are no PCBMAIL systems in the dialing directory!"
    exit
  endif
```

```
;this next set of code just may look a little familiar <G> This is the dialog box which we
;worked with in the previous lesson, now part of the script it was originally intended
for.

  dialogbox 80 45 245 186 13
     groupbox 10 4 224 39 shadow
     text  18 12 209 28 center "Use this dialog box to set the parameters which the\
PCBMAIL script needs to operate. The Join Conference? parameter is optional."
     text  10 51 81 8 right "Select a PCBoard system :"
     combobox 100 49 135 55 pcblist name sort
     text  10 71 60 8 left "QWK filename?"
     editbox 75 69 45 12 board 8
     text  125 70 69 8 left "Mail door # (name)?"
     editbox 195 68 40 13 door
     text  10 91 60 8 left "Mail directory?"
     editbox 75 89 50 12 maildir
     text  130 91 60 8 left "Join Conference?"
     editbox 195 89 40 12 defconf
     groupbox 10 110 225 30 "Rename mail options"
     radiobutton 20 122 100 10 "Standard format - QWA" rb1
     radiobutton 130 122 100 10 "Alternate format - with date" endgroup
     iconbutton 10 147 iconvar pwfile icondx
     iconbutton 40 147 iconvar2 pwfile icondx2
     pushbutton 76 153 40 14 "&Save" normal
     pushbutton 134 153 41 14 "&Delete" normal
     pushbutton 195 153 40 14 "E&xit" cancel
  enddialog

;if you recall our previous discussion of the identification integers assigned to
the ;various elements of a dialog box, and the table which we "created" during
that ;discussion, you will now see the first implementation of using those identification
;integers, in particular, the column labelled CONTROL ID. When the dialog above
first ;appears, the item "(None)" is the default selection in the combobox. We have read
the ;default "sample" values from the INI for (None), and want them displayed, however, we
do ;not want a user to edit these, since they are only sample values. As a result, what we
;want to do is to display those values but disable the various dialog box elements so that
;the user can't use them. Once he has selected a system from the combobox listing, we will
;re-enable various controls, but for now, we will use the CONTROL ID integers from
our ;DIALOG ITEM table to disable some of our dialog.

  ;disable the editboxes
  disable ctrl 230 233

  ;disable the iconbuttons
  disable ctrl 210 211

  ;disable the one group of radiobuttons
  disable ctrl 50

  ;disable the first two pushbuttons
  disable ctrl 10 11

;use the value of $DIALOG to see if anything in the dialog box has been modified, and
act ;accordingly

  dlgstatus = $DIALOG
  while dlgstatus != 1
     switch dlgstatus

;based upon the CONTROL ID entry in our table, case 170 indicates that the combobox
has ;been selected in the dialog box, and we thus, handle the selection of a name from
that ;drop down box combobox. One thing we have to remember... it is possible that a user
might ;first select a system from the list, and then later, re-select the (None) entry.
```

Our code ;has to handle this contingency, and disable controls again if *(None)* is the selected ;entry.

;Here also, we see the first use of the other column of entries in our DIALOG ITEM ;identification table, i.e. the UPDATEDLG VALUE column. What this does is quite simple... ;once you have changed, via the script, the values for any variables on screen in a dialog ;box, they are not changed on screen automatically. Your script has to issue an **UPDATEDLG** ;command, indicating which controls are to be updated on the screen. The integer value ;issued with the **UPDATEDLG** command is the sum of the individual integer UPDATEDLG VALUE ;entries from our table. There is one other "value" which can be used, that being a -1, ;which updates ALL controls in the dialog box, without concern for their individual or ;cumulative values.

```
        case 170
           if strcmpi name "(None)"
```

;if the user selects *(None)* in the combobox, disable the same controls as previously, read ;the default data from the INI file, and then update the **editbox** (128) and **radiobutton** (2) ;controls in the dialog

```
              disable ctrl 230 233
              disable ctrl 210 211
              disable ctrl 50
              disable ctrl 10 11
              read_ini()
              updatedlg 130
           else
```

;otherwise, enable all of those controls previously disabled, read the INI file data for ;the chosen system (if any data exists), and then check to make sure the INI defined mail ;directory (if there is one) is valid

```
              enable ctrl 230 233
              enable ctrl 210 211
              enable ctrl 50
              enable ctrl 10 11
              read_ini()
              updatedlg 130
              if not null_str(maildir)
                 source = 1
                 chk_dir()
              endif
           endif
        endcase
```

;**case 210** and **case 211** handle a press of either iconbutton in the dialog box. What is ;unique here is the use of the **SDLGINPUT** command. In Wasp, user defined dialog boxes are ;not the only ones available from a script. Wasp may also access "standard" dialogs. These ;are such things as standard file open and file save dialogs, and the one used here, a ;standard input dialog. There are two advantages to using this form of dialog... 1) they ;are simple to use, since they do not have to be "designed" by the scripter, and 2) they ;CAN and WILL appear over top of a user defined dialog box. In Wasp, there is only one ;user defined dialog box allowed on screen at any given time. By using standard dialogs, ;it is possible to get additional user input without destroying the original dialog box, ;and then recreating it when needed again later.
;Both procedures work the same way... bring up a standard input dialog, asking for a "new" ;value for the appropriate string. Check if the string exists, and if so, was it changed. ;If changed, save the new information.

```
        case 210
           user2=user
           redo:
           sdlginput "UserID" "Enter / Edit the UserID : " user DEFAULT
```

```
         if success
            if null_str(user)
               goto redo
            endif
            if not strcmpi user user2
               profilewr ini name "userID" user
            endif
         endif
       endcase
       case 211
         pword2=pword
         again:
         sdlginput "Password" "Enter / Edit the Password : " pword DEFAULT
         if success
            if null_str(pword)
               goto again
            endif
            if not strcmpi pword pword2
               profilewr ini name "pword" pword
            endif
       endif
       endcase
```

;case 10 takes care of saving the data set in the dialog box. We first make sure that all ;the data we need has been entered, then check to make sure that the mail directory ;entered is valid, save the data for the system, and even check the normal download ;directory for old mail packets (QW?) for this system, moving them to the mail directory ;if we find any.

```
      case 10
      if not (null_str(user) || null_str(pword) || null_str(board) ||  null_str(door)\ ||
null_str(maildir))
         strlen maildir lendx
         lendx--
         strpeek maildir lendx charval
         if charval == 92
            strdelete maildir lendx 1
         endif
         ren_def = $NULLSTR
         if rb1 == 2
            ren_def = "date"
         endif
         fetch dnldpath dldir
         if null_str(ren_def)
            same_dir()
            if flag
               flag = 0
               exitswitch
            endif
          endif
          source=2
          chk_dir()
          if flag
             flag = 0
             exitswitch
          endif
          strfmt save_set "Save configuration for %s?" name
          sdlgmsgbox "Confirming..." save_set QUESTION YESNO button 1
          if button == 6
             strupr maildir
             strupr board
             strupr defconf
             write_ini()
```

```
            is_old_mail()
            statmsg "Configuration for %s saved!" name
            updatedlg 130
         elseif button == 7
            read_ini()
            updatedlg 130
         endif
      else
         wrong=$NULLSTR
         if null_str(user)
            wrong=str_make(wrong, "UserID")
         endif
         if null_str(pword)
            wrong=str_make(wrong, "Password")
         endif
         if null_str(board)
            wrong=str_make(wrong, "Qmail packet name")
         endif
         if null_str(door)
            wrong=str_make(wrong, "Open door")
         endif
         if null_str(maildir)
            wrong=str_make(wrong, "Mail directory")
         endif
         errormsg "Blank Field(s) are not allowed for - %s ." wrong
      endif
      endcase

;case 11 performs a simple "delete" of a system from PCBMail, by removing all INI
file ;settings. The system is not really deleted, but all entries in the INI are set to
null. ;There is command in Wasp, when using an INI file format to actually delete entries
once ;made, other than via a text editor, so rather than bother with it, we simply set all
;entries for a given system to nothing (null) in order to delete it.

      case 11
         if not null_str(name)
            strfmt del_set "Delete configuration for %s?" name
            sdlgmsgbox "Confirming..." del_set STOP YESNO button 1
            if button == 6
               if not null_str(board)
                  board = $NULLSTR
                  door = $NULLSTR
                  maildir = $NULLSTR
                  defconf = $NULLSTR
                  ren_def = $NULLSTR
                  write_ini()
                  statmsg "Configuration for %s deleted!" name
                  updatedlg 130
               else
                  errormsg "%s is not configured!" name
               endif
            endif
         endif
       endcase
    endswitch
    dlgstatus = $DIALOG
  endwhile
endproc              ; end of main procedure
```

This is not the end of the script, merely the end of the main procedure, but once again we are getting rather lengthy for safe posting of a single "message". So, next.....

---------- Lesson # 6 ----------

OK, let's continue with the discussion of the **SETMAIL** script structure, by adding a series of procedures called from the main procedure to perform various tasks repetitively. We'll also write a couple of functions, like those we wrote in our early version of **PCBLOG**, so that we can make multiple tests of various conditions in an **IF/ELSE** construct, even when Wasp doesn't want us to. You've already seen the references to these procedures and functions in the last lesson, the main procedure of **SETMAIL.WAS**, so I won't explain that end of it. <GG>

To create the entire **SETMAIL.WAS** script, simply cut and paste the following code after the previous lesson's main procedure code.

```
; a simple procedure which reads the INI file for data which is stored there. Used several
;times, whenever data in MAIN needs updating.

proc read_ini
  profilerd ini name "board_ID" board
  profilerd ini name "userID" user
  profilerd ini name "pword" pword
  profilerd ini name "door_ID" door
  profilerd ini name "mail_dir" maildir
  profilerd ini name "def_conf" defconf
  profilerd ini name "rename_as" ren_def

;the dialog box for this script does not use a string variable for the method of
renaming ;a packet, but rather a radiobutton setting. This little bit of code sets a value
for that ;radiobutton based on whether the variable ren_def is null or not.

  rb1 = 1
  if not null_str(ren_def)
     rb1 = 2
  endif
endproc

;another simple little procedure, this time the reverse of above, i.e. it writes data
to ;the INI file

proc write_ini
  profilewr ini name "board_ID" board
  profilewr ini name "door_ID" door
  profilewr ini name "mail_dir" maildir
  profilewr ini name "def_conf" defconf
  profilewr ini name "rename_as" ren_def
endproc

;This function may seem strange, since it does exactly what nullstr does, however, it does
;serve a purpose. Windows Aspect only allows the testing of a single nullstr command in an
;"if" conditional line. By using this function to replace it, we can test multiple strings
;for nulls, making the if/else conditionals simpler and faster.

func null_str : integer
  strparm test_str
  integer result
  nullstr test_str result
  return result
endfunc

;all this function does is take two strings passed to it, and concatenate them. The
trick ;is that it adds a semi-colon and space to the resulting string, making the string
useable ;as a variable for our combobox in the main dialog
```

```
func str_make : string
  strparm parm1, parm2
  if not null_str(parm1)
     strcat parm1 "; "
  endif
  strcat parm1 parm2
  return parm1
endfunc

;Fairly straightforward.... if standard renaming is used, the mail directory and
default ;download directory can not be the same. This proc justs checks for that.

proc same_dir
  if strcmpi dldir maildir
     errormsg "If standard renaming is used, the mail and download directories must be\
different!"
     profilerd ini name "mail_dir" maildir
     updatedlg 130
     flag = 1
  endif
endproc

;Again, fairly straightforward... check to see if the mail directory set in the
dialog ;exists. If not, ask if the user wants it created.

proc chk_dir
  integer button
  string curr_dir, dirmsg
  getdir 0 curr_dir
  if not chdir maildir
     if source == 1
        mkdir maildir
     elseif source == 2
        strfmt dirmsg "The directory %s does not exist. Create it?" maildir
        sdlgmsgbox "Warning!" dirmsg EXCLAMATION YESNO button 1 BEEP
        if button == 6
           mkdir maildir
        elseif button == 7
           flag = 1
           profilerd ini name "mail_dir" maildir
           updatedlg 130
        endif
     endif
  endif
  chdir curr_dir
endproc

;This proc is real fun stuff! <G> After we have saved the information for a system,
check ;the download directory for old mail packets. If one (or more) is found, ask the
user what ;to do with them... either delete them or rename them according to the settings
in the ;dialog box. A whole bunch of steps just to do something apparently that simple,
huh?

proc is_old_mail
  integer button, count, test = 0, char, char2, max = 0, ltr, len, rltr
  string oldqwk, msg, newqwk, renqwk, oldfile, root, sdate
  strfmt oldqwk "%s\%s.QW?" dldir board
  if findfirst oldqwk
     strfmt msg "Old mail for %s has been found. Rename it? [NO] will delete the old\
mail." name
     sdlgmsgbox "Warning!" msg STOP YESNO button 1 BEEP
     if button == 6
```

```
;if we are renaming the mail that was found, we have to determine how we are going
to ;rename it. If we are using the default scheme (packet name with a changed
extension), ;things are simple. If we are using the alternate scheme (up to 4 characters
of the packet ;name, combined with 4 characters representing the date) then we have to
determine what ;root we will use for the renamed mail packets

        if null_str(ren_def)
           root = board
        else
           substr sdate $DATE 0 5
           strdelete sdate 2 1
           strfmt root "%s%s" board sdate
           strlen root len
           if len > 8
              len -= 8
              strdelete root 4 len
           endif
        endif

;silly though it may seem, we also check the mail directory to make sure that, if by
some ;chance there already is mail stored there, we don't overwrite it when renaming and
moving ;the "new" mail.

        strfmt newqwk "%s\%s.QW?" maildir root
        if findfirst newqwk
           max++
           while 1
              if findnext
                 max++
              else
                 exitwhile
              endif
           endwhile
        endif
        max--

;now we check to make sure that any packets found in the mail directory are
named ;properly, and in order. If we find a gap in the extension (since both renaming
sequences ;use a QW? type of extension) for these packets in the mail directory, we rename
them ;until they are in proper order.

        for count = 0 upto max
           char = 65 + count
           strfmt newqwk "%s\%s.qw%c" maildir root char
           if findfirst newqwk
              loopfor
           else
              for test upto 25
                 char2 = char + test
                 strfmt newqwk "%s\%s.qw%c" maildir root char2
                 if findfirst newqwk
                    strfmt renqwk "%s\%s.qw%c" maildir root char
                    rename newqwk renqwk
                    exitfor
                 endif
              endfor
           endif
        endfor

;and now, we rename and move the mail from the download directory to the mail directory.

        ltr = 65
        rltr = max + 65
```

```
        for count = 0 upto 35
           strfmt oldqwk "%s\%s.QW%c" dldir board ltr
           if isfile oldqwk
              rltr++
              strfmt renqwk "%s\%s.QW%c" maildir root rltr
              rename oldqwk renqwk
              if success
                 statmsg "%s moved as %s.QW%c" oldfile root rltr
              else
                 statmsg "%s not moved!" oldqwk
              endif
           endif
           if count == 25
              ltr = 47
           endif
           ltr++
        endfor
     elseif button == 7

;If we aren't going to keep the old mail which has been found, this procedure will
delete ;it.

        ltr = 65
        for count = 0 upto 35
           strfmt oldqwk "%s\%s.QW%c" dldir board ltr
           if isfile oldqwk
              delfile oldqwk
           endif
           if count == 25
              ltr = 47
           endif
           ltr++
        endfor
     endif
  endif
endproc
```

Before we continue with an examination of the PCBMAIL.WAS script which uses the information obtained through SETMAIL.WAS, I think it's approriate to take a small break from actual scripting (before we overload on code <G>). So next time, we're going to discuss some tricks and tips for writing a script, including some tips on organizing and logical structure, and why this kind of planning can assist you when writing a script. Then, we'll look at some tricks and tips which involve string manipulation under Wasp, and the use of time variables (BTW, did I mention that the Wasp SUSPEND UNTIL command does not work properly, and we need to develop work arounds for that fact?? <GG>)

---------- Lesson # 7 ----------

David Lecin did a rather nice job of discussing the logic and organizational skills necessary for script writing in his course on DOS Aspect, but not all of you have seen that course (I am posting this in three additional nets, along with those two where David posted his DOS course).

We won't discuss this too heavily, but I thought that it might be nice to take a break from the heavy-duty coding that we have been doing, and just relax the tone of things a bit.

Basically, I suppose that every programmer has his own idea of what makes up good programming practice. In my opinion, this good practice consists of two basic routines.... planning in advance what you want the script to accomplish, and roughly how it might do so, and writing the code you need in a modular fashion.

Let me explain.....

It is quite difficult to decide what a script (or any code) needs do next, when you, the author, have no idea what you want it

to do next. The code won't "get anywhere" if you, the programmer, have no idea where it is that it is expected to go.

Generally, before I begin work on any code, I attempt to write out, in English, what I want the code to accomplish, and where I want it to end up when it is done. This gives me a basic word picture of what I hope the code will do when it is finished. To relate this directly to Wasp, it also helps to force you to look at what is happening on the terminal with a more careful eye, as you attempt to follow what is happening in order to create that word picture.

But... the first rule is to start out simply. Don't get too fancy, and don't try to do too much with the first draft of a script. I prefer a modular approach to script writing, where one can add features and functions simply by adding a new procedure to a basic script. In this way, a first draft script can be kept quite simple, and then have other routines added to it as they are tested.

However, this does not preclude increasing the size of the main procedure. Often, once a procedure has been tested and found to work properly, if it is to be called only that one time, I will simplify the structure of the script by removing it from a separate procedure, and adding it to the main procedure where the sub-procedure was called from initially.

Basically, when I begin work on a new script, or upon a new procedure to be added to a script, I make sure of two things... 1) that I have lots of paper for writing the English, flow chart, and code information, in rough, and 2) that I have plenty of disk space available for various versions of the code being tested., not to mention backups of it, just in case! <GG>

I remember one case where I was away for the weekend, but brought a printout of the code I was working on, and a pad of paper to write modified code. My daughters complained bitterly that I must have destroyed three trees to write that relatively small bit of code, because I went through so much paper writing it. But this brings up another routine that I use frequently... rather than physically testing the code while on line, I often use diagrams, etc. to work out, on paper, what should happen, based on the code written, in order to test the logic of the script before actually compiling it, and running the code.

And that can be supremely important... the logic of the script, espceially when trying to track down a "bug" or mistaken action. One of the main benefits of using English "coding", and flow charting a script is that it tends to improve the logic of the script. Rather than bouncing here and there, from one procedure to another, all over the code, it becomes easier to write the various procedures and routines in a more logical fashion. This makes the script easier to follow later, when a problem develops, or you want to change some section of it. A logical arrangement of sub-procedures and routines within procedures makes it far easier to locate a section which you want to change.

To that end, I also attempt to use descriptive variable names, and procedure/function names, where ever possible. As example, in my PCB Freedom script, the procedure which does the physical dialing of a system, and manages the on line functions is called "proc dial_boards". The procedure which creates a dialog box to modify system settings while off line is called "proc edit_dlgs". The routine to add a new system to the configuration list is called "proc add_item".

I am sure that you can see how this might prove advantageous. On a little script like our PCBLOG.WAS created earlier, this is not of such importance, as the script is fairly short. But PCB Freedom is currently around 3,500 lines of Wasp code, and locating a particular section within that 3,500 lines can be quite difficult without some "sign posts". I use descriptive variable and procedure names, with a logical connection to what they are for, as my "sign posts".

I also use the search and replace feature of my editor to my advantage. Often times, a variable, or procedure may start life with a particular name, descriptive of it's purpose and place within the logic of the scipt. However, as the script takes on new features and functions, that descriptive name may no longer be valid, or useful. When this happens, "search and replace" allows for easy change of one name or variable in use, into another, which may be more informative under the current code in use.

There is a further feature of my editor which I use regularly, and one that I wonder how anyone who writes scripts can do without... that is the MDI interface of the Norton DeskTop Editor. Almost without fail, when working on a script, I will have more than one file open at a time. Sometimes it is both the WAS file and the LIB (which extension I use to indicate that it is an "INCLUDE" file for some WAS file) for it, other times it is my actual script and a file containing the code segment that I have been working on. With a good MDI interface, cutting and pasting from one file to another becomes easy. When I am not using Windows to work on my code, I use QEdit, which also allows multiple files open at the same time (right now, this file and FREEDOM.WAS are both open in QEdit, to make referencing procedures, etc. in Freedom easier for me <GG>)

The thing to remember is that there are no "set" rules for methods or procedures to write a good script. What works best for one code maven may be deadly to another. I know my old CompSci prof from many years back would probably have a fit about that comment, as he constantly emphasized following rules, etc. when coding, but, in the real world, I have found that those "rules" do not always work.

That prof used to always tell us to never use a GOTO label, but to always call a sub-routine instead. Theoretically, this may work, but in the real world, there are many occassions when you do not want the script to return to a given spot, but want it instead to branch off through a different set of code. GOTO works rather well for this, while calling a sub-routine can be tricky to do the same sort of thing. And since Wasp is not Fortran nor Cobol, and I don't run Procomm on a mainframe, I don't think that the rules he used to drill into us are applicable.

And there is one other thing that I find important about breaking the rules... you just might discover something that helps! While working on the early versions of GHOST BBS, Toby and I would swap beta code around (Toby lives around 100 km. from me... we did everything by modem, most often using early drafts of GHOST as the means to transfer modified code.), and more times than I care to remember, one or the other of us, when discussing changes by voice, would say "But you can't do that!"

The trick is that, sometimes, what conventional wisdom (the rules) says can't be done, just might be possible. But you will never discover these "huh?" code segments if you follow all of the "rules". In both Freedom and GHOST, there are several bits of code which, when I first wrote them, were discarded because examination on paper "proved" that I couldn't do that. But, when all else failed, or the conventional methods grew too cumbersome, I would invariably fall back on the "impossible" code, and generally, found that what appeared impossible on paper, worked quite well when compiled. <GG>

However, remember that there is a corollary to this "trick"... sometimes what appears, at first test, to work, really is "impossible", and can bite you when you least expect it. One pitfall to being a "ground breaker" is that sometimes, you find that, instead of "breaking new ground", you are over the edge of the cliff with an anvil for a parachute.

Never discount the "impossible"... fairly early in the beta of Freedom, I ran into a problem with some very strange responses being sent by the script, when the code was written to delete all characters from the string variable, resulting in a null string, and nothing being sent. When I discussed it with the folks from Datastorm, I was told that what I claimed to have happening was impossible, and that deleting one by one, the characters of a string HAD to result in a null string when the last character was deleted.

Further testing on my part, and very careful observation (and notes) of what was being sent when nothing should have been sent, showed me that, somehow, the string, once all characters were deleted, was being assigned a value which seemed to be the central six characters from the LAST string variable accessed before the current string variable had it's last character deleted (does this make sense?? <G>) The solution... when the last character was supposed to be deleted from the string, instead of actually deleting it, I began assigning the system variable $NULLSTR to the string variable. Now, impossible or not, the string variable actually was a null when I wanted and expected it to be one.

Datastorm still says the results that I experienced are impossible, but I can duplicate them any time I wish by restoring the original code segment to that function. I must agree with Datastorm in that, logically, the results are absolutely impossible, however, it did happen, on my system, and several beta sites.

In other words, never say never, and always suspect that what is not possible for a script to do, just may be possible. (and never turn your back on a "completed" and "fully tested" script.... they can be mean, vicious and downright despicable! <GG>)

I suppose that I have rambled enough for now. I just thought that a break from "code" might be nice here, and that telling you of some of my experiences with planning and writing scripts might help uyou establish some procedures for writing scripts which work for you....

---------- Lesson # 8 ----------

System Variables

We have already seen and used quite a few "system" variables, such as $PASSWORD, $D_NAME, etc. The uses for these pre-defined and reserved system variables in a script can be numerous. Some of them contain basic operations data for PCP/Win which can be invaluable in a script (such as $PWTASKPATH, the directory where PW.EXE, the main ProComm executable, is stored, and $ACTIONBAR, which returns the current status of the icon bar on screen), others contain information on hardware states (such as $DIALING and $FLOWSTATE), information on files ($FATTR and $FDATE), and still others hold data on different actions taken by scripts, allowing us to check their current status ($FILEXFER, $OBJECT and $DIALOG)

Be careful when using some of these, however... some of them do not hold a value for long and this can confuse a script.

As example....

```
while 1
  if $DIALOG == 50 || $DIALOG == 30
     do_something()
     destroydlg
     exitwhile
  endif
endwhile
```

Logically, this should check to see if the user has accessed either a radiobutton or update pushbutton in a dialog, and if they have, call a subroutine, destroy the dialog box when we return from that subroutine, and then exit the "perpetual loop" setup of the while/endwhile construct.

However, pressing the update pushbutton, in this case, would be rather disappointing... nothing would ever happen. Why? $DIALOG is a system variable which is CLEARED TO 0 AFTER IT IS ACCESSED! Thus, in the above code, we "access" $DIALOG, and evaluate the expression "30 == 50" (since we pressed an update pushbutton, the value of $DIALOG would be 30), which is false. So the IF then evaluates the second expression on the line, and also determines that it is false.

"Huh??", you say... "I selected the update button, so the value of $DIALOG is 30, and the expression should be true. You just said so above!" Well, "0 == 30" is false, is it not? And the value of $DIALOG is now 0, not 30, since our test for the first expression has already accessed $DIALOG, and Wasp has cleared it before we evaluate the second expression.

Watch out for this with the following variables...

$DIALOG - cleared to 0 after it is accessed

$FILEXFER - if the value of the variable is either 2 or 3 (a transfer completed either successfully, or unsuccessfully), it is reset to 0 after being accessed.

$MENU - cleared to 0 after it is accessed

$OBJECT - cleared to 0 after it is accessed

$PKRECV - cleared to 0 after it is accessed

$PKSEND - cleared to 0 after it is accessed

The easiest way to avoid this problem is to access the system variable only once, when assigning it's value to a user defined (global or local, although local works easiest) variable. Then, rather than checking the system variable, check the value stored in the user defined variable. As example, using the previous code....

```
while 1
  dlgstat = $DIALOG
  if dlgstat == 50 || dlgstat == 30
     do_something()
     destroydlg
     exitwhile
  endif
endwhile
```

This one would work as you would expect.

But there is one additional "type" of system variable which I have deliberately not mentioned yet, nor used in the scripts that we have been working with. These consist of pre-defined global variables for each type of variable possible in a Wasp script, i.e. S0, I0, F0, and L0. There are 10 of each type of variable allowed, i.e. S0 through S9, etc., but one should use these with caution.

These variables need not be defined in our script, and furthermore, are the only variables which can be "passed" from one script to another by using the EXECUTE command. They have one additional advantage, being the fact that they already exist in memory, and if you are running short of memory during a compile of a script, replacing declared global variables with one or more of these pre-defined system variables can often regain enough free memory to add a few more

lines of code (we'll discuss this further in a later "lesson")

There are, however, many drawbacks to using these variables....

1) Since they can be passed to a script called via the EXECUTE command, and the value they contain when the child script ends is passed back to the parent script, they are too easily inadvertently modified in such a way as to lose control of script actions based on their value. If you use a user defined global, it's value can not be "passed" to a child script, however, likewise, it's value can not be modified inadvertently by the child script.

2) Their names are not exactly descriptive, in keeping with our logic of variable names. Thus, six weeks from now, that script which was clear as a bell today, may be of absolutely no use to you, since you can't remember what the value of I3 is used for later in the script.

3) They suffer from the basic drawback of all global variables, i.e. they can be too easily modified in an obscure procedure, resulting in totally unexpected values elsewhere in the script, and totally unexpected, and often, inexplicable, actions based on those values.

In the end, there are times when using these predefined globals can be quite useful, and the only way to go. When GHOST BBS 2.00 is released, it will have the capability to shut itself down at a preset time, and run a secondary script. When it does so, it also will pass, to that secondary script, either an integer variable, or a string variable, or both. Of course, the secondary script will need be written to act on those passed variables (PCB Freedom 1.50 will act on the integer, using a 1 to tell it to dial the listed systems right away, and when done, shut down to return control to GHOST BBS. A 2 passed to Freedom will do the same thing, but will eliminate the "System Options" dialog box), but the method of passing the variables is, of course, through these predefined global system variables, and there is no other way of doing so.

Both GHOST BBS and PCB Freedom use other system variables for other reasons (mostly to save memory <G>). One method which I have used successfully to remind myself of which system variables are in use, and what they are being used for, is to add a comment section at the beginning of the script to remind me of this information. Something which you may want to do, if you use a system variable such as these in your scripts.

#DEFINE Macros

Defined macros have a multitude of uses in a Wasp script. In their simplest form, they represent a variable which may have a different value for different users of a script, but need be set only once. As example, suppose that your script assumes that a user will have the same "name" on all systems which he calls. If you were to use a line like this :

```
#DEFINE USERID "Gregg Hommel^M"  ; replace with your name and compile
```

at the beginning of your WAS file, the user could replace the quoted text with his name, and then compile the script. In the WAS, any appearance of USERID would be replaced by the string which is defined in the line above.

This, of course, is perhaps one of the simplest uses of a defined macro. Once common one is to make reading your source code more understandable. As example, suppose that, throughout your script, you tested quite a few variables for a true or false state (false being a value of 0, true being a value of 1, for an integer variable). Testing for this is quite simple, and can be written as follows :

```
if my_var == 1              ; if my_var is set to true
```

or even

```
if my_var
```

however, six months from now, would it not be simpler to look at this line?

```
if my_var == TRUE
```

In my opinion, this is much easier to read, and understand. To accomplish a test such as this requires two statements in your script prior to the main procedure (i.e. in the section for declaration of global variables) :

```
#DEFINE FALSE 0
```

#DEFINE TRUE 1

From then on in the script, every occurence of the macro FALSE would be replaced at compile time, with a 0, and every occurence of TRUE with a 1. The tests would be valid, but the WAS file would be far more readable.

We use a similar "trick" in GHOST BBS. Althought they are not used all that often (actually only about 3 times in two procedures), the information stored for mail in GHOST includes data regarding it's "public" state, i.e. if it is private, received, etc. These are integer values. However, when reading the source code, it is difficult to remember that 0 is public, 1 private, etc. etc. So, we used #DEFINE macros for those values, and now it is simple. Reading code such as

flag = PRIVATE + NEWMAIL   or
flag = PUBLIC + DELETED

is much easier than remebering what

flag = 3   or
flag = 4

means <GG>.

A further use for defined macros is to replace frequently used code strings. An example of this is from the coding of GHOST BBS. In order to improve the speed of screen redraws, one thing that Toby and I did was change from scrolling screens to redrrawn screens. To do this locally in the script is simple.. the Wasp CLEAR command clears a local screen, resulting in a redraw of the next screen. However, to accomplish the same procedure on a remote user's machine is a touch more complicated. What this requires is the transmission to the user's machine of the ANSI clear screen command (^L).

In order to increase the speed of displays in GHOST, this command must be transmitted frequently (I think the last time I checked, it appeared some 250 times in the code), but it is ALWAYS the same command. To simplify the code for GHOST, and so that we would not have to remember that ANSI code each time we needed it, we placed this line at the beginning of the source code for GHOST BBS...

#DEFINE CLS   transmit "^L^M"   ; send ANSI clear screen to remote

Now, in the source code for GHOST, instead of having to remember what the ANSI code for a transmitted clear screen is, we merely remember the DOS level command for it, CLS, and put that in the code (all 250 times <G>)

There is another use for #DEFINE macros which I like (and I am sure that you will think of many other ways to use them <g>), although I have yet to use it a great deal. This is conditional compilation....

Although this is not a prime example of a use for this, it will serve to point out how it can be used. Let's imagine that you have written a script to log onto many different BBS. In one way, it is similar to my PCB Freedom script.... the default WAX file includes support for encryption of things like passwords in the INI file. The procedures for that encryption are proc encrypt and proc uncrypt, which are stored in a library, SECURITY.LIB, which, for obvious reasons you can't give out to users. Equally as obviously, without that LIB file, they can't compile the source code for the rest of the routines themselves.

However, many of the people using your script say that they don't need or want that encryption storage, and are quite happy if their passwords are stored in clear text. Do you write a separate script for them to use if that is the case?

I suppose you could, releasing two versions of the file, one with and the other without encryption. But there is another way to do it....

Suppose at the beginning of your source code (the portion which you can release to the public), you put something like :

#DEFINE USESEC "yes"

In this case, the actual contents of the macro don't really matter. What does matter is the definition itself!

Later in your script, the following code exists.....

#IFDEF USESEC

```
  #INCLUDE "SECURITY.LIB"
#ENDIF
```

and further on, when you read the password from the INI, this code is there :

```
#IFDEF USESEC
  pword = uncrypt(pword)
#ENDIF
```

What happens here?? Well, basically, you are telling the script to function differently at compile time, and run time, IF a certain macro is defined (what it is defined as is immaterial.. only that it is or is not defined). In other words, at compile time, IF the macro USESEC is defined, include the SECURITY.LIB file during compilation. And, when running, if the macro USESEC is defined, decrypt the stored password.

So, how does this "help" your users?? Well, you can issue a WAX file based on the macro being defined. It will use your security procedures based on those on your system when you compiled it. But, if a user does not want to use these "secure" passwords, he can take your "safe" source code (i.e. the WAS without the LIB file), comment out the one line "#DEFINE USESEC "yes"" and compile the result. It will not look for the SECURITY.LIB file, and will not decrypt the password when it reads it.

I know that this is not a very good example, but I think you get the drift... the script comiled "conditionally", i.e. if a macro is defined, it compiles one way, if the macro is not defined, it compiles another way.