Welcome to the Advanced System Manager's Console. (ASMC)

****
TABLE OF CONTENTS
****

Please note! This document contains descriptive information about software which is not normally included in the ASMC Package. It can be provided on request should you need it.

*. This package contains software I've put together over many years to assist in managing VMS and in developing so called "System Programs". Much is my own and much is based on public domain software written by Brian Schenkenberger of Tmeis and Hunter Goatly formerly of the University of Kentucky. I owe a debt of gratitude to them, particularly Brian who has helped me numerous times to understand what the software was doing and getting it running.

I'm also indebted to Ralf van Diesen of Advanced Virtualization Technologies in the Netherlands. He has most generously given me his software and helped me set it up so that I could run a two node AXP cluster and write these programs to work on it.

Some of this software has been improved slightly by making it a bit easier to use. For example, many programs require the PID of a process to work. I've modified things so that the terminal name will also suffice. Much of the "internal workings" of these programs have common sections and this has been moved to a shared image saving memory and disk space while improving reliability. I have also re-written the programs using MACROS that make MACRO32 and MACRO64 fully structured languages thus significantly improving their readability and maintainability. This is controversial, but I feel strongly about it.

This work is a labor of love and represents many years of my life. Use it in

good faith.

Rick Marsh Ph.D.
Tulasi Software Systems
Florence, Oregon
rick@rickmarsh.com
2023

LEGAL DISCLAIMER:
****

This software is provided "AS IS" and is supplied for informational purpose only. No warranty is expressed or implied and no liability can be accepted for any actions or circumstances incurred from the use of this software or from the information contained herein. The author makes no claim as to the suitability or fitness of the software or information contain herein for a particular purpose.

Permission is hereby granted *ONLY* for the "not-for-profit" redistribution of this software provided that ALL SOURCE and/or OBJECT CODE remains intact and ALL COPYRIGHT NOTICES remain intact from its original distribution.

(!) NO TITLE TO AND/OR OWNERSHIP OF THIS SOFTWARE IS HEREBY TRANSFERRED. (!)
****

(!) FROM THE AUTHOR (!)
****
Tulasi Software Systems is a one man shop. I'm running emulation and operating system software, some of which has been donated by friends who share in the opinion that OpenVMS is a highly significant operating system which should be advanced and maintained into the future.

The software described in this document and provided free of charge in various forms is a labor of love. I am running a two node cluster using AVT's Alpha emulation software, running VMS 8.4 and have tested these programs to the extent possible. I have neither the equipment nor the manpower to do extensive testing under high stress situations.

Most of this software runs in a highly privileged mode, high IPL and some loads code into the operating system. If you run any of this software and something evil happens, you have been warned.

One special program that can be used to run most of what will be discussed in this document, is MENU.EXE written in MACRO32 and driven (user controlled) by a data file whose file specification is passed on the command line. Thus the menu program is generalized for others to use. It is also in color. We use a terminal emulator called Putty. It is available on the net for free.

The MENU program will allow the designer to either run an image or execute a DCL script. Such a script may set up an environment to facilitate the running of an image. In any case, the person who has written the controlling data file is responsible for choosing.

MENU is especially designed to take advantage of SYSMAN for use in clusters. It can produce scripts that run SYSMAN and perform certain functions on a selection of nodes, all the nodes in a cluster or just on your node. You can even use MENU to run your own specific command procedure or program on a cluster wide scale that is not included in the menu data file.

If you are running MENU on a cluster, you will notice that some menu entries are followed by a red asterisk. These can be run on other nodes from the one you are logged into. Some of these have blue descriptions rather than green indicating they are cluster wide by their nature. Those entries which are in blue do not use SYSMAN. Green entries followed by an asterisk, do.

Regardless of whether you are on a cluster, an entry who's description is white indicates that it must be run at the same level (not as a sub-process) as MENU and thus will terminate MENU when selected.

Note! the logical EXED must point to the location of the executable images, the logical COMD must point to the location of the scripts and the logical TEMPD must point to a location where we can write a whole bunch of temporary files.

The data file that drives MENU is constructed as described in the file, DATAD:MENU.DAT. In addition, we refer to ADVMGR and R$ as rooted logicals interchangeably. Both are defined for you.

Disclaimer: In all cases, the reader is referred to the standard DEC/COMPAQ/HP/VMSI documentation for authoritative reference.

WARNING!!!!!

Running programs using the menu while logged in into a node over DECnet will guarantee crashing that node. It's the way menu is constructed and we have no intentions of changing it. If logged in through DECnet, I.e. "SET HOST", define symbols to run my programs. The exception is that you may use the selection 1. All Security by running "@comd:rsecurity". It is not a program rather it is a script.

Many procedures ask for argument(s). Though brief, the request is designed to be descriptive. In most cases, only one argument is needed. In those instances where more than one is required, there is a code. (2sp) means two arguments are needed and they should be separated by a space. (2,) means they should be comma separated. For requests for pid or terminal name, get this information from the display of "Show Processes". Pid is hexadecimal and terminal name should leave out the ':'.

PAGE FAULTS!!!

MARLIB is a shared image that contains a great deal of code including some very specialized subroutines that do address relocation and get S0 space code addresses. These routines are called at high IPL. You have to be sure that your program will not page fault during these calls or the system will crash.

To do this, we must do more than make a call to lock our program in memory. It turns out we must also be sure that the shared image, MARLIB, is mapped into your work space and considered part of your program. The macro ILOCK does this for you by making a call to SLEEP for zero seconds just prior to locking your image into memory. Use it.

****
1. The Advanced Systems Manager's Menu
****

#----------
1) "All Security" - One of the prime responsibilities of a systems manager is security. Please read the security manual. There are eleven categories of objects that allow security related settings. This script will be of

considerable assistance in deasssigning permissions, creating identifiers, granting identifiers and assigning them to objects to control access to them in complex ways.

You will run the AUTHORIZE utility through this script as well as the ACL Editor. In addition, you may set/remove the current privileges for any process on the system, including your own, regardless of the authorized privileges assigned to that user. This assignment will last for the duration of the process. You have control of the process priority through this menu and can encrypt/decrypt any file.

Additional information and suggestions are available from that menu.

#--
2) "Show Processes" - This selection runs a program that produces output somewhat similar to "SHOW SYSTEM" but is more brief and includes the terminal name for those processes that have one connected.

#--
3) "Shutdown" gives you options including: Cluster, Power Off and Reboot.

#--
4) "Roll you own". Trial and error. Uses SYSMAN.

#--
5) "Suspend/Resume a Process" - Suspend/Resume will either suspend a process or resume it, depending on its present state.

#--
6) "Kill an RWASTed Process" - Special Process Delete is more than the DCL "STOP PROCESS_NAME" or "STOP/ID=XXXXX". STOP will not work if the chosen process is in resource wait state. However, this method sneaks into the process' PCB table and resets the resource wait bits prior to deleting the process. This one runs in KERNEL mode.

#--
7) "Show Open Files On a Device" = This will show you who has what file(s) open on a drive.

#--
8) "Show Other's Default Directory" - Other's Default Directory will use an executive image to determine the process' default device and directory. Runs in kernel mode and uses an execulet.

#--
9) "Show Others Commands" will print out the full command recall buffer of the other process. Runs in kernel mode and uses an execulet. It is useful for helping others.

#--
10) "Start EVE" - Start the eve editor as a sub-process.

#--
11) "Find the Lock Holder" - Get lock holder takes the name of an IASM file, the specific key for that file's record which is locked, and determines the process name of the process which holds the lock. This is useful for finding out who got the lock and went fishing. You can free the record by deleting the holder's process.

#---
12) "Clear a Device Error Count" = Does just that plus memory and cpu error count. Especially useful for device driver development.

#---
13) "Show Disk Freespace" - DEC/COMPAQ/HP never thought that a knowledge of the percentage of a disk that was available (free) was of any interest to a system manager. We do. This script lists all mounted disks on the system and shows the percent free space on each one.

#---
14) "Monitor" - Monitor is often run interactively. It displays a bunch of information in an easily read graphical or list display in real time. In many cases, you need to monitor for information over a long period of time. The script produces a sub-process (/NOWAIT) that stops when you tell it to stop, creates a file called ADVMGR:[TMP]MON.LOG that you can look at and notifies you when it's complete. Don't log off when it's running or you'll stop it.

The procedure will allow you to run MONITOR interactively or as a detached task and should be self explanatory.

#---
15) "Image Accounting" - Accounting is enabled by default. However image accounting is not. This is because it places a very heavy load on the operating system. Do not run this procedure for long periods unless you're willing to field the complaints.

This procedure creates a sub-process which will enable image accounting for the delta time you specify and then produce a report of the results in ADVMGR:[TMP]ACT.LOG. It takes two arguments, the delta hours and the delta minutes.

#---
16) "Show Working Sets" - It is often desirable to know about each process' use of working set pages. This selection will type out a list of all processes and information on their working set use and parameters.

#---
17) "Login as Other" - Login as another user does not require that user's password, just the user name. It can be used to test a program in another environment.

#---
18) "Display Instructions" - Displays a brief note referring to this file.

#---
19) "Date and Time" - This will give you the date and time plus a bunch of other information and in living color. I run it at login.

#---
20) "Spy" - Monitor another terminal on this node.

****

This is the end of the menu description. The discussion that follows is general and describes some software that is not normally distributed with the Advances Systems Manager's Package but is contained in my own system which I can distribute. The software is available individually on request.

2. The good, the bad and the weird. or how to be a Systems Programmer.

This has been a a long and exciting learning journey. It started forty years ago at the old datanex, Inc. working with my friend and fellow self taught hacker/programmer, Paul Klisner. We were busy crashing the company VAX computers nearly every morning, much to the chagrin of the boss. I learned a lot, but times changed, datanex got sold, eventually closed and we moved on.

Over the last ten or so years, I've managed to get my own alpha (AXP) system and have been porting what I learned over to it. Not always easy. Below is a description of the results.

Along the way, I've met and often made friends with some very smart and very nice folks. It seems that OpenVMS has not gone away and may well be making a resurgence. Well it should, it's often the best OS available for many business and engineering/scientific operations.

Let's start with the difficult and fun stuff.

2.1. Executlets

Please note: All work involving executlets is based on code written by Brian Schenkenberger (aka VAXman) of Tmesis. There is more work described later on that required my contacting him. He was most cooperative, gracious and generous with his time. Many thanks.

IMPORTANT NOTE!!!! Most alpha systems have a page size of 8192 which is equivalent to the /BPAGE=13 link option (default). If you determine that your system has a different page size, "$ WRITE SYS$OUTPUT F$GETSYI("PAGE_SIZE")", you must modify all the link option files (.OPT, .COM, .SW) in my area to reflect this. See the DCL HELP LINK /BPAGE discussion for details.

AST_LOAD.EXE and AST_UNLOAD.EXE are used to load and unload special code to/from system memory. It is the compiled and linked "executlet" file that serves as a substitute for the old VAX "copy the code to S0 and queue an AST to it" that does not work on ALPHA. The "executlet" code, linker switch and option files are in R$:[AST]AST_LEI.*.

You can add your own executlet code to this file. It will hold as much as you want and contains a good deal of reusable (common) code for the piggy_back return ASTs. I've made it simple enough that, with a little patient reading, it should be clear in its operation. When stable, it should be loaded at boot. Note! It is done this way to allow the system to queue an AST. Putting the code into an "executlet" means there is no address relocation required as there is no linkage psect. The loading system subroutine takes care of addresses. No linkage means there is another mechanism for queuing the AST. It is the same as done for the OS executlets (system API's) that are loaded during the system boot process.

The loader adds in special code that handles queuing the AST and, when you call the code at the address returned from FIND_LDRIMG in the shared library MARLIB, you are actually getting the address of this special code. This code queues an AST to your victim and eventually, in the process of changing mode to kernel, gets your victim to run your code. (Whew! That was a mouthful). It actually

makes writing the code easier.

Making use of executlets is demonstrated in DEFDIR, GETCMD, DISMOUNT, FORCEPRV and PURGWS.

****
2.2. TSR
****

TSR.EXE is Paul Klisner and my "Terminate and Stay Resident" program that sits up in P1 space where it stays as long as the process remains. It captures control keys (out of band keys) and executes code asynchronously to other operations in the same process. Thus (in its present form) a CNTRL-X will clear the screen, a CNTRL-D will write a nice looking calendar in the upper right corner of the screen with today's date highlighted and a CNTRL-V will toggle the DCL verify state even in the middle of executing a script. Can be very useful when debugging DCL scripts.

A new addition is the ability to capture the file id’S of all the files a process has open at any given instant. It does this with the CNTRL-F key. Rather than pour file ID’s all over your screen in the middle of a process, the ^F key appears to do nothing. However, it actually creates a logical (RICK_FID) that contains all the file ID’s. When you are back to a command prompt, FID_TO_NAME.COM will translate all the file ID’s to full file specifications and print them on your screen, or put them in a file (TEMPD:OPF.TXT) if you pass the script an argument.

CNTRL-G will switch the undocumented and unsupported DCL command “SET WATCH FILE” which documents all the file operations happening in your process. This is a toggle.

Finally, a CNTRL-J will print out the startup greeting message (instructions) on your terminal.

We have also made things more flexible by allowing changes in what control keys are associated with what operations and even what operations are enabled. You control this by defining a symbol to run TSR.EXE and passing it an argument. The argument consists of a string of characters without spaces. It consists of variables separated by commas and must be in pairs. The first item in the pair is a number representing the operation as displayed when TSR is run or when you hit ^J. The second item in the pair is either a letter representing the control character which will be trapped and will activate the function described or a null (two commas in a row without a space) which will disable the function and not trap the control character associated with it by default. ^J will show the results. An “^” followed by a space indicated that the function is disabled.

****
2.3. Spy
****

SPY.EXE (SPY) requires the change mode to kernel privilege. If unprivileged users need this program, it must be installed.

Run it from the menu or

```
$ SPY :== $(WHEREEVER)SPY.EXE
```

At the DCL command line, type:

$ SPY YYxn[,letter,letter] where YYxn is the name of the terminal to be

monitored. Optionally you may specify single letter arguments for the two control characters used; the first for disconnect and the second for command entry. For example:

```
$ SPY TNA4,f,b
```

Brief instructions will be printed and from that point on, whatever appears on the screen of the monitored terminal will appear on your terminal as well (with exceptions noted below).

By default, CTRL-\ will decouple you from the other screen and exit the program, returning S0 space to the OS.

CTRL-B will prompt you to enter one DCL command for the other terminal (process) to execute.

TERMINAL EMULATOR SETTINGS:

We use Putty. Spy works best with the following settings that seem to be different from the default:

Settings: IP and Port as appropriate. Con type: other - telnet
Keyboard: (127), ESC[n~
Window: 132X40, change size of font, the rest as you like.
Translation: ISO-8859-1:1998 (Latin-1, West Europe)
Connection: Disable Nagle's algorithm, IPv4
Telnet: BSD, Active

BACKGROUND:

SPY is a labor of love. We did it over a period of more than six years and without the benefit of driver listings. Thus it was done by means of deductive reasoning, "poke and hope" and a terrifying number of system crashes.

Along the way we detoured down a variety of dead ends and made some wrong turns, but, in the end, we have something that we feel is useful, but not perfect. It seems to handle any terminal driver based on the TT class terminal driver and can be run by a person logged in over DECnet. It handles large amounts of output without dropping characters and seems to handle escape sequences generated programmatically very well. Command line editing works as advertised but does not always display correctly which can make it a bit frustrating, especially to first time users.

Other versions (commercial) may have overcome problems we are still working on. We have no experience with them.

KNOWN DEFICIENCIES:

One of the problems we have encountered right from the beginning is the propensity of the Alpha class driver to omit a leading character from the output stream placed there at the beginning of a $QIO. It took some time, but patterns started to emerge and we have been able to compensate for this.

So far We've tested: OP, TT, VT, TN, FT, RT (as your terminal) and we suspect it wil work on TX. There is one known exception; while you may monitor someone by doing a SET HOST to their node, you cannot monitor a person who has logged in over DECNET. Much to our surprise, doing so did not crash the system, it just didn't work.

Those programs which utilize the SMG$ Screen Management Facilities, including DECTPU, will not display their buffers on the monitoring screen.

Additionally, if you SPY from (not too) a SWIM (see below) window, change the first control character. The SWIM system uses the CTRL-\ character as an out-of-band character so you can't disconnect if you use it for SPY.

If you come up with a situation where you have a terminal device name that uses the class TT driver and SPY tells you it's a bad choice, call us. We'll make arrangements.

Command line editing can be mildly frustrating. You will notice that things sometimes echo strangely on both terminals. This is most notable deleting characters with the backspace key. It echos the state of the command line on a new line and, on the terminal being monitored, will not overwrite (delete) the first character entered. Actually it does and full line editing (arrow key movement, CNTRL-R and toggle over-strike) all work just fine. If it doesn't echo exactly right, CNTRL-R will display the actual command line.

We're working on it. You may discover other problems. It's complicated.

****
2.4. Multi-session
****

We will describe all the methods we have for multi sessions here. We have several:

1) BOSS.EXE
2) SWIM.EXE
3) PSWITCH.COM

BOSS.EXE is someone else’s 'multi-session on a single terminal' code. It should be installed with the non-threatening privileges of PHY_IO and SHARE so everyone can use it. It has an advantage of being able to buffer some output so that the screen can be refreshed on switching processes.

Users who will be in menu driven software (typical business) and have a need to use two terminals/windows, will be well served with BOSS properly started at login. An example might be a user who is at the counter filling out an invoice for a customer and has a need to look up an inventory item to see if it's available. Use the following setup in LOGON.COM:

```
$ BSL[0,8] == 28
$ CI[0,8] == 9
$ CJ[0,8] == 10
$ S1 == "'''BSL'''CI'''BSL'''CJ'YOUR MAIN MENU"
$ S2 == "'''BSL'''CI'''BSL'''CJ'YOUR MAIN MENU"
$ BOSS/START=(A,B)/STUFF=("'''S1'","'''S2'")
$ EXIT
```

This creates two sub-processes, each running the main menu. The user switches between them by typing CTRL-\ followed by A or B. The screen will be refreshed automatically with this setup.

The disadvantages are that BOSS is full screen even on a terminal emulator and logging out is a bit complicated.

SWIM.EXE is someone else's "multi-session on a single terminal" code. It should

be installed with the non-threatening privileges of PHY_IO and SHARE so everyone can use it. It is pleasant to use in that there are multiple "windows" on the terminal you can switch between/among with CTRL-]. The windows can be moved and resized. They will overlap and will repaint when selected.

If you are running business applications using terminal emulation (Putty?) software which can present a screen much larger than the VT100 standard 24 X 80, you may prefer, as we do, using SWIM with its 24 X 80 terminal windows. Each window can be maximized to fill the entire emulated terminal window using CTRL-\ X and quickly returned to their previous size and positions using the same two key sequence. Use the following setup in your LOGON.COM:

```
$ WINDOWST
$ IF FT
$ THEN
$   do some stuff and run a program
$ ELSE
$   WINDOWS
$ ENDIF
$ SET TERMINAL/APP
$ EXIT
```

The above is good for users who are menu driven. In my own case:

```
$ WINDOWST
$ ...
$ IF .NOT.FT THEN $ SEDIT     ! Start my editor as a sub-process.
$ ...
$ TSR
```

The symbols WINDOWST, FT and WINDOWS are defined in SYS$MANAGER:SYLOGIN.COM. They execute scripts in my area that test for the FTAx: terminal driver, set the global symbol FT as 0 or 1 and do other things that make life easier. In the most simple form, modify your LOGON.COM file to do WINDOWST early on and then test FT and don't start the editor as a sub-process if true. It saves on process slots and avoids unnecessary clutter. If I want a couple of windows, I simply type WINDOWS at the command prompt.

PSWITCH.COM is a script in my area that allows many sessions to be started and run at your terminal, up to the limit in your authorize database. We have enabled virtual terminals on this system so that you can create and switch multiple processes using DCL commands - CONNECT or DISCONNECT. This process has been automated to a degree. Provided that you are logged in on a terminal which is capable of disconnecting (not TCP/IP), PSWITCH.COM will:

1) Detach from your single process and allow you to log in to create a second process.

2) If you have two processes, it will switch you back and forth.

3) You may log out of your existing process while switching by passing the argument "L".

4) You may force more than two processes by passing the argument "I". When you have more than two processes, PSWITCH will list them and give you a choice of processes to which you can connect when switching. You will need to remember the virtual terminal name for the process you want. It does not work for terminals connected through TCP/IP (TNxy:) because they cannot be disconnected.

Recommendations:

We lean in the direction of SWIM for business (menu driven) applications because I've modified it so it can be automated easily, it is flexible and it now does color. If you choose to use SWIM, call WINDOWST early in the LOGON.COM procedure or things will not work well for you.

Note! We automate things in part by using BOUNCE.EXE to put a carriage return into our typeahead buffer so that SWIM goes its merry way without intervention. You need the PRMMBX privilege to use BOUNCE and minimum privileged users don't have it. We gave it to USER to smooth the way. This also means that unprivileged users can't easily use POKE.EXE to create mischief.

For development, swim is a good choice. It also works well coming in on TCP/IP.

****
2.5. Cluster considerations
****

I stumbled upon Advanced Virtualization Technologies and somehow made contact with their CEO, Ralf van Diesen. This fellow could not have been nicer or more helpful if he had lived next door rather than in The Netherlands. He spent many hours with me in Zoom meetings and a flood of emails getting his software up and running on my systems. Many Thanks.

His software does not run on a host operating system, thus eliminating that barrier to stability, which, I know from experience, is formidable. I run one node on an old Dell laptop Intel I5 X64. We refer to the installation as "bare metal" and it just sits there and runs. The second node is running on my more modern Dell laptop which has Windows 11 and VMware installed. It acts more like a yo-yo because it runs on top of Windows which is always being updated as well as requiring a virtual machine software such as Vmware which is always being updated.

So my VMS system now consists of a two node cluster and a quorum disk running on two computers.

A large number of my programs have been written and/or adjusted so they work on multiple nodes through the use of the utility program, SYSMAN. Of course some are cluster wide by nature. Menu indicates this. We attempt to set an environment that will make it unnecessary for the system manager to log into a node through DECNET to manage it.

When attempting to execute a procedure on other clusters, please remember the limitations of the "DO" command in SYSMAN. It will run the procedure using your privileges but does not do a login nor does it pass process context. You are limited to one command line. Input to the procedure is not possible except by passing arguments on the command line. This requires proper setup of the command and requires that the procedure is available on the clusters you chose.

Learn SYSMAN. It will help.

DECnet:

When using DECnet to log into other nodes, use some caution on how you go about running these programs. WARNING!!!!! Using my menu system to run stuff will crash the system if done over DECnet. The program, MENU, attempts to shove some characters into the terminal typeahead buffer. This works on terminal drivers based on the TT class driver but not on the RT driver. We check for the RT driver

and refuse to run if found, but there are other protocols that can log you into another node and may very well crash the system as well. We can’t test for them all. You have been warned.

GUARANTEED CRASH!!!

Safe computing.

****
2.6. Use of color
****

There are functions in MARLIB associated with the IMPUTM, DISPLAYM, etc. that relate to colorizing the text you are using. Most terminal emulators, including Puttty, now respond to modern ANSII color escape sequences. We have opted to use the longer escape sequence that will represent any color you want, rather than the older and shorter escape sequences that were rather specific to a particular terminal/emulator.

<ESC>[38;2;r;g;b;48;2;r;g;bm

Where r g and b are numbers from 0 to 255 (standard RGB).

We have a table of colors predefined in header/include/library files used in the three languages in which we program.

These colors must correspond to COLORTAB in MARLIB.

Any changes to COLORTAB, require similar changes to:

R$:[COM]COLOR.COM
R$:[TP]IO.H
R$:[DBL]HEADER.INC
R$:[DBL]COLOR.INC

which define the colors as integer indexes off COLORTAB.

```
DEFAULT =   0
RED =       1
LTRED =     2
ORANGE =    3
GOLD =      4
GREEN =     5
YELLOW =    6
BLUE =      7
LTBLUE =    8
MAGENTA =   9
VIOLET =    10
CYAN =      11
WHITE =     12
BLACK =     13
NORMAL =    14

COLORTAB:
      .ADDRESS    DEFAULT_COLOR
      .ADDRESS    RED_COLOR
      .ADDRESS    LTRED_COLOR
      .ADDRESS    ORANGE_COLOR
      .ADDRESS    GOLD_COLOR
```

```
      .ADDRESS    GREEN_COLOR
      .ADDRESS    YELLOW_COLOR
      .ADDRESS    BLUE_COLOR
      .ADDRESS    LTBLUE_COLOR
      .ADDRESS    MAGENTA_COLOR
      .ADDRESS    VIOLET_COLOR
      .ADDRESS    CYAN_COLOR
      .ADDRESS    WHITE_COLOR
      .ADDRESS    BLACK_COLOR
      .ADDRESS    NORMAL_COLOR

      .ALIGN      QUAD
DEFAULT_COLOR:
      .ASCID      <ESC>"[0m"
      .ALIGN      QUAD
RED_COLOR:
      .ASCID      <ESC>"[38;2;255;0;0;48;2;0;0;0m"
      .ALIGN      QUAD
LTRED_COLOR:
      .ASCID      <ESC>"[38;2;255;128;128;48;2;0;0;0m"
      .ALIGN      QUAD
ORANGE_COLOR:
      .ASCID      <ESC>"[38;2;255;165;0;48;2;0;0;0m"
      .ALIGN      QUAD
GOLD_COLOR:
      .ASCID      <ESC>"[38;2;255;192;0;48;2;0;0;0m"
      .ALIGN      QUAD
GREEN_COLOR:
      .ASCID      <ESC>"[38;2;0;255;0;48;2;0;0;0m"
      .ALIGN      QUAD
YELLOW_COLOR:
      .ASCID      <ESC>"[38;2;255;255;0;48;2;0;0;0m"
      .ALIGN      QUAD
BLUE_COLOR:
      .ASCID      <ESC>"[38;2;0;0;255;48;2;0;0;0m"
      .ALIGN      QUAD
LTBLUE_COLOR:
      .ASCID      <ESC>"[38;2;128;128;255;48;2;0;0;0m"
      .ALIGN      QUAD
MAGENTA_COLOR:
      .ASCID      <ESC>"[38;2;255;0;255;48;2;0;0;0m"
      .ALIGN      QUAD
VIOLET_COLOR:
      .ASCID      <ESC>"[38;2;238;130;238;48;2;0;0;0m"
      .ALIGN      QUAD
CYAN_COLOR:
      .ASCID      <ESC>"[38;2;0;255;255;48;2;0;0;0m"
      .ALIGN      QUAD
WHITE_COLOR:
      .ASCID      <ESC>"[38;2;192;192;192;48;2;0;0;0m"
      .ALIGN      QUAD
BLACK_COLOR:
      .ASCID      <ESC>"[38;2;0;0;0;48;2;192;192;192m"
      .ALIGN      QUAD
NORMAL_COLOR:
      .ASCID      <ESC>"[0m"
```

For MACRO32, we have two functions used by the input/output routines that handle color. They are discussed below and should be used in conjunction with the I/O

routines.

For general programming, we have the function INSERT_COLOR and the associated MACRO, PR_C, which takes the argument of the COLOR you want and assumes R10 points to the insert point. The function updates R10.

The programming languages have the color offsets defined in appropriate header/include files and rely on the functions associated with I/O in MARLIB.

DCL calls COLOR.COM which defines color symbols as escape sequences to be written to the terminal.

If you add your own colors, you might consider adding them to the end of the list to avoid missing something somewhere.

Special considerations:

For reasons not entirely clear, two operations we know of seem to have trouble handling color changes; namely SMG$ terminal operations and SPY terminal output on the watching terminal. We are suspicious that there is a timing problem but can't prove it.

We have gotten around many of the difficulties by inserting the DEFAULT escape sequence, <ESC>[0m, before the color change. This seems to help a great deal. Also in SMG$ software, due to limited user definable operations, the number of colors we can represent is limited.

****
2.7. Command line prompt
****

Many operating systems employ part or all of the default location where the user is working as the command prompt. VMS doesn't. It should.

Enter VAXman. Using a modified execulet that doesn't initialize in the normal fashion, all calls to the system service to change the default directory will be trapped and the location containing the prompt string modified to contain the directory structure. The reason it is done this way is that it must actually write over the dispatch code for the system service, SYS$SETDDIR. This makes it unique but general for intercepting (trapping) system services. In this case, the code is found in my [SSDDIR] directory along with a unique BUILD.COM script. The process puts SSDDIR.EXE in SYS$LOADABLE_IMAGES and SSDDLOADER.EXE in R$:[EXE]. Running the loader is all that's necessary to get it going.

The benefit of doing it this way is that it causes any call to SYS$SETDDIR to set the prompt for that user immediately. However, there two disadvantages, namely, it is universal for the entire node and it's irreversible until reboot. You can run it or ignore it. Your call. The call to the procedure is located in ADV_SETUP.COM and STARTSTOP.DAT.

****
2.8. Programmer's tricks
****

Over the years, I've developed some tricks and shortcuts that make programming in this environment a bit easier. I'm sure everyone who has programmed on VMS has done so. I'll share mine with you.

Customizing the EVE editor and running it as a sub process is well worth the

effort to get it set up. It makes the editor yours and very quick and easy to use. There is a very detailed discussion of how to do things below.

Defining symbols, logicals and keys in your LOGON.COM file can get carried away. Years ago, it would take me twenty seconds or so to log in and placed a noticeable load on the system. Each definition had to endure the overhead of image startup and rundown; significant on VMS. Hunter Goatly wrote a program that did all this in one place. It is written in MACRO32, but don't let that scare you. I added key definitions. Modifying a copy of R$:[MAR]LOGIN.MAR to suit your needs should be straight forward. I strongly suggest you use it.

If you are willing to structure your directory environment in a certain way, I have a DCL script that makes compiling and linking programs written in any language on the system, very easy. You can copy it from my area, R$:[COM]BUILD.COM, make a quick change to one line and it's yours.

What you do is create a few sub-directories one level down from your default directory that can be reference by your hidden logical just as I've described above in referencing my stuff. You will want to create and use: [COM], [EXE] and directories for each language in which you intend to write programs i.e. [MAR].

Set your default directory to each language directory and do the following:

$ COPY R$:[‘language_extension’]’language_extension’.* []

For example: $ COPY R$:[MAR]MAR.* []

This will give you default switches and options (.CSW, .SW and .OPT) for compiling and linking that BUILD.COM will use. For more advanced situations, any/all of the .CSW, .SW and .OPT files can be copied to files with the original extension and the same name as the program to which they will apply. BUILD.COM will use it/them instead of the default ones you copied over from my area. You may modify your new files for complete customization.

If you log in as ADVMGR, you will notice it has been set up this way.

SYSTEMS PROGRAMMERS!

If you are a systems programmer, you know that you need to be very careful not to try and manipulate a device/process that isn't there. It crashes the system. You must also lock appropriate databases and raise IPL, etc. checking for errors and handling them in an appropriate fashion. It is easy to overlook a check that you should have made or to program in such a way that you either can't do what you want or you crash the system.

We have produced four (five?) routines that help this process a great deal in that they are tested tried and, we believe, true. They are:

GRAB_TERMINAL
GRAB_DEVICE
GRAB_PROCESS
GRAB_KAST

And another subroutine to be used with the first, TH_STUFF. They are all in MARLIB, the shared library (DLL). The naming convention for the fourth is misleading at best but I wanted to group them as they are used in a similar fashion. For each of the first three, you do:

```
      $CMKRNL_S -
```

```
        ROUTIN = GRAB_XXX, -
        ARGLST = KARGS
```

KARGS varies with the situation and is described in detail in the documentation found at the top of MARLIB.MAR and later in this document. We will just note the operational characteristics here.

The first three are similar. They attempt to gain exclusive access to a valid named device/process, performing all the desired checks along the way. If successful, they will then execute a named JSB subroutine in your program whose address you passed in the argument list. They will then release everything appropriately. By the way, it is up to you to make sure everything is locked in memory so there are no page faults. We're operating at high IPL.

Should the grabbed device be a terminal of TT class driver type, and the name of the JSB routine to be executed is TT_STUFF (place a string into the type-ahead buffer of the terminal), the JSB routine is in MARLIB and will use a passed address of the string descriptor you desire.

The difference between GRAB_TERMINAL and GRAB_DEVICE is that GRAB_TERMINAL includes checks that apply only to terminals. Use the wrong one and it's crash city.

GRAB_KAST is slightly different in that it is not getting control of another device/process but rather building an ACB in S0 space so an AST you've written and put in AST_LEI.MAR will be executed by the intended process. There is an option to execute a JSB routine in your program, should you need to put some information into the ACB prior to your victim executing the AST. See DISMOUNT.MAR as an example.

Another difference is that you will use two MACRO's, KAST_OFFSETS and KAST_ARGUMENNTS in your program. The optional subroutine is provided in a unique place and used rather differently from the other GRAB_XXXX routines.

These routines are used in most of my systems programs and make for a dependable execution environment and much easier programming. They are described in more detail in section 6. Note! If you add your routine to AST_LEI.MAR, please remember to update KAST_OFFSETS in MACROLIB.LIB and rebuild the .MLB's.

****
2.9. Symbionts
****

A symbiont is an executable program that attaches itself to a queue through the queue manager and which processes in some manner, that which has been placed in the queue. Two examples supplied by VMS are the batch queues and the print queues. The batch queue symbiont simply creates a new detached process and then uses the contents of the queued file as a DCL command source. This constitutes a “Batch Job”.

The print symbiont usually does more in the way of processing what is in the queued file on its way to a printer device.

Sometimes there are advantages to being able to queue files to a named queue for further and possibly sequential processing. Rather than running some batch process passing it parameters that will eventually be processed by the batch queues or doing the processing by hand, A simple “$ COMMAND FILE” is often the most simple.

John Osudar of Argonne National Laboratory wrote a symbiont that is intended for

general processing. It is well suited to this in that it utilizes a user supplied DCL procedure to do the processing. This procedure can do pretty much what it pleases.

In our environment, printing is difficult. What we have done is to write a Windows program that periodically scans the default folder for the Windows FTP server process for files with the .LIS extension. On the VMS system, we use the "$ PRINT FILE" command to queue up the file to be transferred to Windows for printing. The SYS$PRINT queue is initialized with the processor being the EXECSYMB symbiont and our DCL processing script, REMPRT.COM, as a parameter. RMTPRT.COM copies the submitted file to one with the .LIS extension before FTPing it to Windows.

We have extended this by setting up two additional queues for PUTing and GETing files to/from the Windows FTP default directory. We define FPUTA, FPUTB, FGETA and FGETB for sending/fetching both ASCII and binary files. It is convenient. FGETx is more complex. Using the QUEUE will not work unless the file to be fetched already exists in the default directory at the time the command is issued (the nature of SUBMIT.EXE on which it is based). To get around this, FGETA points to a DCL script of the same name that either uses the QUEUE or does the FTP operation directly in the case that either the file is not present or you use a wild card.

At the end of R$:[COM]OUR_SITE.COM, which is called from SYSTARTUP_VMS.COM on startup, is a line that points to the script, @R$:[COM]QOPERATIONS.COM. You may comment this out to skip the QUEUE operations. Otherwise modify the script to suit your needs.

In our case we make sure that this script is running on the primary node of the cluster or a stand alone node by checking that SYS$TOPSYS is SYS0. The remainder of the script is very specific to our needs but gives some good examples of using queues and EXECSYMB.

****
2.10. Miscellaneous notes
****

TCP/IP NTP Time Service:

The TCP/IP stack provided by HP leaves a bit to be desired. Much works but some services partially work, are confusing to configure or don't work. The NTP time service was working but then stopped working all together. Turns out the FACEBOOK time server sends out an illegal datum, yielding a divide by zero error in NTPD.EXE and pukes on your shoes. If you experience this, try different time servers. TIME4.GOOGLE.COM works pretty well. I also discovered that Windows does not like you to use numerical IP addresses to reach the internet when you are doing it from AXP emulators, at least some of them. As soon as I properly configured the BIND in TCP/IP core services with a host name in the Hosts file, everything seemed to work.

Daylight savings time changes seems to confuse the time service. Politicians keep messing with daylight savings and I don't trust the automatic VMS procedure. Just setting the new time by hand yields a "sanity check" in NTP. This will continue on restart. The best way, or one that works, is to: stop and disable the service. Disabling the service appears to be necessary. Doing so removes the DCL command NTPDATE. So issue the following command: $ NTPDATE == "SYSSYSTEM:TCPIP$NTPDATE.EXE" and then issue a $ NTPDATE 'Time Service URL' command. Restart/enable the time service.

If you are in a cluster, it is probably best to use one node as the node that

reaches out over the internet to get the time and have it broadcast to the other nodes. In the primary configuration file, you need at least one “server ‘Time Service URL’” line and then one “broadcast x.x.x.255” line. In my case the x’s are 192.168.1.255, the first three numbers representing my Local Area Network.

On the other nodes, you should have a server line that points directly to the primary. In my case: “server tulasi”. Start the time service on all nodes and things go well. You will find that the time on all the nodes is very accurate and even more precise except under the most adverse of circumstances. Make sure your HOSTS file has your local addresses and names set up correctly.

P2_COMMON User Space:

P2_COMMON space needs a bit of discussion. I have an MARLIB (dll) callable subroutine that makes use of it and is fairly flexible.

On AXP boxes (and Integrity I assume) there is a very large address space not used by VMS. It simply does not exist on the old VAX. The 32 bit OS uses addresses ^x00000000 through ^x7fffffff for user space and ^x80000000 through ^xffffffff for the operating system. In operating system space, addresses would be negative in two’s compliment arithmetic. User space is divided into two areas, P0 - ^x00000000 through ^x3fffffff for programs and P1 - ^x40000000 through ^x7fffffff for control and stack space.

The AXP box is 64 bit and puts the operating system in negative address space just like on the VAX. It does so by extending the high order bit of the lower 32 bit address into the upper 32 bits. This leaves addresses ^x00000000 80000000 through ^x7fffffff ffffffff as unused and inaccessible to a 32 bit OS. This is a lot of space, about 10^18 bytes.

The later versions of VMS provide system services for accessing this space. SYS$CRMPSC_PFN_64 is one example and the system service call I use in the MARLIB subroutine called P2_COMMON. My code can allocate about two million 8192 byte CPU pages. The addressing is tricky and requires a passing familiarity with MACRO64 assembler language. This is why I provided the routine. It’s relatively easy to use and performs a variety of useful operations for you. One of the benefits is that you can allocate quite a chunk of memory for the personal use of your process and you will not have to worry about running out of program space or worry about having your stack overwrite your data or your data overwriting symbol and logical space.

Read the details below.

Decwindows:

WML = Windows/Mac/(31 Flavors of Linux)

I am very aware that the modern paradigm insists on having windows and a mouse. Ever since Gates and Jobs stole the idea from Xerox, people have been convinced this was the new way of doing business. You could have two or more programs running simultaneously and switch between them with a click of the mouse. This is a very handy thing to have. The mouse made highlighting text simple. A lot of effort and time have been spent on rather elaborate word processing programs, spread sheets and other programs for producing elaborate output for consumer consumption.

There are things people need to do in a very large area of business and personal life for which WML computers are indispensable. They are not going away.

That does not mean they are ideal for all areas of business. Having two or three windows running separate programs open at the same time is very desirable for a lot

of business operations. It is easily accomplished in VMS without elaborate windows.

People are accustomed to using a mouse to the point that they do not realize how much time is wasted using the mouse to select text boxes for the next item of input. It is not efficient. In WML programs, the screen layouts are often confusing and/or misleading while the old numbered input with masked data entry items are very easy to understand and moving from item to item with the tab key is very efficient, much more than using a mouse.

Individuals are not going to spend the money on OpenVMS, nor should they. Business and engineering operations of some size will benefit greatly from the fact that OpenVMS is so efficient and safe when properly configured. Sometimes the old fashioned way is better. It depends on the circumstances.

I have Decwindows running on my system. I find it basically useless. I will not adapt my software to a useless, high overhead paradigm that simply gets in the way.

****
2.11.Advanced Virtualization Technologies (AVT) Tape Operations
****

This section applies to the AVT AXP emulator only.

Backups of the OpenVMS system can be tricky if you are running an emulator rather than old hardware. Some emulators allow you to add a physical tape device to the system, some do not.

If you are running an emulator in a hosted environment, you can simply use it to back up the disk devices. I’ve been doing this for years with emulators such as FreeAXP and CHARON. The AVT emulator, which I have found to be superior to the others, is a slightly different story. This gets a bit complicated so stay with me.

AVT’s AXP emulator runs on an X86 64 bit processor. It does not use a host operating system, but installs and runs “bare metal”. This is just one reason I feel it is superior. There are many more.

So, for most installations, you will have X86 boxes that act like AXP boxes on which you will run OpenVMS. The X86 boxes may or may not have physical tapes attached and AVT has you covered in either case.

No tape:

Like all the emulators with which I’m familiar, the emulator software allows you to create virtual disks and treat them like physical disks mounted to the box. These virtual disks can be copied to a system connected to the same Ethernet switch that has some backup scheme already in place. For example, you might have a Windows system to which you can copy the disks. Then you just back up your Windows system. This works but is inefficient.

In some cases you may have the AVT emulator running on VMWare or Turbobox. The virtual disks will be in a disk created by VMWare running on the host which can be backed up normally.

In either case, bare metal or a virtual box, you can define a mount point on the host system, mount it in the AVT emulator and access virtual disk created on it. This is more efficient.

Lastly, you can use the TAPEMGR tape manager facility provided by AVT which gives

you a very versatile tape environment.

Tape:

The AVT emulator allows you to use physical disks as well as virtual disks and both physical and virtual tape drives. Virtual tape drives are actually virtual disks that can grow as needed. In my environment, I use virtual drives exclusively.

AVT's TAPEMGR system installs on VMS using VMSINSTAL and adds TAPEMGR as a DCL command. The TAPEMGR command allows you to create virtual tape drives (disk files with a name of your choice) as well as loading and unloading them. This gives you flexibility. An example from my script, RBACKUP.COM will demonstrate:

```
$    tapemgr set 'tapedev'/compress
$    tapemgr unload 'tapedev'
$    tapemgr create 'file'
$    tapemgr load 'tapedev' 'file'
$    mount/for/over=id 'tapedev'
$    backup/image/noalias/verify-
/ignore=(interlock,label_processing) -
'devname' -
'tapedev''snam'.bck/init/rewind
$    dismount/unload 'tapedev'
$    tapemgr unload 'tapedev'
```

Note! 'snam' is the mounted path to the virtual tape drive file. I use /image for tape backup operations.

The restoration is only a bit more complicated. I define a virtual disk as removable. This allows the machine to start even if the named disk file is not present. It also allows you to "$ DISMOUNT/UNLOAD" the disk drive and rename the restored drive/file to what you want. The 'restdev' file referenced below must be as big or bigger than the drive previously backed up. It is created with the AVT software and must initially be named to the name provided when the disk was defined to the system.

```
$    tapemgr load 'tapedev' 'file'
$    mount/for/over=id 'tapedev'
$    mount/for/over=id 'restdev'
$    backup/image/ignore=(label)/verify -
'tapedev''snam'.bck/sav -
'restdev'
$    dismount/unload 'tapedev'
$    dismount/unload 'restdev'
$    tapemgr unload 'tapedev'
$
```

Now the 'restdev' file can be renamed as you like. If you are using this backup/restore process to enlarge a disk/file, there are two steps to take to make the extra space available to VMS.

Mount the restored device privately and issue the following two commands:

```
$ SET VOLUME/LIMIT 'RESTDEV'
```

If an error occurs at this point, see the AVT manual for how to handle it.

```
$ SET VOLUME/SIZE 'RESTDEV'
```

This will make the added space available.

The virtual tape files (with compression active) are much smaller and easily copied and incorporated into a backup scheme.

****
2.12.Help
****

There are two help procedures (also found in my menu) that the manager and systems programmer will likely find useful.

The first is R$:[COM]RHELP.COM. This, using the MOST display program, will either display a variety of files you can examine or use the LIBRARY/EXTRACT facility to produce a .HLP file from a .HLB file you may peruse.

For systems programmers we offer a means of looking through a variety of system resources for the definitions of a rather large number of symbols. These sources include the .H “C” language include files that are the same as the extraction macros of the form $xxxDEF used in MACRO32 programs contained in STARLET.OLB. You may also extract global and universal symbols from the .STB files created by the linker for shared images. For images for which .STB files have not been requested, you can extract them from .EXE files. Finally, The image SYS$BASE_IMAGE.EXE which is automatically linked into executable files by the ALPHA version of the linker, has a .MAP file associated with it. We will extracted defined symbols based on a search string you provide or show the whole thing, your choice.

****
3. System Programs (not all of these come with ASMC)
****

BOUNCE.EXE is one of several programs that use the ability of POKE.EXE to send characters to a processes type-ahead buffer. In this case, the terminal is always the terminal running BOUNCE. It takes a command consisting of two arguments separated by a comma. The first argument is a delay in milliseconds and the second, character(s) to be put in the buffer after the delay. There is a special character used which is interpreted two ways. The reverse accent '`' as the first character in the second argument means the next character is to be converted to a control character and sent. Thus '`M' puts a carriage return in the type-ahead buffer. The reverse accent at the very end is replaced with the carriage return.

CLRERR.EXE will clear the error count on a device and, optionally, on the cpu and memory. Often useful for driver development.

CUST.EXE and WCUST.EXE, written in Synergx's DBL, are sample programs to add/delete/update a customer to/from an ISAM database, R$:{DATA}CUSTOMER.DAT. It is used to test the LOCKHOLDER.EXE program which will allow the manager to discover who has a particular record locked and has gone home. It also has been used to demonstrate rights identifiers and their use in application security. You can only access the database if you have been granted the CUST identifier. The "W" version uses DBL's windows code that emulates windowing (sort of).

DEFDIR.EXE will display the current default device and directory of another process. It makes use of execulet code in AST_LEI.EXE.

DISMOUNT.EXE will force another process to dismount and/or deallocate a device it supposedly has forgotten about. It makes use of execulet code in AST_LEI.EXE

and the JSB (R9) subroutine call into your code.

DKFREE.EXE shows the free space on all disk drives; something that VMS has never deemed desirable. It is to me.

DX.EXE is someone else's code to navigate your directory tree and assist in performing various operations on files.

EDITTP.EXE is used in the overall process of editing a file with the command: "e 'file-name'" when the EVE editor is running in a sub-process. This is discussed in detail below.

GETCMD.EXE was originally written by Hunter Goatly for the VAX. In porting it to AXP, he chose to use the executive read system service developed for the AXP. It works but it was written in BLISS and not elegant. I re-wrote it in MACRO32 using an execulet. I like it better and you can use the other persons terminal name to identify the process rather than the cumbersome PID. It does it with a single KAST and piggyback KAST return rather than multiple executive reads.

It prints out the other processes entire command recall buffer LIFO. This can be useful helping another person under circumstances where SPY cannot be used. It has nothing to do with terminals except displaying stuff on yours.

GLOGIN.EXE will allow the privileged user to log into the system as another user. The user's password is not needed, just the user name. Thanks Hunter Goatly.

IDS.EXE - stupid output, but exercises a call into an interesting MARLIB DLL routine that scans through all the PID's and executes a user supplied subroutine when a valid PID is found. In addition, it shows how to use P2 space on the Alpha computers.

LEARN64.EXE source can be edited to set up a MACRO64 instruction which will be executed and the contents of the involved registers (whatever) along with the results will be displayed. It is very helpful in learning this rather unusual language. It also has code to test structured macros from MACROLIB64,

LINKAGE.EXE is designed to demonstrate the actual ability on the ALPHA architecture to copy code into S0 system space and execute it. It exercises my MARLIB DLL routine to fix the addresses in a copied/moved linkage PSECT even when the address values are negative.

LIOH.EXE is a hook into the LOGINOUT.EXE system. It is made active by using LIOH.COM which defines a system wide logical (executive mode), installs the executable from SYS$SHARE: and modifies the active parameter list with SYSGEN.

It is written in C because it's complicated and the example in the book was in C. It's purpose is to allow the user to limit the successful logins for certain users to certain terminals or terminal types.

I feel this is an important addition to the security of VMS.

LOCKHOLDER.EXE will show the user who has called up a particular record in a shared (ISAM database) file for update and gone fishin'. You may then proceed to get nasty.

LOGIN.EXE is a MACRO32 program, originated by Hunter Goatly, that unprivileged users may copy and use. It takes a large number of symbol definitions, logical definitions and key definitions and causes them all to be defined in a single

program invocation. This means that user's LOGON.COM files which used to take a very long time to execute because a program was run from the start (demand zero page faulting, initialization operations and image rundown for each symbol, logical and key defined) may now login in in a very short time.

MACROLIB.LIB and MACROLIB64.LIB are full of .MACRO's used in my MACRO32 and MACRO64 programs and described in detail below. Some of the macros turn MACRO32 and MACRO64 into structured languages. It is controversial but important. When updating these files, be sure and re-create the .MLB files by running the script CRELIB.COM.

MOPTM.EXE, MOPTTP.EXE and MOPTD.EXE are written in C++ and are used in the updating and subsequent re-linking of programs using shared libraries (DLL's) for MACRO32, C++ and DBL respectively. They are run from DCL command procedures; UPDATEM.COM, UPDATEX.COM and UPDATED.COM (or UPDATE_ALL.COM invoked by the symbol UPDATE) and should not be run independently. Along with the DEC librarian, these programs automate the arduous task of making shareable executables (DLL's). This is especially difficult for C++.

MARLIB.EXE is the shared library (DLL) used in programs written in MACRO32, C++ and DBL. It will not run independently but is installed with privileges and linked into your program by the image activator.

MOST.ALPHA_EXE is somebody else's program to replace the "TYPE" command. It's help file in R$:{HELP} will describe the considerable flexibility of this program.

PDT.EXE prints a greeting with some system information and the date and time in a more readable format than SHOW TIME.

POKE.EXE is the recipient of mailbox messages which cause characters and strings to be placed into your or someone elses type-ahead buffer at appropriate times. For example, when you use the "e file" editing function to put the new file you want to edit into the edit buffer after attaching to the sub-process, POKE is responsible for processing this.

There is a general form of POKE, #3, which will parse a message consisting of "3," followed by the terminal name plus "," followed by the character string you want put into any terminal's type-ahead buffer. "4",terminal name,delay in ms, string or `X is a way of putting a control character or a string into your own type-ahead buffer.

The messages consist of strings sent to the permanent mailbox called "PSWMBX" created by POKE which is started from the system startup procedures described below.

PROCESSES.EXE shows a bunch of information about all the processes on the system, including terminal name.

PURGWS.EXE is a program that goes back a very long way. I wrote it originally for VAX computers that were having trouble because of insufficient physical memory. I gave it to the 4J school district in Eugene where they had a gazillion people running DEC's ALL-IN-ONE office suite. Nobody would log out when they were finished using it and consequently, whenever an idle process decided to become active again, much thrashing (page file content manipulation) would occur getting the process going again, making the user wait a significant time and slowing down all the other processes on the system. DEC tried to sell them very expensive memory (BI bus).

The algorithms VMS uses to control working sets favors keeping them as large as possible. This is good when the process is not idle for long. PURGWS takes a different view and causes idle processes to purge their own working sets to the page file thus freeing up physical memory so that other processes can get it when needed. For almost all normal business situations, this program will cause more bother than help.

This program saved the school district about a million dollars. I never even got thanked.

RMTC.EXE makes use of POKE.EXE to send a message, command, script to execute to a whole list of terminals. The list can be hand typed in at run time or stored in a data file. You might use this to log everyone out when needed.

Note: This program uses the SENDMAIL in MARLIB to send a string to POKE.EXE to do the actual type-ahead stuffing. Anyone with PRMMBX privilege can use this since POKE is installed with the necessary privileges. Be careful who knows about it as a fellow can create a lot of mischief with it. Use access control list on the mailbox for better security.

RWAST.EXE will kill a process even if it is in RWAST state, something "STOP PROCESS_NAME" won't do.

FORCEPRV.EXE will enable/disable the current (not authorized) privileges of any process on the system. Its input is set up through RSECURITY.COM. Since the system service, $SETPRV works only on the process calling it, this program uses an execulet to queue a KAST to the target so that it calls the service. You need CMKRNL privilege to use this program. It will enable/disable ANY current privilege for ANY process on the system since the process will run it in KERNAL mode. Be careful.

SUSPEND.EXE will suspend and resume another process.

UNZIP.EXE is the standard VMS unzip program.

VERB_UTILITY.EXE is someone's program that displays and creates a Command Definition input file from an existing command. It's useful for creating a Command Definition Language input file of your own. It puts a .CLD file in TEMPD: that you can rename and edit.

****
4. EVE Editor
****

This segment describes how we have gone about setting up modifications to the VMS standard editor for our own use. You may modify the files to suit yourself.

The VMS default editor is called "EVE" for "Extensible Versatile Editor" (cute). It is based on the DECTPU (Text Processing Utility) written by DEC. In other words, EVE is written in TPU and has key definitions that execute TPU procedures. When you type "EDIT" at the command line, EVE is what you get.

There are two logicals we define in the login processes before EVE is evoked:

```
$ DEFINE TPU$SECTION R$:[EVE]EVE.SECTION
$ DEFINE TPU$COMMAND R$:[EVE]EVE.TPU
```

The above should be changed to point into your own directory structure so that you have your own procedures and definitions.

And two symbols:

```
$ SEDIT :== @R$:[COM]SEDIT.COM
$ E :== @R$:[COM]EDITTP.COM
```

The above should be defined by your login procedure just as shown. The first starts up EVE as a sub process. The second edits a file of your choosing.

These must be defined before you can customize EVE and use it as we do.

To actually perform the customization steps, do the following:

1) Define TPU$SECTION and TPU$COMMAND to point into your area.

2) Define SEDIT and E symbols

3) Copy over R$:[EVE]EVE_KEYS.EVE to your area. Edit it (if you can) to make changes appropriate to your needs or desires.

4) Start EVE by typing 'EDIT' at the DCL command line. Hit the DO key, PF4, and at the editor command line type: @[your location]EVE_KEYS.EVE. This will load and execute the EVE commands in that file. This file defines your keys.

Note! If you have defined procedures, they may be run now.

5) Issue the command: SAVE ATTRIBUTES, (F9, using my definitions). This will save all the changes into the binary section file which is loaded at startup.

The flow at startup is for EVE to: load the binary file defined in the logical TPU$SECTION followed by the startup ASCII file defined in the logical TPU$COMMAND.

One of the things we've done is to set things up to aid in using the editor in the real world environment. We run EVE as a named sub process. This is done by issuing the command 'SEDIT' which will run the DCL procedure R$:[COM]SEDIT.COM. You should execute this procedure in your startup procedure. It is shown below.

```
$ N = F$PROCESS()
$! unique to each process
$ DEFINE/JOB EVE_TEMP R$:[TEMP]     (use your temp area)
$ SPAWN/PROCESS=EVE_'N' EDIT
```

When you are in the parent process at the DCL command prompt and wish to edit a file, you may type "e 'file_spec'". "e" is defined as the command procedure @R$:[COM]EDITTP.COM. This file attempts to resolve the entered file specification including supplying device and directory of the default directory at the time of entering the command. This is provided you did not provide this information. If not found in your default directory, it then searches a list of areas on the system (which you can modify) and, if found, inquires if that is the file you wish to edit. As a last test, it opens the file and then closes it to make sure the specification is correct and you have at least read access to it. EVE will tell you if you are "Read Only". If all fails, it types an error message.

The command "e" executes the following DCL command procedure named R$:[COM]EDITTP.COM:

```
$ IF P1 .EQS. "" THEN EXIT
```

```
$ F = F$SEARCH("''P1'")
$ IF F .EQS. ""
$ THEN
$   LOC1 == "DKA100:[000000...]"
$   LOC2 == "DKA200:[000000...]"
$   LOC3 == "VDA0:[000000...]"
$   LOC4 == "SYS$LIBRARY:"
$   LOC5 == "SYS$MANAGER:"
$   LOC6 == "SYS$SYSTEM:"
$   LOC7 == "SYS$UPDATE:"
$   LOC8 == ""
$   C = 0
$   CONTEXT = 0
$ SLOOP:
$   C = C + 1
$   PREFIX = LOC'C'
$   IF PREFIX .EQS. "" THEN $ GOTO BADF
$   F = F$SEARCH("''PREFIX'''P1'",CONTEXT)
$   IF F .NES. ""
$   THEN
$      WRITE SYS$OUTPUT "Found: ''F'"
$      INQUIRE YN "Edit this file"
$      IF YN THEN $ GOTO END_LOOP
$   ENDIF
$   GOTO SLOOP
$ ENDIF
$ END_LOOP:
$ OPEN/READ/ERROR=BADF FILE 'F'
$ CLOSE FILE
$ ETP == "$R$:[EXE]EDITTP.EXE"
$ N = F$PROCESS()
$ ETP 'F'
$ ATTACH EVE_'N'
$ EXIT
$ BADF:
$ WRITE SYS$OUTPUT "File not found: (''P1')"
$ EXIT
```

Next it runs a program, R$:[EXE]EDITTP.EXE, that sends information to POKE.EXE which in turn dumps a key sequence into the typeahead buffer of the terminal after you have attached to the editor sub-process that will open the file you indicated for editing. It uses SENDMAIL in MARLIB.

SEDIT.COM defines the "ENTER" on the ten key pad to attach to your eve process. Note: your eve editor key definitions must have PF4 defined as the "DO" key for all this to work.

The entire process makes it possible for anyone to run EVE in any number of processes and sub-processes and still keep everything straight.

Below are my key definitions. Operations following the "/" are the GOLD operations. It should serve you as a starting point.

```
            EVE Editor
(ten key pad)
Find/Find Wild          Find Selected     Find Next   Do
Select                  Del L/Rest L      Cut         Replace
Start of Line/Open Line Del Char    Copy
End of Line/Mark  Write Buf/Go To   Paste
```

Next Buffer Gold Attach Parent/Sh Key

(six key pad)
(insert) Top(home) Page UP
Delete Selected/Restore(delete) Bottom(end) Page Down

F1 = Help
F2 = Set Right Margine 80/132
F3 = Uppercase Word/Lowercase Word
F4 = Quit
F5 = Fill Paragraph/Capitalize Word
F6 = (Leave this alone and don't use it)
F7 = Show Buffers/Buffer Messages
F8 =
F9 = Save Attributes
F10 = One Window
F11 = Next Window
F12 = Two Windows

We also have a command procedure called SCREEN.COM that will ask for a full command to be executed. You would use this procedure to process a command that will likely produce more than a page of output and you would like to be able to go back and forth or search..

The command is piped into a unique file name in the temporary directory which is then either edited with your EVE editor (if you call SCR*EEN and pass it an argument, or will process it with the MOST processor, if you don't supply an argument.

****
5. Data Entry Routines
****

The input routine is especially designed to behave in a way similar to old fashioned heads down data entry on an IBM terminal. You can declare the type of input accepted and the minimum and maximum length. Errors are flagged immediate for correction, there is no autotab (extra characters are simply not echoed and ignored and escape sequences are swallowed whole and ignored. The tab character is interpreted as "close the field and move to the next field". There are only two edit operations: 1) CTRL U which means start over and the backspace key to erase the characters to the left. The paint character is "." and the input and output can be masked, e.g. "(...) ...-...." for phone number.

Post processing such as compression and parsing with capitalization with appropriate re-display is available. Numeric input can be re-displayed right justified, right justified with a period and right justified with period (dollars) and commas.

SETLANG(^VAL(opt)) is used to declare which language you are using. The default is 0 for C++, 1 means DBL. This is done to allow the programmer to use the default argument passing for each language. The value opt must be passed by value. This could expand in the future for different languages.

SET_COLORN(int)

These two color routines use the <ESC>/[38;2;r;g;b;48;2;rgbm/ escape sequence to set the text color to one of the colors below. The background color is set to BLACK with the exception of BLACK and WHITE.

```
DEFAULT
RED
LTRED
GREEN
YELLOW
BLUE
LTBLUE
MAGENTA
CYAN
WHITE
NORMAL
BLACK

 SET_COLOR(char*)

This version allows you to put your own escape sequence into the argument as a
string. The escape sequence is very much more flexible than just color and you
can do all sorts of things with it.

Arg1 is a foreground/background color code set by the ANSII standard.

There is a more detailed description of these routines as part of the
description of MARLIB.MAR found is section 6.

INPUTM(char*, int, int, int, int, int, int, char*)
      arg1 is an imput buffer, max size = 80 characters
      arg2 is the input type:
            VAL[ID]     = 1          any printable character whose value <= 32
            ALPHA = 2          Alphabetic plus space
            NUMB  = 3          number or "-", left justified
            ANUM  = 4          ALPHA + NUMB
            RJINT = 5          NUMB right justified
            DLLR  = 6          NUMB right justified + "."
            DLLRC = 7          NUMB right justified + ","
      arg3 is the minimum
      arg4 is the maximum
      arg5 is the X coordinate (row)
      arg6 is the Y coordinate (COL)
      arg7 is the option:
            COMP  = 1          eliminate multiple spaces
            CAP   = 2          capitalize the first letter in the input
            PAR   = 4          Parse and capitalize each word
            UP    = 8          all letters upper case
            DW    = 16         all letters lower case
      arg8 is the mask or ""         "(...) ...-...." for phone

DISPLAYM(int, int, char*, int, int, char*)
      arg1 is the x (vertical) screen position (1 = top of screen),
      arg2 is the Y position (1 = left).
      arg3 is the address of the string (or descriptor) to be displayed.
      arg4 is the characteristic
            NORM  = 4    Normal
            BOLD  = 1    Bold
            BLINK = 2    Blink (not always available)
            REVERSE     = 3    Reverse
            KEEP  = 8    The screen reverts to NORM unless
                         the value KEEP is ored into the arg.
      arg5 is the screen clear option:
            0           = 0   Do nothing
```

```
            EOL         = 1   clear to end of line
            EOS or EOFS = 2   Clear to end of screen
            ALL         = 3   Clear entire screen
            RESET       = 4   Reset the terminal
      arg6 is the mask, e.g. "(..) ...-...." for a phone number or "".

The screen clear options are performed after the buffer is displayed. Therefore
you should do a "clear the whole screen" in a separate operation.

DISPLAYN(int, int, int, char*, int, int)
      arg1 is the X coordinate.
      arg2 is the Y coordinate.
      arg3 is the number of characters to be overwritten with blanks.
            this is for re-displaying input.
            It can be zero if not needed.
      arg4 is the data string (descriptor).
      arg5 is the right position where the display is to end (right margin)
      arg6 is a number for the mask to use:
            DNINT = 1          just numbers
            DNDLR = 2          number + period, "0.00"
            DNDLRC      = 3          DNDLR + commas

This displays numbers right justified.

INMESSAGE(char*, int, int)
      arg1 is the address of the buffer (or descriptor) that receives the
      possible input.
      arg2 is the type
            MESNOIN is for messages that do not expect a value.
            MESIN expects a numeric value to be returned in arg1.
            MESINA expects an ALPHA value to be returned.
      arg3 is the index into an internal table of messages.
            if they are warning messages, "Hit return to continue is
            automatically handled.
      The messages are:
            HRTC  = 0    "Hit return to continue."
            TSH   = 1    "Too short."
            BAD   = 2    "Bad character for this type."
            CWN   = 3    "Change what number ?"
            SEL   = 4    "Selection ?"
            RNF   = 5    "Record not found."
            RAE   = 6    "Record already exists."
            RLC   = 7    "Record locked. Try again[y/n] ?"

Additional details will be found in section 6.

************************************************************************
6. THE MACRO LIBRARY, MACROLIB.LIB & MACROLIB64.LIB
************************************************************************

        Macros for VMS MACRO32 & MACRO64

The VMS assembly language is a powerful set of complex instructions which, when
combined with the library of system services (executive images), allow the
competent programmer to write code that can do just about anything for which the
author has privilege. Although a dying art, assembly language programming has
its place for times when the maximum efficiency, speed and compactness is
needed. In these cases, the skill of the programmer and his knowledge of how the
assembly instructions work, can produce results about which the high level
```

language programmer can only dream.

I remember back at datanex, Inc. our company was writing EZBRIDGE for transferring files between a VAX and an AS400. This required translation of ANSII to EBCDIC and back. They wrote it in "C". Translation took forever (seconds) and really slowed the file transfer process noticeably. I wrote a routine in MACRO32 they could call which required one instruction to do the translation for one character. They were amazed.

A programmer will likely choose assembly language for so called systems programming wherein certain abilities not provided by the operating system and its executive images are needed. These cases, which are by far the most common out in the real world, do not really require compactness and speed. Developed system programs usually are used infrequently. What is really needed is maintainability which means readability. The macros described below provide this.

Now I've heard all the arguments about how "unnecessary" all of this is and how it just adds more "complexity" to the process. Back in the days before VAX, we were writing business programs in DBL from then named DISC (now SYNERGEX) and had the usual spaghetti code and associated headaches. I wrote a pre-compiler program that used exactly the structured approach described below. Ken Lidster hated it but eventually introduced CASE and IF/THEN/ELSE structure to DBL.

My own experience was that, as soon as my team got over the initial shock of something new, they loved it. Code became readable. Bugs were dramatically reduced. Bugs were more easily found and productivity quadrupled in a matter of a couple of months. So before you dismiss my approach out of hand, read some of my code.

*****

We have designed these macros so that they appear to look alike and function the same for both MACRO32 and MACRO64 The differences are:

1) MACROLIB64 uses three scratch registers whose contents are unknown to the programmer after the invocation. They are: R0, R1 and R23. MACROLIB32 does not alter registers.

2) In MACROLIB64, branches can take you anywhere so the "W" (word branch) form available in MACROLIB32 isn't used.

3) MACROLIB64 adds the letter "Q" to the comparison size; B, W, L, and Q.

The macros provided are designed to turn assembly language into a structured language in that there are no visible branch/jump instructions in the program and, except for the entry name, no labels are required. A program might have the rough appearance of:

```
.ENTRY  NAME
INSTRUCTION(S)
REPEAT
        INSTRUCTION(S)
        UNTIL   condition
        INSTRUCTION(S)
        IF      condition
                INSTRUCTION(S)
        ORIF    condition
                INSTRUCTION(S)
```

```
        ORIF    condition
                INSTRUCTION(S)
        ELSE
                INSTRUCTION(S)
        ENDIF
        INSTRUCTION(S)
        WHILE   condition
        INSTRUCTION(S)
ENDR
INSTRUCTION(S)
RET

.END NAME
```

Repeats can be nested indefinitely and if/orif/else/endif to twenty-five.
Actually, any depth greater than about five renders the code unreadable anyway.

The following is a list of the macros and their arguments:

REPEAT
    This forms the beginning of a loop. It takes
    no arguments and does not produce any code.

UNTIL ARG1, condition, ARG2, LENGTH
    Until the condition is met, repeat the loop (go back to the beginning).

        ARG1 and ARG2 are any valid location
        in any valid addressing mode.
        Condition is the comparitor. It can be:

                EQL
                NEQ
                GTR
                GEQ
                LEQ
                LSS

        Each of these may have a 'U' appended for testing
      unsigned. It would be meaningless for EQ & NE.

        LENGTH is the address size. For example:
        UNTIL @(R0), GTR, R10, W
        will compare words. Options are:
      B, W, L and for Macro64, Q

WHILE   ARG1, condition, ARG2, LENGTH
        While the condition is met, continue execution.
        Otherwise break out of the loop and resume
        execution after the ENDR.

BREAK
        Unconditionally breaks out of the loop. Handy
        when you do an IF and some stuff and then
        terminate the loop.

CONTINUE
        Unconditionally branches to the beginning of the loop.

ENDR

The end of the loop. If execution reaches here,
it branches back to the REPEAT of the same level of
nesting.

IF ARG1, condition, ARG2, LENGTH
If condition is true, execute the code between
the IF and the next ORIF/ELSE/ENDIF.

ORIF ARG1, condition, ARG2, LENGTH
If condition is true, execute the code between
the ORIF and the next ORIF/ELSE/ENDIF. Multiple
ORIF's are possible between an IF and ENDIF.

ELSE
Execute the code between the ELSE and the next
ENDIF. An ORIF after an ELSE makes no sense
and will never be reached. It also won't compile.

ENDIF
Terminates this level of the IF sequence. Required for each level of IF.

Discussion:

For the sake of efficiency, if the ARG2 value is identical to #0, a TST instruction is used rather than the CMP for MACRO32. For MACROLIB64, it is a simple branch on register test.

There are versions of the macros where no CMP or TST instruction is produced but rather a branch is created that assumes an instruction has just been executed that sets the condition codes. The macros are:

MACROLIB32:

```
IFC     condition
ORIFC   condition
UNTILC  condition
WHILEC  condition
```

MACROLIB64:

```
IFC     condition,register
ORIFC   condition,register
UNTILC  condition,register
WHILEC  condition,register
```

In other words, those macros that have ARG1 and ARG2 arguments have this form as well.

The following macros may have the letter W appended to the names in order to produce a word branch rather than the default byte branch. Compile errors that indicate branch out of range may be solved by appending the W: This is not part of MACROLIB64.

```
UNTILW
UNTILCW
WHILEW
WHILECW
BREAKW
CONTINUEW
ENDRW
IFW
```

```
        IFCW
        ORIFW
        ORIFCW
        ELSEW
```

Additionally, we have IF_ERR and IF_OK followed by ENDIF for checking the low order bit of R0 after a call or test.

These macros, while very efficient, cannot handle all cases where there is a strong need for compactness and/or speed. But they more than make up for this in readability and maintainability.

Other macros of interest:

The following macros call the library routines which lock and unlock the entire image. On the AXP system, this is usually the most safe and efficient way of locking code in memory because you have to include the linkage psect(s) which can be difficult to do.

```
ILOCK       locks the image and prevents page faults.
IUNLOCK     unlocks the image.
```

```
Error checking with ERCK     [string],[destination],[branchword]
```

Error checking usually looks something like:

```
      IF_ERR
            JSB   DSPERR
            RET
      ENDIF
```

or something to that effect. We have a more generalized macro that takes arguments and does some (but not all) operations needed. The macro always produces the "IF_ERR" and ENDIF" pair. The first argument is a string of characters (upper case) that can contain the following letters in any order:

```
[D],B,R,S
```

B, R and S are mutually exclusive in that B produces a BRB to the second argument. R produces a RET and S produces a RSB. The D does a JSR DSPERR in MARLIB followed by MOVZBL #SS$_NORMAL, R0. If a third argument is provided and a B is used, the branch will be a word branch.

If no arguments are given, an RET is produced.

****

RMS (Record Management System)

RMS is one of the best pieces of software DEC wrote. It is awesome. It is jaw droppingly versatile and supports every type, style and variation of file you can imagine and a few you can’t.

It is also horrifyingly complex. While the documentation defines the variables, it does not explain much about their use and which ones go together to produce a given result. Examples are both sparse and esoteric, leaving the user to either drop back to a high level language or find his way alone. It took me days of reading, searching and just plane trial and error to figure out how to read and write a simple variable length text file in MACRO32.

```
OPENR name, address of file spec descriptor, address of record buffer
      descriptor, [address of default file descriptor]
REOPENR     Same as above
OPENW name, address of file spec descriptor, address of record buffer
      descriptor, [address of default file descriptor]
REOPENW     Same as above
OPENI name, address of file spec descriptor, address of record buffer
      descriptor, [address of default file descriptor],
      [<OR'ed additional operation bits enclosed in angle brackets>]
READ  name, address of record buffer descriptor
WRITE name, address of record buffer descriptor
CLOSE name
ERASE name
```

That is exactly what the above macros do. They take care of all structures ($FAB, $RAB) and put them in the PF$DATA psect, same as the PRINTF macro described below. The OPENI is for ISAM files and the record operations variable are the standard: RLK ASY and SYNCSTS bits needed to update with record locking. The OPERATION argument allows the user to add bits for special purposes (see LOCKHOLDER.MAR).

When you reopen a file, please use exactly the same name and arguments or things will get confused.

If special operations are required or information needed, the naming convention for the $FAB and $RAB is: the name you give as the first argument of the macro with an "F" appended for the $FAB and an "R" appended for the $RAB.

*****

The macro PERR takes one argument, a single character and calls SERR. R0 is used and preserved. It writes to the system error log and works in kernel mode.

The following macros make it a bit easier to use the LIB$SYS_FAO library call or the call to DECC$GPRNTF in the MACRO64 world.

```
PRINT <delimited character string>
PRINT <delimited character string>,\MN    ; MACRO64
```

Stores the character string in a psect called PF$DATA. Can be used anywhere in a CODE PSECT because it saves and restores the current PSECT.

```
PRINTF      <delimited character string>,arg1,arg2.... or
PRINTF      <delimited character string>,\MN,arg1,arg2....  ; MACRO64
```

Stores the character string in a psect called PF$DATA. Can be used anywhere in a CODE PSECT because it saves and restores the current PSECT. In MACRO32, the arguments replace FAO arguments embedded in the string (see below for some FAO arguments).

For MACRO64, the character string uses "C" formatting.

!AC Inserts a counted ASCII string. It requires one parameter: the address of the string to be inserted. The first byte of the string must contain the length (in characters) of the string.

!AD Inserts an ASCII string. It requires two parameters: the length of the string and the address of the string. Each of these parameters is a separate argument.

!AF Inserts an ASCII string and replaces all non printable ASCII codes with periods (.). It requires two parameters: the length of the string and the address of the string. Each of these parameters is a separate argument.

!AS Inserts an ASCID string. It requires one parameter: the address of a character string descriptor pointing to the string. $FAO assumes that the descriptor is a CLASS_S (static) or CLASS_D (dynamic) string descriptor. Other descriptor types might give incorrect results.

!AZ Inserts a zero-terminated (ASCIZ) string. It requires one parameter: the address of a zero-terminated string.

!XL Converts a long word value to the ASCII representation of the value's hexadecimal equivalent. It requires one parameter: the value to be converted.

!ZL Converts an unsigned long word value to the ASCII representation of the value's decimal equivalent. It requires one parameter: the value to be converted.

There are a lot more. See the Systems Services Reference Manual under $FAO.

In addition, for MACRO64, we have PUSHR and POPR macros which take an identically ordered string of comma separated registers to be pushed and later popped onto the stack. This is an easy method of saving a few registers temporarily. It is slightly more efficient to "roll your own".

PR_C This macro takes the argument of one of the colors as defined in SMAC and causes the escape sequence of that color to be placed where R10 points. It uses the subroutine INSERT_COLOR in MARLIB.

****
7. Functions/Subroutines in MARLIB.EXE (DLL)
****

MARLIB is a shared executable image (Library, Dynamically Linked Library, DLL) containing multiple position independent re-entrant routines that can be linked to programs written in any language. Symbols are defined by including SYS$SHARE:MARLIB/SHARE as a line in the .OPT file used in linking. The actual linkage is accomplished by the image activator at run time.

The following (leading with a semicolon) is a direct copy from the MARLIB.MAR source code.

```
;
;     library routines to be linked to macro programs
;
;     List of routines and calling conventions
;
;     Note!       The CRLF, HIT, PUT_NUM and PUT_TWO routines
;           will only work in user, super and exec mode
;
;     CRLF
;           Print a blank line.
;           Input: none
;
```

```
;     RCONT
;           Prompt "Hit return to continue: "
;           Input: none
;
;     HIT
;           print "Hit" on the screen
;           Input: none
;
;     PUT_NUM
;           Output a hex value on the screen.
;           Input:      R4 - binary value
;
;     PUT_TWO
;           Output two hex numbers on the screen.
;           Input:      R1 > 0 - Indicate a change
;                 R4 - first value
;                 R5 - second value
;
;     SERR
;           Send an error message to the error log
;           Will print a single character in the message
;           so that you can trace them more easily.
;           Input: R0 - Enter with the character in R0.
;
;     SERRH
;           Send an error message to the error log.
;           Input:      R0 - hex value to be included.
;
;     FIXUP
;           Fix up the addresses of the linkage psect
;           after copying to high address.
;           Input:      R0 - base address (x1000) usually
;                 R1 - key, neql zero means print out
;                       all the values in the linkage
;                       psect. BITT #2 set means write
;                       to errlog.
;                 R2 - address of beginning of linkage
;                 R3 - address of end of linkage
;                 R10 - amount of fixup
;
;     FIXUPD
;           Fix up the addresses in the linkage psect
;           of descriptors which have been copied there.
;           Input:      R0 - pointer to pair table
;                 R10 - amount of fixup
;
;     NUM_OUT
;           Convert a value to decimal or hex
;           and put it in the output. Best used
;           in this library.
;           Input:      R3 - neql zero means hex out
;                 R4 - value to be output
;                 R2 - address of output descriptor
;
;     ATOI
;           String converted to integer.
;           Input:      R2 - address of input descriptor
;                 R3 - neql zero means string is hex
;                 R4 - address of long output value
```

```
;
;     ATOQ
;           String converted to integer.
;           Input:      R2 - address of input descriptor
;                 R3 - neql zero means string is hex
;                 R4 - address of quad output value
;
;     DSPERR
;           Display the error message associated with the
;           error value in R0.
;           Input:      R0 - System generated or user supplied
;                       standard error value.
;     SLEEP
;           Sleep for n milliseconds.
;           Input:      R0 - number of milliseconds
;
;     SCAN_PCBS
;           Scan the PCB table and execute a user supplied
;           routine for each one found. It will scan the
;           entire large table calling the user supplied
;           routine only for PCB's that are valid, i.e. not the
;           SWAPPER and are not DELETE PENDING. The user routine
;           can terminate the scan by returning lbc in R0
;           Input:      R0 - > 0 means use $CMKRNL rather than $CMEXEC
;                 R9 - address of user subroutine
;           Output:     R4 - a valid PCB
;           Comments:
;                 The user supplied routine is .JSB32 and must
;                 pay particular attention to register usage.
;                 Its address is placed in R9 for use herein.
;                 When called, R4 will contain the address of the
;                 PCB structure for a process. Do what is needed.
;                 R6 - R10 must be restored. R0 is the return
;                 status.
;
;                 If you choose to use Kernel mode, the image will
;                 be locked in memory and run at IPL SCHED (8) for
;                 the duration. Use this only if it really is needed.
;
;     XLATE
;           Logical name translation with index
;           Input:      R3 > 0 means the string should be interpreted as hex
;                 R5 = index (start with zero) and increment in subsequent calls.
;                 R6 = Pointer to name table descriptor.
;                 R7 = Pointer to logical name descriptor.
;                 R8 = Pointer to result descriptor.
;                 R9 = Pointer to long numeric value
;                       make R9 zero to skip
;           Output:     results pointed to by R8 and R9
;
;     PARSEM
;           Parse a string into it's pieces based on one or more separators.
;           A string  containing separators may be enclosed in double quotes.
;           PARSEM checks to make sure the length of the output is not exceeded
;           so be sure to reset a descriptor length that is reused.
;           Input:      R7 - Descrip;tor address of separators
;                 R8 - Descriptor address of string to be parsed
;                 R9 - Descriptor address of output piece
;           Output:     R10 - index into the string for next call.
```

```
;                       Start with zero and leave alone.
;                 R0 - lbs = more available
;
;
;     SENDMAIL
;           Write a message to the mailbox created by POKE in order
;           to get material put into the type-ahead buffer of a terminal.
;           Input R10 - points to the descriptor of the message to be written.
;           Output      R0 - status
;
;     FIND_LDRIMAGE
;           Finds the S0 address of AST_LEI that has been loaded into
;           the space for system api's. We index a name off the returned
;           address to get the address of the special code to queue an
;           AST to the target.
;
;     GETCLITEMS
;           Use CALLS form with a variable stack depending on how the items
;           are to be obtained.
;           The fist argument on the stack is the item list. The item
;           list consists of multiples of two long words, the first being the
;           type of conversion to be done in the lower 16 bits and the buffer
;           size in the upper
;           16 bits for string output.. The second long word is the address of
;           either the descriptor of a string to receive the command line item or
;           the address of a long word storage for the binary value of the input
item.
;           e.g.
;           .word 1
;           .word string size
;           .address    string descriptor
;           .long 3
;           .address    long word storage address
;           .long 0     ; item list terminator
;
;           The routine will process all of the items. If a string exceeds the
length in the
;           string descriptor, it will be truncated. If less than the value in the
string descriptor,
;           the descriptor length value will be adjusted.
;           If the command line contains fewer items than the item list, descriptor
length
;           values of the missing values will be zeroed or the binary values will
be zero.
;           Excess command line items will be discarded.
;
;           types are:
;                 1 = string
;                 2 = terminal name or epid. Uses GETPID or a variation to return a
;                       valid EPID or zero
;                 3 = decimal long word
;                 4 = hexadecimal long word
;                 5 = decimal octa word
;                 6 - hexadecimal octa word
;
;           The first argument  on the stack is the item list address. This
;           is required. The next and optional argument is the address of a prompt
descriptor
;           that will be used if no command line items are provided. This argument
```

```
;           must be zero if you push a third argument onto the stack. The third
argument
;           is the address of a string descriptor that will serve as a command line
input.
;
;           So push arguments in the order: [command], [prompt], item list.
;
;           Command line arguments are separated by commas or spaces.
;
;
;       P2_COMMON
;
;       There are many ways for the programmer to extend his address space for a
;       program. These methods return the space during image rundown.
;       Global Sections
;       are another way to get extra address space and can be made permanent so
that
;       other processes can map to them.
;
;       The code in this subroutine takes its space from the caller's P2 space, P2
space
;       being unique to alpha and integrity computers. It has a huge address space
which
;       can be used with little chance of interfering with anything else going on
with
;       the process or operating system. It is private to the process and survives
image
;       rundown so that other programs run in the same process can access it so
long as
;       the process survives.
;
;       The advantages of this subroutine are that is uses private P2 space, it is
;       convenient in that the space can be set up as "records" and the routine
will
;       even parse these records into variables. In addition, addressing in P2
space is
;       a little tricky and unless the programmer is familiar with MACRO64, can be
;       frustrating. This routine make this transparent to the programmer.
;
;       By default, the caller is given one alpha CPU page of space (usually 8192
;       bytes) that he can use as desired. If more space is needed, up to
;       about 1X10^18/cpu-page-size pages or approximately two billion pages
;       can be requested. We also have a scheme that will allow the caller to break
the
;       space into indexed "records" and those records may also be used to pass
comma/space
;       separated variables that can be evaluated by a reader program using an item
list
;       identical to that used by GETCLITEMS.
;
;       Note: we do not check for overflow of your allocated space. Things are done
;       in user mode and we don't care if you crash your program because you can't
;       add.
;
;       Argument structure and explanations:
;
;       A MACRO is provided, P2_STRUCTURE, whose address is passed to the
;       P2_COMMON subroutine. The arguments are keyword with all the items
;       defaulting to zero except the pages requested which defaults to one.
```

```
;       All the names for the arguments are the names of the variables minus
;       the “P2_” part.
;
;       Structure label:
;       P2_STRUCTURE    -
;           P2_RW:      .LONG 0     ; RW = 0 ; 0 for write, 1 for read
;           P2_INDEX:   .LONG 0     ; INDEX = 0 ; record number
;           P2_LENGTH:  .LONG 0     ; LENGH = 0 ; record length
;           P2_TEXT:    .ADDRESS    0 ; TEXT = address of descriptor
;                                     ; Address of a descriptor pointing to a
;                                     ; record which will be read or written.
;                                     ; Use even if the record is fixed field.
;           P2_ITEMS:   .ADDRESS    0 ; ITEMS = address of table
;                                     ; Address of an item list designed like that
;                                     ; for GETCLITEMS. Use zero if you are not
;                                     ; going to parse the record for
;                                     ; comma/space delimited variables.
;           P2_PAGES:   .LONG 1     ; PAGES = 1 ; Number of cpu pages to be
;                                   ; allocated.
;           P2_FIXED:   .LONG 0     ; FIXED = 0 ; 0 = text, 1 = fixed field
;           P2_BASE_ADD .ADDRESS    0 ; BASE_ADD = 0. address of QUAd word
;                                     ; for returned QUAD word base
;                                     ; address of the allocated space
;
;           Another MACRO, P2_LOAD_ADD, will allow the programmer to load
;           a register with the correct address of the base address which
;           was placed into the QUAD storage location addressed in P2_BASE_ADD.
;           Simply using MOVL will not do it correctly. The resulting
;           register can then be used to directly address the P2 storage
;           allocated.
;
;           P2_LOAD_ADD ‘QUAD BASE_ADDRESS_STORAGE ADDRESS’, REGISTER
;
;           P2_STORE_ADD allows the programmer to store a 64 bit address
;           in a register into a QUAD memory location.
;
;           P2_STORE_ADD      ‘QUAD STORAGE ADDRESS’, REGISTER
;
;           Incrementing the register containing the base address will
;           work well for most operations. The base address is almost always:
;           ^x00000000 80000000 giving you about two billion bytes of space
;           before 32 bit operations will get you into trouble with automatic
;           high order bit extension. Another MACRO, P2_ADD_QUAD can be used on
;           the base address register to avoid any complications.
;
;           P2_ADD_QUAD ‘ADDRESS OF QUAD VARIABLE CONTAINING AMOUNT’, REGISTER
;
;       CVTEPID
;           Converts a string containing either a terminal name or a hex number
;           into an EPID and checks the validty. Used by GETPID and GETCLITEMS.
;           Input:
;                 R4 - Address of the string descriptor.
;           Output:
;                 R4 - EPID or 0
;
;       GETPID
;           Uses LIB$GET_FOREIGN to get either a PID or a terminal name.
;           iT calls TTOPID. Otherwise it assumes a hexadecimal EPID.
;           In either case, it calls CHECKPID in EXEC mode to see if the EPID
```

```
;           is valid.
;           Input:
;           Output:     R4 - EPID
;
;     TTOPID
;           Get the EPID from a terminal name
;           Input:
;                 R3 - Not zero outputs the EPID in hex on your terminal.
;                 R4 - Address of the terminal name descriptor
;           Output:
;                 R4 - binary EPID value
;
;     CHECKPID
;           Get the process PCB from EPID to see if it is valid EPID.
;           Input:      EPID as the single argument in the call.
;           Output:     R0 as the status of the test.
;
;     CHECK_TERM
;           This is used by POKE and MENU to determine if the process running
;           or addressing them is logged in over DECnet. These programs call
;           POKE or in some way go into KERNAL mode to stuff strings into the
;           typeahead buffer. If you are logged in over DECnet, it will crash
;           the system, guaranteed.
;           Input:
;                 R9 - Address of terminal name descriptor.
;           Output:
;                 R0 - Least significant bit set if not the RTxy: terminal
;
;     GRAB_DEVICE
;     GRAB_TERMINAL
;           This routine was created from a program designed to put strings into
;           another process' terminal type-ahead buffer. It does a number of tests
;           checks and gains locks and raises IPL. To be consistent and allow
;           proper error passing, we decided to put the guts here and allow the
;           user to CMKRNL to it. It will either do it right and then process
;           the way you desire or it won't and will pass back an error message.
;
;           Since this stuff can crash a system pretty easily, we wanted to be
;           consistent. It is called by naming the routine in a $CMKRNL call
;           passing it a list af arguments described below.
;
;           $CMKRNL_S -
;                 ROUTIN = GRAB_DEVICE, -
;                 ARGLST = KARGS
;           ERCK
;...
;           KARGS:
;                 .LONG 1/2/3
;                 .ADDRESS    xxx   ; Address of terminal descriptor
;                 .ADDRESS    yyy   ; Address of the .JSB routine
;                                   ; you have provided to do something
;                                   ; with the device you have captured.
;                 .ADDRESS    zzz   ; Address of string to stuff descriptor
;                                   ; If yyy is the address of TH_STUFF,
;                                   ; zzz is provided
;                                   ; and you call GRAB_TERMINAL,
;                                   ; we do it all for you.
;
;           If the number of arguments is "1", it will test to make sure the
```

```
;           device can be grabbed. If "2", it will grab and execute your
;           routine. If "3", it will grab the device, make use of the third
;           argument and "yyy" must point to "TH_STUFF". The last part works
;           for GRAB_TERMINAL only. GRAB_DEVICE takes a maximum of two arguments
;           and does not make the UCB length test we use for terminals.
;
;           For your own routine, GRAB_DEVICE will give you a pointer to the
;           device's UCB in R5.
;
;     GRAB_PROCESS
;           For the same reasons as GRAB_DEVICE, we standardized getting control
;           of a process so you can do some stuff to it. This requires you to
;           separate copying stuff into S0 space from getting control of a process.
;           Like GRAB_DEVICE, if the number of arguments passed is "1", the process
;           will be tested. If "2", the routine will execute the process pointed to
;           by the second argument.
;
;           KARGS:
;                 .LONG 1/2
;                 .ADDRESS    xxx   ; Address of the EPID long word
;                 .ADDRESS    yyy   ; Address of the subroutine you wrote
;
;           Unlike GRAB_DEVICE, nothing is passed in R5. However, the internal PID,
;           IPID, is passed in R6, and the callers PCB is passed in R4. Write your
;           subroutine with R4 and R6 as input and preserved.
;
;     GRAB_KAST
;           For the same reasons as GRAB_DEVICE, we standardized on calling code
;           to cause a KAST in AST_LEI to be queued to another process. This
;           does not require a subroutine, rather it requires the use of the
;           macro KAST_OFFSETS and a KAST_ARGUMENTS MACRO as follows:
;
;           .MACRO      KAST_ARGUMENTS
;           KARGS:      .LONG 7
;           KRT:  .BLKL 1           ; KAST routine offset
;           STS:  .BLKL 1           ; status of operation (like IOSB)
;           SAN:  .BLKL 1           ; Sanity check (number of images in process)
;           PID:  .BLKL 1           ; EPID of target process
;           DST:  .BLKL 1           ; Destination string (not descriptor)
;           ED:   .BLKL 1           ; Used in SETPRV
;           PMV:  .BLKL 1           ; Used in SETPRV for privilege mask (octa)
;           PMVH: .BLKL 1
;
;     _KRT = KRT-KARGS
;     _STS = STS-KARGS
;     _SAN = SAN-KARGS
;     _PID = PID-KARGS
;     _DST = DST-KARGS
;     _ED  = ED-KARGS
;     _PM  = PM-KARGS
;     _PMH = PMH-KARGS
;     .ENDM KAST_ARGUMENTS
;
;           Before the $CMKRNL_S call with ROUTIN = GRAB_KAST, move
;           the offset (for GETCMD it is GETCMD_ROUTINE) into KRT,
;           the address of STATUS into STS and
;           the address of the write string where the piggyback KAST will write
;           the results into DST. See GETCMD.MAR as an example.
;           Other locations are filled as needed.
```

```
;
;           You will notice that all the KAST stuff, including AST_LEI, makes
;           reference to KAST_OFFSETS. It is a macro that defines a bunch of
;           stuff including the offsets of the routines you will be calling
;           in S0 space. If you add a routine, do so in AST_LEI.MAR and the
;           KAST_OFFSETS macro in R$:[MAR}MACROLIB.LIB.
;
;           To handle special cases, _SAN or SANITY is entered with the value
;           of zero. If not, it is the address of a JSB subroutine which will
;           be executed prior to the KAST being queued.
;
;     GET_DATE_TIME
;
;           Full CALLS routine to get/print a nice dat and time string.
;           Push the address of a storage location containing the 64
;           bit time to be formatted, or containing zero for current date
;           and time. Then push the address of a storage location containing
;           the pointer to where the formatted string is to be copied. The
;           contents of this location will be updated. If the storage location
;           contains zero, the string will be printed on the screen.
;
;
;     START_STOP_TIMER
;     is a JSB routine that will start a repeating
;     timer that will execute a JSB AST routine in your code.
;     Input:
;           R10 - Descriptor address of a time string of the form
;                 .ASCID "DD HH:MM:SS.CC" for Delta time.
;           R9 - Address of the JSB routine you want executed.
;           R8 - Unique to this timer integer id value
;
;     To terminate the timer, call the routine with the unique id in R8
;     and zero in R9.
;
;
;       GET_PCB_COUNT
;               Full CALLS to get the maximum number of PCB table slots
;               gened into your system.
;
;               4(AP) is the address of a long word to hold the count.
;
;       CALC_PAGES
;               This routine calculates the number of CPU pages (alpha & integrety)
;               that you will need to allocate for P2_COMMON,
;
;               8(AP) is the address of the returned number of pages
;               4(AP) is the address of the total amount of space requested.
;
;*****************************************************************************;***
*****************************************************************************
;
;     Full CALLS calls for displaying data, imputing data and
;     displaying messages are provided. They are intended to be
;     linked into C++ and DBL.
;
;     The input routine is especially designed to behave in a way similar to
;     old fashioned heads down data entry. You can declare the type of input
;     accepted and the minimum and maximum length. Errors are flagged
;     immediately for correction, there is no autotab (extra characters are simply
```

```
;     not echoed and ignored and escape sequences are swallowed whole and
;     ignored. The tab character is interpreted as "close the field and move to the
;     next field. There are only two edit operations. 1) CTRL U which means start
;     over and the backspace key to erase the characters to the left. The paint
;     character is "." and the input can be masked, e.g. "(...) ...-...." for phone
;     number. Post processing such as compression and parsing with capitalization
;     with appropriate re-display is available.
;
;     SETLANG(^VAL(opt)) is used to declare which language you are using.
;     The default is 0 for C++, 1 means DBL. This is done to allow the programmer
;     to use the default argument passing for each language.
;
;     DISPLAYM(int, int, char*, int, int, char* or "")
;     arg1 is the x (vertical) screen position (1 = top of screen),
;     arg2 is the Y position (1 = left).
;     arg3 is the address of the string (or descriptor) to be displayed.
;     arg4 is the characteristic; NORM, BOLD, BLINK & REVERSE.
;           the screen reverts to NORM unless the value KEEP is ored
;           into the arg.
;     arg5 is the screen clear option, usually 0 or EOL, EOS, ALL, RESET
;     arg6 is the mask, e.g. "(..) ...-...." for a phone number or "".
;
; Sometimes things need to be displayed right justified. Right Justified
; data entry makes no sense so we have provided a routine for displaying
; right justified data with one of three possible masks. First is no mask at
; all and can display anything, second is right with a decimal point for
; dollars and third is dollars plus commas.
;
;     DISPLAYN(int, int, int, char*, int, int)
;     arg1 is the X coordinate.
;     arg2 is the Y coordinate.
;     arg3 is the number of characters to be overwritten with blanks.
;           this is for re-displaying input. It can be zero if not needed.
;     arg4 is the data string (descriptor).
;     arg5 is the right position where the display is to end.
;     arg6 is a number (1, 2 or 3) of the mask to use.
;
;     INPUTM(char*, int, int, int, int, int, char* or "")
;     arg1 is the address of the string buffer that receives the input.
;     arg2 is the type of input accepted; ALPHA, NUMB, ANUM or
;           VAL which is any printable character whose ASCII value
;           is at ar above 32 (space).
;           in addition, RJINT will right justify a number,
;           DLLR will add a period and DLLRC will add commas.
;     arg3 is the minimum number of characters accepted.
;     arg4 is the maximum number of characters accepted.
;     arg5  is the x position to start input
;     arg6 is the y position
;     arg7 is the processing option;
;           COMP strip leading/trailing spaces and multiple spaces.
;           PAR parse the string and capitalize each piece.
;           CAP capitalize the ALPHA character in position 1
;           UP make each ALPHA character upper case
;           DW make each ALPHA character lower case
;           arg8 is either a string that contains the input mask or "".
;           The mask contains paint and non-paint characters.
;           The paint character is "."
;           e.g. "(...) ...-...." might be for a phone number.
;
```

```
;     INMESSAGE(char*, int, int)
;     arg1 is the address of the buffer (or descriptor) that receives the
;     possible input.
;     arg2 is the type;
;           MESNOIN is for messages that do not expect a value.
;           MESIN expects a numeric value to be returned in arg1.
;           MESINA expects an ALPHA value to be returned.
;     arg3 is the index into an internal table of messages.
;           if they are warning messages, "Hit return to continue is
;           automatically handled.
;     The messages are:
;     0 "Hit return to continue."
;     1 "Too short."
;     2 "Bad character for this type."
;     3 "Change what number ?"
;     4 "Selection ?"
;     5 "Record not found."
;     6 "Record already exists."
;     7 "Record locked. Try again[y/n] ?"
;
;-------------------------------------------------------------------------
;
*************************************************************************
```

8. Performance Management

The best documentation on this subject, in our opinion, is "OpenVMS Performance Management".

This section contains information that will help a system manager obtain useful information on the running of the system. Help libraries will be of great assistance. This section is mainly a lead-in to setting up a unique batch of processes or actions for the manager to perform.

If you are a software development organization, right out of the box, the two best things to do to improve performance is to disable disk high water marking and to decompress the system libraries.

```
$ SET/VOLUME/NOHIGHWATER_MARKING duxx:
$ @SYS$UPDATE:LIBDECOMP
```

Monitor is very helpful in watching for problems on a running system.

```
$ MONITOR
MONITOR> HELP
```

or

```
$ MONITOR SYSTEM
$ MONITOR PAGE
$ MONITOR PROCESSES
$ MONITOR IO
$ MONITOR DISK/ITEM=QUEUE_LENGTH
$ MONITOR MODES
$ MONITOR CLUSTER
```

The above produce a real-time graphical display of a bunch of interesting system parameters. If you can't be there in person, below is a way of doing things that can produce a file of the same information.

```
$ IF P1 .EQS. "" THEN P1 = "SYSTEM"
$ IF P3 .EQS. ""
$ THEN
$   P3 = "0"
$ ENDIF
$ OPEN/WRITE OF SXMGR:[TMP]M.COM
$ WRITE OF "$ MONITOR/INTERVAL=60/NODISPLAY/RECORD=SXMGR:[TMP]MON.REC-
/ENDING=''P2':''P3' ALL_CLASSES"
$ WRITE OF "$ MONITOR/INPUT=SXMGR:[TMP]MON.REC/DISPLAY=SXMGR:[TMP]MON.LOG ''P1'"
$ CLOSE OF
$ SPAWN/NOWAIT/NOTIFY/INPUT=SXMGR:[TMP]M.COM
```

Accounting can also be helpful. This is especially true for determining the resources used by a particular image. Be careful about using this facility on images. To perform accounting on an image, you must specifically enable it. This is because doing so puts a heavy load on system resources all by itself. Thus you should use it judiciously.

```
$ SET ACCOUNTING /ENABLE=IMAGE
$ WAIT 00:'P1':00.0
$ SET ACCOUNTING /DISABLE=IMAGE
$ ACCOUNTING /OUTPUT=R$:[TEMP]IMAGE.ACT -
/SUMMARY=IMAGE -
/REPORT=(PROCESSOR,ELAPSED,DIRECT_IO,FAULTS,RECORDS)
```

Automatic working set adjustment (AWSA) is a way for VMS to give physical memory to processes which need it and take away physical memory from those that don't (right at the moment). There are two basic approaches to maximizing performance depending on the type of environment (type of user) you have. It is best to quote directly from the above mentioned manual.

3.5.9 Performance Management Strategies for Tuning AWSA

By developing a strategy for performance management that considers the desired automatic working set adjustment, you will know when the AWSA parameters are out of adjustment and how to direct your tuning efforts. Sites choose one of the following general strategies for tuning AWSA parameters:

. Rapid response.

Tune to provide a rapid response whenever the load demands greater working set sizes, allowing active memory reclamation to return memory from idle processes. To implement this strategy: Start processes off with small values for their working set defaults. Set PFRATH low (possibly even to zero). Set a low value for AWSTIME. Set a relatively large value for WSINC. Set BORROWLIM low and WSEXTENT high (even as high as WSMAX) to provide either large working set quotas or generous loans.

This is the default OpenVMS strategy where both BORROWLIM and GROWLIM are set equal to the value of FREELIM to allow maximum growth by active processes, and active reclamation is enabled to return memory from idle processes.

. Less dynamic response.

Tune for a less dynamic response that will stabilize and track moderate needs for working set growth. To implement this strategy: Establish moderate values

for AWSTIME, WSINC, and PFRATH. For example, set WSINC equal to approximately 10 percent of the typical value for WSDEFAULT. Provide more generous working set defaults, so that you do not need to set BORROWLIM so low as to ensure that loans would always be granted.

The first strategy works best in the time-sharing environment where there can be wild fluctuations in demand for memory from moment to moment and where there tends to be some number of idle processes consuming memory at any moment. The second strategy works better in a production environment where the demand tends to be more predictable and far less volatile.

At some time, you are likely to consider changing some of the system parameters. Do so with care as in many circumstances, a mistake can be fatal and require emergency procedures to recover the system.

To obtain a list of system parameters:

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN> SET/OUTPUT=filename
SYSGEN> SHOW/ALL
SYSGEN> SHOW/SPECIAL
SYSGEN> EXIT
$ PRINT/DELETE filename
```

This list is hardly complete. I don't know of a way to generate a full list. The only way to learn of most of them is to read the System Utilities Guide under SYSGEN.

Modify parameters in SYS$SYSTEM:MODPARAMS.DAT with a text editor. Then run AUTOGEN.

```
$ @SYS$UPDATE:AUTOGEN SAVPARAMS REBOOT FEEDBACK
```

Please read up on feedback in the system managers guide before choosing FEEDBACK/NOFEEDBACK.

Should you mess up and produce an unbootable system, here is how to go about trying to recover:

First, you should know what parameters you changed and what their previous values were so you can reset them. This implies that you're not trying to change everything at once. Then, after bringing the system completely down, do a conversational boot from the system console as follows:

```
>>>boot -flags 0,1      ; In a cluster, the zero could be any other number.

SYSBOOT>SET/STARTUP OPA0:
SYSBOOT>WRITE SYSPARAMS 0
```

At this point you should change any system parameters that you previously modified back to their original values so that you will again have a working system. Below is a quick example:

```
SYSBOOT>SHOW MAXPROCESSCNT
SYSBOOT>SET MAXPROCESSCNT 64
.
.
```

.

After setting any system parameters you need to set, issue:

```
SYSBOOT>CONTINUE
$ SET NOON
$ SPAWN
$ @SYS$SYSTEM:STARTUP
```

SET NOON is important because, if you throw an error, you'll be logged off. This procedure will run the normal startup in a sub-process. This is desirable because, when complete, the sub-process will be logged off. Using a sub-process leaves you logged in as a fully privileged user with the ability to run whatever other system utilities you may need. For example:

```
$ SET DEFAULT SYS$SYSTEM:
$ RUN SYS$SYSTEM:AUTHORIZE
AUTH>MODIFY SYSTEM/PASSWORD=newpassword
AUTH>EXIT
$
```

When you feel you have done enough to get back a fully operational system, log out, log back in and continue with the tuning operation.

To worry about working set sizes, @R$:[COM]WORKING_SET.COM will display the working sets of all the processes it can.

To show the status (fullness) of the swap and paging files, issue the command:

```
$ SHOW MEMORY/FILES/FULL
```

To check for too little memory problems, check the free page limit using MONITOR SYSTEM.

IO problems

$ SHOW DEVICE/FILES will show all open files and who has them open. This can be helpful, along with monitoring top direct IO, to determine which files need to be on a faster/larger/smarter disk. FYI disks use Direct IO, terminals use Indirect/Buffered IO.

$ @SYS$UPDATE:LIBDECOMP can be used on a new/ungraded installation to decompress the libraries. It greatly reduces their access times, for example, in linking.

$ SET/VOLUME/NOHIGHWATER_MARKING duxx: will turn off the security feature which prevents other processes from reading files to which they should not have access. Privileged users can do this, if they're smart enough. However, this security step is costly and, if you can do without it, you can save a lot of IO and CPU cycles.

****
9. Special considerations and miscellaneous material.
****

The system startup script is where you get the system ready to support users. It is run automatically at boot. The system manager will edit this file. The kinds of things done here are: start DECNET, start TCP/IP networking, Start DECWINDOWS (doesn't work here because of HP's disregard for the modern TCP stack) and other

things specific to this environment.

These specific things often change fairly frequently. Rather than making the system manager edit SYS$SYSTEM:SYSTARTUP_VMS.COM each time, we have chosen to execute a specific script in the system manager's private area. So, the startup script, SYS$MANAGER:SYSTARTUP_VMS.COM, mounts the required disk, defines a concealed logical and then executes a script in the manager' private area called OUR_SITE.COM, in this case R$:[COM}OUR_SITE.COM.

OUR_SITE.COM mounts all the rest of the disks as appropriate to the emulator configuration files, defines more rooted logicals, runs other utilities as needed and finally executes the script, STOPSTART.COM in the same private area.

STOPSTART.COM is a script used to stop running programs prior to rebuilding the shared libraries (DLL's, see below) and the rest of the system programs written by the author, remove installed programs and then, after rebuild, re-install and run them. This entire operation is driven by a text data file R$:[DATA]STOPSTART.DAT. This is so that things can be added or changed easily without modifying script code which in turn makes mistakes less likely.

This same script/data file is used at system startup and makes for a clean and easily modified startup for your ever changing environment without having to modify a very important system file where a mistake can seriously delay getting the system back up and running for your users.

Below is the data file so that you can see what drives the script:

```
================================================================================
0, This file contains information that allows scripts to
0, stop/start programs and install/remove libraries/programs
0, The first field is type followed by the full file spec
0, then other appropriate information for the operation.
0, Type 0 are comments and can go anywhere, the other
0, types and programs must remain in order.
0,
0, P1 = "1" is for stopping programs.
1,R$:[EXE]POKE.EXE,"Conn-Edit"
0,R$:[EXE]PURGEWS.EXE,"Purgws"
0, P2 = "2" is for remove/install
2,SYS$SHARE:MARLIB.EXE,/OPEN/HEADER/SHARE/PROT
2,SYS$SHARE:THE_PACKAGE.EXE,/OPEN/HEADER/SHARE
2,SYS$SHARE:TULASI.EXE,/OPEN/HEADER/SHARE
2,R$:[EXE]TSR.EXE,/OPEN/HEADER/PRIV
2,R$:[EXE]POKE.EXE,/OPEN/HEADER/PRIV=(PHY_IO,OPER,CMKRNL)
2,R$:[EXE]SWIM.EXE,/OPEN/HEADER/PRIV=(PHY_IO,SHARE)
0,R$:[EXE]PURGEWS.EXE,/OPEN/HEADER/PRIV
0, Images to be run after installs done
3,R$:[EXE]POKE.EXE,/DET/OUTPUT="NL:"/PROC="Conn-Edit"
0,R$:[EXE]PURGEWS.EXE,/DET/PROC="Purgws"
0,=========================================
0, P1 = "3" is for the site specific startup
0, Scripts to run
4,SYS$MANAGER:SYNERGY_STARTUP.COM
4,SYS$STARTUP:CHARONCP_STARTUP.COM
0, Insstalled images
5,SYS$SYSTEM:MACRO.EXE,/OPEN/HEADER
5,SYS$SHARE:MARLIB.EXE,/OPEN/HEADER/SHARE/PROT/RES
5,SYS$SHARE:THE_PACKAGE.EXE,/OPEN/HEADER/SHARE
5,SYS$SHARE:TULASI.EXE,/OPEN/HEADER/SHARE
```

```
5,SYS$SHARE:EXECSYMB_KM.EXE,/SHARE/PROT
5,R$:[EXE]TSR.EXE,/OPEN/HEADER/PRIV
5,R$:[EXE]POKE.EXE,/OPEN/HEADER/PRIV=(PHY_IO,OPER,CMKRNL)/RES
5,R$:[EXE]BOSS.EXE,/OPEN/HEADER/PRIV=(PHY_IO,OPER)
5,R$:[EXE]SWIM.EXE,/OPEN/HEADER/PRIV=(PHY_IO,SHARE)0,R$:[EXE]PURGEWS.EXE,/OPEN/
HEADER/PRIV
0, Images to run
6,R$:[EXE]AST_LOAD.EXE
6,R$:[EXE]SSDDLOADER.EXE
6,R$:[EXE]POKE.EXE,/DET/OUTPUT="NL:"/PROC="Conn-Edit"
0,R$:[EXE]PURGEWS.EXE,/DET/PROC="Purgws"
0, Definitions of system wide terminated logicals
7,P$,DKA100:[PAUL.]
7,J$,DKA100:[JEFF.]
7,S$,DKA200:[STUFF.]
0, Definitions of some system wide logicals
8,DX$HOME,R$:[DX]
8,WNDC,W/D,R/D,G/D,M/D,Y/D,B/D,C/D
0, Terminals to set to disconnect
0, Now done by setting TTY_DEFCHAR2 in MODPARAPS.DAT
0,OPA0
0,TTA0
```

Please note that MARLIB.EXE is installed /SHARE/PROT. It has several entry points which are called by changing mode to kernel and must be protected from being written from user or executive mode. And, by the way, I discovered that you can install an executable image with privileges but not a shareable image.

In our environment, we also create/deploy/start various queues. Basic batch and print queues for the normal operation are discussed and done in SYSTARTUP_VMS.COM. Here, we reference a script, R$:[COM]QOPERATIONS.COM, Which handles ALL queue operations. It is designed specifically for our environment, but shows how to do things in general for both stand alone and cluster configurations. It also demonstrates the use of a special operations symbiont.

==========

DLL Hell:

Libraries of callable subroutines and/or functions have been around for a very long time, certainly preceding the VAX. However, It was not until Virtual Memory System, VMS, came into existence that it became possible to reorganize a library in such a way that a single instance of it in memory could be shared with multiple programs running in multiple processes.

Thus came the terms "Dynamically Linked Library" (Windows) and "Shared Image" (VMS). They are the same thing. The idea is that the code is position independent in that addressing is relative and the library can reside anywhere in memory. The associated data areas are "copy on reference" in that a copy of writable, non-shareable data is made on reference by each program linking to the library at run time. Writable data is unique, executable code is re-entrant and cam be shared. This cuts down on disk storage and reduces memory usage - dramatically.

We have three shared images. Because of the common calling methods used by VMS, they can be used by programs written in any language. Producing these shared images traditionally involves hand editing the link option file to produce ordered global links. This is especially difficult for C++ because of the

mangled names required for multiple instances of classes. We have automated it. UPDATEx.COM where x = "M", "X", "D" or "A" for the executets discussed at the beginning, will do this. "UPDATE" runs R$:[COM]UPDATE_ALL.COM" which handles stopping and removing all the files installed and/or run at startup and then installing and restarting everything when complete.

MARLIB is the shared MACRO32 library which is referenced by most of my programs including some written in DBL and C++ and some which operate in kernel mode at high IPL. You will note that it is installed resident as well as privileged. This is so there is no startup overhead to use it.

THE_PACKAGE is similar to my DLL of the same name on Windows. It contains lots of useful functions for the C++ programmer. TULASI is similar but for DBL programs.

Installing the TSR privileged allows anyone to run it and use it. It requires CMEXEC privilege.

You might want to check out some of the DCL procedures in R$:[COM]. We will not go into them here to minimize boredom, but one procedure should be mentioned because it it helpful for the systems programmer. It is LIST_SYMBOL.COM. Run it without arguments to get instructions. It is used to print out symbols in $XXDEF macros, .MLB system files and .STB files read by the operating system and related to system services.

We installed a spelling checker. It works pretty well. To use it, define a symbol, SPELL :== R:[SPELL]SPELL.ALPHA_EXE. The screen makes it self documenting.

LOGIN.EXE is used to execute all the key definitions below. At present, all but the ENTER key are defined programaticly in LOGIN.EXE. The old file is listed here as it is more readable.

```
$ set terminal/application
$ n = f$process()
$ na = "EVE_" + n
$!
$! Note the technique for setting the "gold" state below.
$! If you use $ SET KEY, you will set it until you issue another
$! that sets  it to DEFAULT. Notice the /noterminate. Without
$! the "no", it fails with some nonsense about /nolock.
$! Using this technique sets the state for the next keystroke
$! only. This is what most people want.
$!
$ define/key PERIOD "" /set_state=gold/noterminate/noecho/nolog
$ define/key/if_state=gold PERIOD "@r$:[com]settime.com" /terminate/echo/nolog
$ define/key ENTER "attach "'na' /terminate/echo/nolog
$ define/key PF1 "set default r$:[data]" /terminate/echo/nolog
$ define/key/if_state=gold PF1 "set default s$:[zip]" /terminate/echo/nolog
$ define/key PF2 "set default r$:[help]" /terminate/echo/nolog
$ define/key/if_state=gold PF2 "set default s$:[lp]" /terminate/echo/nolog
$ define/key PF3 "set default r$:[dev]" /terminate/echo/nolog
$ define/key/if_state=gold PF3 "set default dka400:[000000]" /terminate/echo/nolog
$ define/key PF4 "set default r$:[eve]" /terminate/echo/nolog
$ define/key/if_state=gold PF4 "@sxmgr:[com]sx_logon.com" /terminate/echo/nolog
$ define/key kp0 "run r$:[exe]menu.exe" /terminate/echo/nolog
$ define/key/if_state=gold kp0 "run sxmgr:[exe]sx_menu.exe" /terminate/echo/nolog
```

```
$ define/key kp1 "dir" /terminate/echo/nolog
$ define/key/if_state=gold kp1 "dx" /terminate/echo/nolog
$ define/key kp2 "set default r$:[com]" /terminate/echo/nolog
$ define/key/if_state=gold kp2 "processes" /terminate/echo/nolog
$ define/key kp3 "set default r$:[exe]" /terminate/echo/nolog
$ define/key/if_state=gold kp3 "screen e" /terminate/echo/nolog
$ define/key kp4 "set default sys$login" /terminate/echo/nolog
$ define/key/if_state=gold kp4 "login" /terminate/echo/nolog
$ define/key kp5 "set default r$:[dbl]" /terminate/echo/nolog
$ define/key/if_state=gold kp5 "rec" /terminate/echo/nolog
$ define/key kp6 "set default r$:[tp]" /terminate/echo/nolog
$ define/key/if_state=gold kp6 "type r$:[com]keydefs.txt" /terminate/echo/nolog
$ define/key kp7 "set default r$:[cxx]" /terminate/echo/nolog
$ define/key/if_state=gold kp7 "clean" /terminate/echo/nolog
$ define/key kp8 "set default r$:[mar]" /terminate/echo/nolog
$ define/key/if_state=gold kp8 "rbckup" /terminate/echo/nolog
$ define/key kp9 "set default r$:[ast]" /terminate/echo/nolog
$ define/key/if_state=gold kp9 "sedit" /terminate/echo/nolog
```

****

10. System Security

****

I've been using computers, developing systems and automating business for over fifty years. I've learned that automating a business is an excellent opportunity to put yourself out of business. Integrating the computer into your operation is a technically difficult process filled with traps.

While the internet is an amazing invention expanding knowledge and the ability to communicate, unfortunately, it has also increased the opportunity for fraud, deception and disaster. I am a great admirer of the internet. But it is dangerous and often used for operations for which it should not be used.

We feel that in order to run a secure computer installation, the core business operations should be isolated from the outside world. The company should handle the storage, backup and usage of all critical data needed for continued operation or any material that is proprietary.

Of course a business should make use of the internet. Those who don't will fall by the wayside. But that does not mean the operation should expose itself to possible danger. The interface between the internet (outside world) and your main system should be managed with security as a primary consideration and should have a human component to oversee security. Any data from the outside world that goes into the company system should be carefully scrutinized and only the trusted systems manager should be allowed to place new programs and/or packages onto it. This takes planning.

The idea of privileged developers being able to log into the system over the internet may sound good, but automatically compromises the integrity of the system. Work from home, in most cases, is not acceptable.

The "cloud" is fine for large amounts of data that aren't proprietary or important, but companies that depend on it for critical operations or data storage will likely find themselves damaged sooner or later. The internet is not safe.

On the other hand, if work involves data entry or lookup and the worker can run from a captive account so that damage cannot be done to the system, logins over the internet can be acceptable and useful.

The basic point is that nobody cares more for the security of your data and operations than you do. Don't farm it out to third parties just to save a little money or because it's convenient or the latest trend.

The OpenVMS operating system is famous for its security (government C2 rated). With proper design, the system manager can be almost certain that only the proper people have access to various files/devices/subsystems/functions as well as define exactly the type of access they have. For example, some people in Human Resources may have a need to see payroll records but not salaries and not be able to change anything. This is easily accomplished for individuals or groups.

When an entry in the user database is made by the system manager for a new user, the manager will, using the User Identification Code, UIC, [x,y], assign that user to a group while making his UIC unique. The user can be granted authorized and default privileges just sufficient to carry out his duties. The fewer the privilege, the less likely it is that the user can harm the operation. The user will have various access permissions based on his group membership. He can also be assigned specific access of specific types to specific resources through the use of identifiers as discussed below.

A highly privileged user can, with the right privileges, bypass the managers security plan and, if malicious, destroy the entire system.. Be very careful of how you go about granting privileges. Since we have included a bit of programming magic to allow the system manager the ability to change a processes current privileges on a temporary basis, if a situation rises where a specific user needs to do something special, the manager might change his privileges, watch over him as he does what is necessary and then reset his privileges (or log him out) so he returns to normal.

The manager can even set up the user to be "captive" so that he may run the program set up in his LOGON.COM file followed by programs to which the original chains, but any attempt to gain access to the DCL prompt causes him to be instantly logged off the system.

There are eleven categories of objects on which you may set permissions or attach access control entries to form access control lists. They are:

"FILE"
"DEVICE"
"VOLUME"
"QUEUE"
"CURRENT_EVENT_FLAG_CLUSTER"
"LOGICAL_NAME_TABLE"
"RESOURCE_DOMAIN"
"CAPABILITY"
"SYSTEM_GLOBAL_SECTION"
"GROUP_GLOBAL_SECTION"
"SECURITY_CLASS"

We have written a DCL script which will, using the appropriate DCL commands and running the appropriate utility programs, allow you to:

1. Set object permissions
2. Create an identifier
3. Delete an identifier
4. Grant an identifier to a user
5. Revoke an identifier from a user

6. Run AUTHORIZE
7. Add an ACE to an object
8. Remove an ACE from an object
9. Run the ACL Editor for the object
10. Encrypt a file
11. Decrypt a file
12. Add privileges to a process
13. Remove privileges from a process
14. Change a process priority

Note! All of the above save the changing of privileges can be done from the DCL command prompt. This script makes remembering the somewhat complex default directories and command switches unnecessary.

The script is R$:[COM]RSECURITY.COM and is normally run from MENU.EXE.

1) Object permissions are owner based. That is, they revolve around the group number to which the owner belongs. A typical file permission (protection) would look like: (System:RWE, Owner: RWED, Group: RE, World: R). This means that the owner has control over the existence of the file, the system user, "[1,...]", can read, write, and execute it, fellow members of the owner's group can read and execute but not change it. Everyone else, (World) can only read it.

2,5) Identifiers are either individual UIC's, UIC's with wild cards or just alphanumeric names you've created. They are put into the rightslist database and granted/revoked to/from users using the AUTHORIZE utility program. Our script will do most anything you need relating to identifiers for you.

6) You can run AUTHORIZE.EXE from here to create/modify accounts as well.

7,8) An Access Control Entry (ACE) is a string that is associated with an object. It has two parts. First is a list of identifiers and second is a list of access permissions appropriate to the class of the object (one of the eleven types listed above). Thus an ACE for a file might look like:

(IDENTIFIER=(CUST,[1,*]),ACCESS=(READ+WRITE)

This means any user holding CUST identifier and anybody with a system (group id of 1) UIC can read and write to the file.

An Access Control List is more than one ACE. Thus the same file might have an ACL of:

(IDENTIFIER=[1,6],ACCESS=CONTROL)
(IDENTIFIER=(CUST,[1,*]),ACCESS=(READ+WRITE)
(IDENTIFIER=(*),ACCESS=NONE)

This means that only one user (presumably the owner) can delete the file and nobody but those in the second ACE can touch the file.

9) You can run the ACE Editor from here.

10,11) For the last bit of security, you can encrypt/decrypt files. That way nobody but you can get at the content regardless of their privileges.

12,13) You may grant or revoke any and all privileges from any process whether or not they have the privilege authorized. This involves an execulet KAST. and the program FORCEPRV.EXE which should not be run except from this script.

14) Lastly, you can change your own or another process' base priority. You may want to do this if the system is slow and some important but short lived process needs to get done. You might consider raising your own priority to real time (16) if a programmer has inadvertently gotten himself in an exceptionally tight infinite loop that won't let him interrupt his program. I've seen it happen and it eats the system until you can delete the process.

LOGIN security enhancement.

We also have the LIOH program written in "C" in combination with the LIOH.COM script can set up a scheme where logins are limited in a variety of ways. For example, as provided, our hook into the login process will only allow a user who's user name starts with "USER..." to log in over TCP/IP. In our system, all those accounts are captive. Note that the privileged user SYSTEM, has to be local. You may devise your own scheme.

****
11. System Manager Utilities
****

The following is a partial list of System Management Utilities with which a system manager should be familiar. We have left out some highly specialized items. For complete descriptions, read the two volume set of "System Manager Utilities". Some of the utilities below get only passing mention.

ACE - Access Control List Editor. See the discussion of security above.

ANALYZE/DISK/REPAIR will repair a disk that has been damaged by software and can free up space by deleting files marked for deletion.

AUDIT has been discussed above. It can be used for any number of situations including charging for system use. ANALYZE/AUDIT is also helpful.

AUTHORIZE is the program that allows the system manager to create and modify user accounts plus set authorized privileges and grant identifiers. See the System Security discussion above.

AUTOGEN is used to change system parameters. SYSGEN can change dynamic parameters for testing, and even make them permanent by writing them out to the master file used at boot. Don't do this. Your changes will be overridden the next time it is done properly using AUTOGEN. Edit SYS$SYSTEM:MODPARAMS.DAT and run AUTOGEN to make any permanent changes.

BACKUP Use it frequently and use the /VERIFY switch. If you don't, I'll be adding you to my list of horror stories.

ELV - Error Log Viewer. Since 7.3 VMS has written the error log in binary. You need this utility to see it. See below:

```
$ SAVED = F$DIRECTORY()
$ SET DEFAULT R$:[TEMP]
$ IF P1 .NES. ""
$ THEN
$  FILE := R$:[TEMP]ERRLOG.SYS
$ ELSE
$  FILE := SYS$ERRORLOG:ERRLOG.SYS
$ ENDIF
$ WRITE SYS$OUTPUT FILE
$ ANALYZE/ERROR/ELV TRAN/FULL/INCLUDE=CONTROL_ENTRIES/SINCE/OUTPUT=ELV.LIS 'FILE
```

```
$ SEA ELV.LIS "UGAL"
$ INQUIRE YN "Delete the file"
$ IF YN
$ THEN
$  DELETE/CONF/LOG 'FILE';*
$ ENDIF
$ SET DEFAULT 'SAVED'
$ SAVED = F$DIRECTORY()
$ SET DEFAULT R$:[TEMP]
$ IF P1 .NES. ""
$ THEN
$  FILE := R$:[TEMP]ERRLOG.SYS
$ ELSE
$  FILE := SYS$ERRORLOG:ERRLOG.SYS
$ ENDIF
$ WRITE SYS$OUTPUT FILE
$ ANALYZE/ERROR/ELV TRAN/FULL/INCLUDE=CONTROL_ENTRIES/SINCE/OUTPUT=ELV.LIS 'FILE
$ SEA ELV.LIS "UGAL"
$ INQUIRE YN "Delete the file"
$ IF YN
$ THEN
$  DELETE/CONF/LOG 'FILE';*
$ ENDIF
$ SET DEFAULT 'SAVED'
```

We copy the error log to our temp directory and delete the one in system on boot. It can get large. You will notice the search of the ELV.LIS file. It is for records which have been written by SERR in MARLIB. Useful for debugging code that runs in kernel mode.

INSTALL is used to make programs known to the system and more responsive. Programs can be installed with privileges not ordinarily given to users. In this way, special programs can be used by the unprivileged user. It is discussed above.

LANCP & LATCP are used to control the network. For most installations, properly configuring things will be enough. In complex cases, familiarity with these utilities can be necessary.

MONITOR is discussed above.

MSA relates to the control of special disk storage controllers. Specialized.

PPPD - controls point to point network protocol (dialup). Hardly ever used now.

POLYCENTER - fancy VMSINSTAL which works very well in the cluster environment. it can also be used to reconfigure the system and uninstall subsystems you don't need (i.e. a language no longer needed).

SAS is used in the control of RAID disks and disk mirroring.

SCACP is used for cluster interconnect (LANIP) circuit management and tuning. For physical clusters this can be an important tuning program. For our cluster, not so much.

SYSGEN, discussed briefly above, will allow you to change dynamic system parameters. Useful for experimental tuning. Can be used to make permanent changes by writing the parameters to the appropriate system file. However, this is bad practice. Changes made this way will be lost the next time the system is

modified properly with changes to MODPARAMS.DAT and AUTOGEN.

SYSMAN is not usually necessary for single nodes. However, it is very useful for clusters as it does the equivalent of many DCL commands across the entire cluster. If you have a cluster, learn this one.

UCB - USB configuration utility. I haven't read this one but it seems strange that you wouldn't treat a USB drive just like any other. Oh well.

VDDRIVER.EXE located in R$:[VDISK], is a driver that allows you to create a virtual disk on a real disk. You can use it as desired except for SHADOWING, for which it was designed. Look in R$:[COM]CREATE_VDISK.COM and MOUNT_VDISK.COM for the details.

This system creates two 256mb virtual disks on "EXTRA", DKA200:, called VDA0: and VDA1:, with a root directory of BASE.DIR referenced by the rooted logicals V1$: and V2$:. V1$: is intended for people to store sensitive data. For example, V1$:[RICK] has an access control list that excludes everyone else and contains files which have been encrypted with the ENCRYPT DCL command.

If you edit R$:[COM]CREATE_VDISK.COM, you will notice that the files used as virtual disks, VDISK?.DSK, are created using a copy operation with the switches of contiguous and allocation values. Contiguous files are required and the assignment of the logical VDA? to the disk will fail without notification should the file not be contiguous. Allocation values have been chosen to mimic a disk type, as seen in the comments of R$:[MAR]ASNVD_64.MAR. Apparently this can be advantageous for certain types of operations.

****
12. kitinstal
****

@R$:[COM]KITBUILD.COM will build a VMSINSTAL kit for a subset of the processes and procedures most used by system managers. It is menu driven.

To create a VMSINSTAL kit, you will use the hidden logical S$ and the kit will be built in S$:[KIT]. Execute @R$:[COM]KITBUILD.com. It will copy all necessary files, including KITINSTAL.COM, and compile all necessary executables. The executables are distributed as .OBJ files and linked on site for safety and compatibility. Many of these programs run in KERNEL mode and would crash the system if there were library mismatches.

The operation is basically driven by R$:[DATA]KIT.DAT below:

```
# Comma separated list of files and what to do with them
# Actions:
# 1 = just link according to type
# 2 = same as 1 but add FIND_LDRIMG for execlets
# 3 = same as 1 but link TTDI
# 5 = move to sxmgr:[com] or [dat]
# 7 = special copy of .MAR files
# 8 = things for AST_LEI
# 9 = things for MARLIB
marlib,mar,9
ast_lei,mar,8
rwast,mar,1
clrerr,mar,1
defdir,mar,2
dkfree,mar,1
```

```
getcmd,mar,1
glogin,mar,1
menu,mar,1
processes,mar,1
poke,mar,1
lockholder,mar,1
pdt,mar,1
purgws,mar,2
forceprv,mar,2
suspend,mar,1
tsr,mar,1
sx,dat,5
monitor,com,5
color,com,5
dkfree,com,5
purgws,com,5
image_accounting,com,5
rsecurity,com,5
creprvlst,com,5
prvlst,dat,5
working_set,com,5
adv_setup,com,5
adv_logon,com,5
adv_inst,com,5
sysmans,com,5
tshut,com,5
who_open,com,5
```

Each line contains three comma separated fields: file name, extension and code for what to do with it.

At the end of the operation which puts everything into a directory, S$:[KIT], the procedure invokes the DEC @SYS$UPDATE:SPKITBLD.COM. If you do not wish to build the kit at this time, say no when asked. When asked for the source, use the explicit device and directory, S$:[KIT].

****

If you've gotten this far, you're a true VMS'er. Congratulations.

No matter what happens from here, I've given this material in good faith, warts and all. I hope it will be of benefit. We who admire VMS think the OS deserves its reputation and hope it will continue. With your help, it will.

Rick Marsh Ph.D.
rick@rickmarsh.com