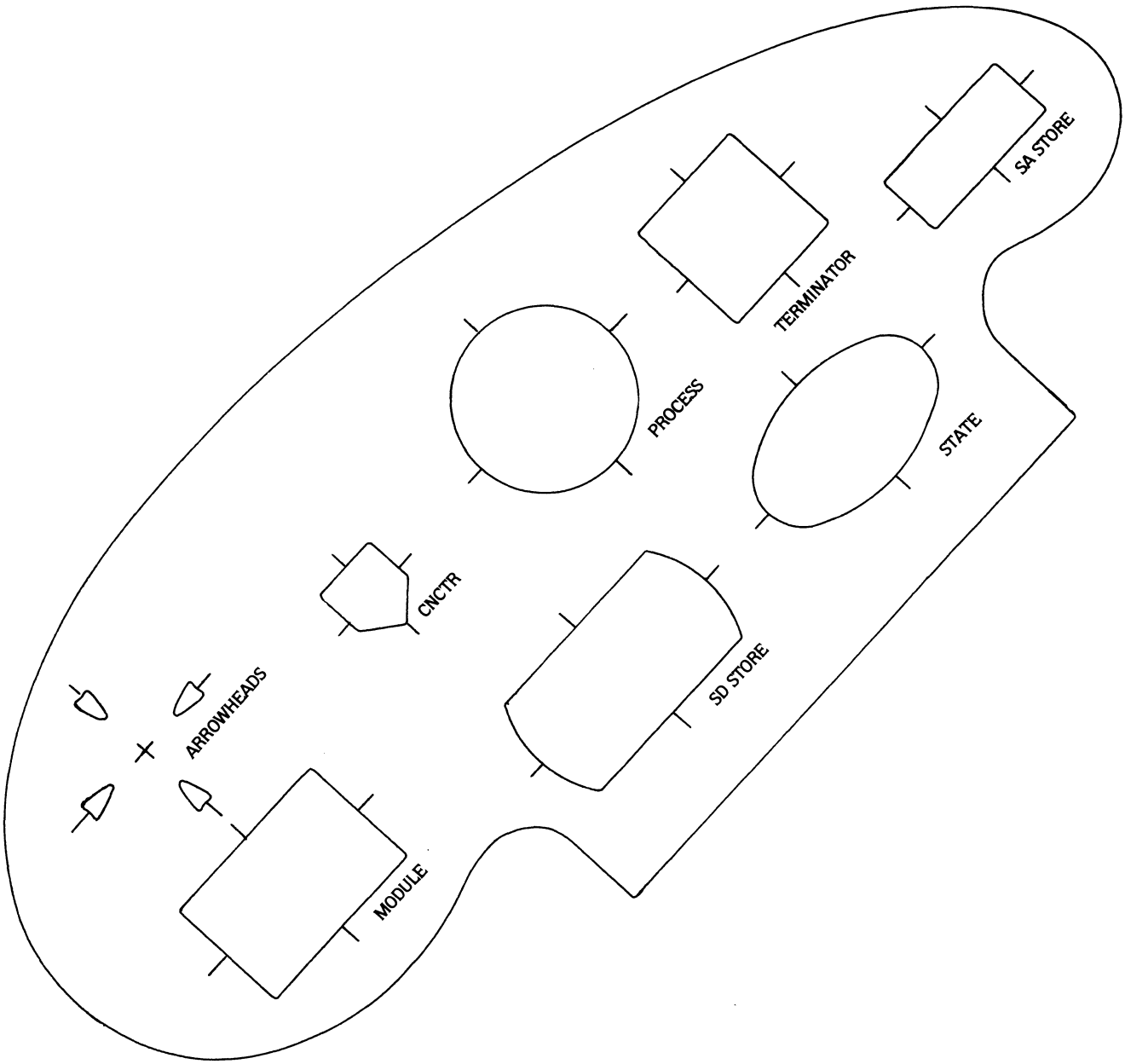
"THE HEAP"

DECUS

FEBRUARY 1985 ISSUE



LANGUAGES AND TOOLS SIG

Table of Contents

---

* Ada is a trademark of the U.S. Department of Defense

Editor's Notes

As you may be aware, the Anaheim symposium was a momentous one for the Languages and Tools SIG. This issue of "The Heap" consists of articles and handouts relating to the many new products which DEC announced in our area of interest. Sessions at Anaheim were jammed, and the presenters frequently ran out of their handouts. I'm trying to publish as many of them as I can get, but if there is a particular handout you didn't get and I haven't provided the information you need in this issue, please let me know and I'll try to get it out in a future issue.

In addition to the articles on DEC products, this issue of "The Heap" contains the LTSIG's latest wishlist. Please take the time to read it carefully, and respond according to the directions. The more responses we get, the greater the weight the survey will have with DEC. Also, I hope to present the results of this wishlist at the New Orleans Symposium, so it is important that you return your responses quickly, if I am to have time to tabulate the results and put together a presentation.

On other matters, I have been underwhelmed by the response to my request for suggestions for a new name for the newsletter. If you folks are willing to live with it we'll just put up with the connotations of "The Heap". I'm not proud. Otherwise, send me your suggestions for a new name. Remember, I'll send an LTSIG Mushroom to whoever submits the winning name. Speaking of the Mushroom, in the interest of giving credit where credit is due, Brian Tillman informs me that Derek Hatley of Lear Siegler is the original designer of the Mushroom. The SIG is grateful to Derek and Lear Siegler for permission to use the design. Also, I am still working on putting together an issue relating to public domain software in which the SIG has an interest. I can't write the articles myself, however. Let me know if you can help out in this area.

Finally, please don't get used to these huge newsletters. Even if DECUS doesn't crack down on me about printing expenses, I'm soon going to run out of articles. The US in DECUS stands for User's Society, and I would prefer to get the articles for the newsletter from users. If you have some experience with a new language, or have had problems with a software tool, or whatever, that experience may be useful to others. Articles don't have to be elaborate. If you think you might have something to contribute, please let me know, and we can discuss any bothersome logistics. My work phone is (215) 674-7154, and my address is:

Alan L. Folsom, Jr.
Dept 431
Fischer & Porter Co.
E. County Line Road.
Warminster, Pa. 18974

VAX Performance & Coverage Analyzer

VAX Performance and Coverage Analyzer Version 1.0

With a keyword for industry in the 80's being "productivity", computer programmers need not only languages and operating systems, but tools to make their systems more productive. The VAX Performance and Coverage Analyzer is a new productivity tool running on VMS and MicroVMS which will help applications programmers write code which will execute more efficiently.

What Is It?

The VAX Performance and Coverage Analyzer has two functions. First, it can pinpoint execution bottlenecks and other performance problems in application programs. Secondly, it provides test coverage analysis by measuring what parts of a user program are executed or not executed by a given set of test data. This product is not a tool for the analysis of operating system performance or for use as an aid in hardware resource planning.

How Does it Work?

The VAX Performance and Coverage Analyzer consists of two parts: the Collector which gathers the performance or test coverage data on the running user program and the Analyzer which later processes and displays the collected data. The Collector gathers the performance data and writes that information to a data file. Once the data has been collected into a data file, the Analyzer can be run using that data file as input. The Analyzer is a separate program which reads that data file produced by the Collector and presents the results as performance histograms or tabular displays.

Both the Collector and the Analyzer are fully symboblic and use the DEBUG Symbol Table (DST) information in the user program to access the symbolic names of program locations. Applications written in any of the VMS languages which produce DST information can be analyzed using the VAX Performance and Coverage Analyzer. These languages include:

VAX Ada	VAX Fortran
VAX Basic	VAX Macro
VAX Bliss	VAX Pascal
VAX C	VAX PL/I
VAX Cobol	VAX RPG II

What Kinds of Performance Data Can be Gathered?

The VAX Performance and Coverage Analyzer can gather and report the following types of performance data:

o PC samples - The PC (Program Counter) of the user program can be randomly sampled to determine what parts of the user program take the most time during program execution.

o Execution counts - Execution counters can be set on specified program locations or on classes of locations, such as the beginning of every routine or on every line of a given routine.

o Test Coverage - The Collector can determine which code paths of a program are executed for a specific set of test data and which are not.

o Event markers - When collecting performance data, the user may wish to mark significant events in the execution of the program to permit later filtering of the data. For example, an event may be entering a routine in an application program.

o Page fault data - Information can be collected each time a page fault occurs.

o System service counts - The Collector can determine the number of times a user program calls each system service for analysis of event-driven or real-time applications.

o Input/Output statistics - Data on a program's input and output characteristics can be collected.

When is This Product Available?

The VAX Performance and Coverage Analyzer was announced at Fall U.S. DECUS in Anaheim, CA. It will be ready for first shipments to customers in the Spring of 1985.

For more information on this product, contact:

Susan Azibert
Product Manager
Digital Equipment Corp. (ZK02-3/Q08)
110 Spitbrook Rd.
Nashua, NH 03062

VAX DEC/Shell

VAX DEC/Shell Version 1.0

The DEC/Shell is a command language that provides an interface to the VMS and MicroVMS operating systems, similar to the interface on a UNIX (1) V7 system.

There are two major components of the DEC/Shell: the command line interpreter and the Shell script language. When combined with common UNIX utilities, these components provide a program development environment familiar to users experienced with the UNIX V7 system.

NOTE: The DEC/Shell is not a UNIX system running on a VAX. The DEC/Shell is a command line interpreter and programming language based on the UNIX V7 Bourne Shell.

The DEC/Shell includes the following features:

- environment variables
- pipes
- user-definable search paths for command execution
- UNIX file name syntax
- input and output redirection to and from files
- commands and utilities similar to those found on a UNIX system
- string substitution
- structured constructs (such as while, if-then-else, case, and for)
- modification of a command's environment
- a Shell run-time library

In addition, the DEC/Shell provides access to DCL commands and VMS programs. This capability allows users familiar with the UNIX V7 system to take advantage of the VMS and MicroVMS operating systems while working in a familiar programming environment.

1 UNIX is a trademark of AT &T Bell Laboratories

The DEC/Shell Environment

The DEC/Shell environment is very similar to the environment on a UNIX V7 system. Input/output redirection, search paths, pipes, environment variables, and other Bourne Shell features all appear to the user as they do on a UNIX V7 system.

You can expand the DEC/Shell environment to take advantage of VMS and MicroVMS features. Symbols and logical names can be exported to or defined in the DEC/Shell. DCL commands and command procedures can be executed from the DEC/Shell.

Invoking the DEC/Shell

You can invoke the DEC/Shell from DCL level, or you can make the Shell your default command interpreter when you log in.

To invoke the DEC/Shell from DCL, you use the SPAWN command with the /CLI qualifier as follows:

```
$  SPAWN/CLI=SHELL
```

This command creates a subprocess with the DEC/Shell instead of DCL as the command interpreter. In this subprocess, you can perform many of the tasks you would do on a UNIX V7 system.

To make the Shell your default command interpreter when you log in, you can do one of two things:

1. Type /CLI=SHELL/NOCOMMAND after your username when you log in to the system:

```
Username:  BUNTHORNE/CLI=SHELL/NOCOMMAND
Password:
```

   This action will override any other default command interpreter that the system manager has specified for your account. The /NOCOMMAND qualifier prevents the Shell from trying to execute your DCL login command procedure (LOGIN.COM). Errors may result if the DEC/Shell tries to execute a DCL command procedure.

2. Ask your system manager to make the DEC/Shell the default command interpreter for your account.

Once you have invoked the DEC/Shell, you can perform many of the tasks that you would normally perform on a UNIX system. DEC/Shell command syntax is, for the most part, the same as that for the Bourne Shell used on the UNIX V7 system.

The DEC/Shell as a Programming Language

While the DEC/Shell can be used primarily as a command interpreter, it is also a powerful programming language. Many of the control structures used in the DEC/Shell are similar to those used in the C language. Given the DEC/Shell script language, control-flow constructs, and utilities, you may find that the DEC/Shell is an adequate programming language for many of your needs.

DEC/Shell Utilities and Commands

The following is the list of commands and utilities provided by the DEC/Shell:

awk	join	sh
basename	kill	sleep
cat	lex	sort
cd	login	tail
chmod	logout	tar
chown	ls	tee
date	m4	test
dc	mcr	times
dcl	mkdir	touch
diff	mv	tr
diff3	od	trap
echo	pr	true
ed	ps	tty
eval	pwd	umask
exec	read	uniq
export	readonly	units
expr	rm	wait
false	rmdir	wc
find	sed	who
grep	set	yacc

Availability

The DEC/Shell begins shipment to customers in January 1985.

For more information on this product, contact:

Susan Azibert
Product Manager
Digital Equipment Corp. (ZK02-3/Q08)
110 Spitbrook Rd.
Nashua, NH 03062

VAX Language-Sensitive Editor

ANNOUNCING THE VAX LANGUAGE-SENSITIVE EDITOR Version 1.0

SUMMARY

The VAX Language-Sensitive Editor is a multi-language, multi-window, screen-oriented editor specifically designed to increase the productivity of programmers in the development and maintenance phases of the software development cycle.

The Editor is "language-sensitive" in that it provides you with knowledge of the syntax of the following VAX languages:

VAX Ada (r)	VAX COBOL	VAX PASCAL
VAX BLISS-32	VAX FORTRAN	VAX PL/I
VAX C		

This knowledge enables both new and experienced programmers to develop programs faster and more accurately through VAX language-specific construct completion, and error detection and correction facilities.

The VAX Language-Sensitive Editor works in concert with these VAX languages and the VAX Multi-language Symbolic Debugger to provide you with a highly interactive, on-line program development environment that facilitates the EDIT-COMPILE-DEBUG portion of the program development cycle.

You can customize the environment by tailoring and expanding upon the features and structures provided by the Editor. In addition, you may extend the editing capabilities provided by the Editor by calling user-written functions written in VAX TPU (Text Processing Utility - reference VAX TPU Section of this article).

"LANGUAGE-SENSITIVE" FEATURES

For each of the above stated VAX languages, the VAX Language-Sensitive Editor provides a set of formatted syntactic models that fit together to form syntactically correct programs. Within the models are strings that indicate places that require the insertion of additional information.

The VAX Language-Sensitive Editor uses these syntactic model definitions to complete partially entered language constructs - automatically inserting required words and punctuation, and indicating syntactic options, which are listed in a menu format.

VAX LANGUAGE-SENSITIVE EDITOR COMMANDS

You control the editing environment with an extensive set of commands. Most often-used commands can be easily bound to keys to

enable quick and efficient programming. All commands are available through "command mode".

COMMAND FEATURES include:

* Commands for language construct completion
* Screen manipulation commands
* Commands to invoke the appropriate VAX language compiler
* Command for compile-time error review
* Commands for compile-time error correction
* Commands for tailoring and saving the editing environment
* Commands for calling and executing VAX TPU procedures(refer to VAX TPU section of this article)
* SPAWN command for suspending the editing session to spawn a subprocess running the DCL command interpreter
* GOTO command to position the cursor at the indicated option (options include FILE, BUFFER, SCREEN, PAGE, etc.)
* SEARCH command for locating specified characters or strings
* SUBSTITUTE command for replacing the text of one string with another
* SHOW command to display characteristics and settings of several options (including BUFFER, COMMAND, KEY, LANGUAGE, ALIAS)

VAX LANGUAGE INTERFACES

The VAX Language-Sensitive Editor interfaces to VAX languages to provide you with an on-line, interactive program development environment. That is, you can create and edit programs, compile, and review and correct compile-time errors -- all within a single session.

FEATURES:

* Use of language-specific syntactic models to facilitate source code entry and minimize the occurence of syntax errors

* COMPILE command to perform compilations without leaving the Editor, and, optionally, to review compilation errors upon compile completion. You may specify DCL qualifiers such as /DEBUG and /LIBRARY when invoking the compiler from the VAX Language-Sensitive Editor. The compilation may be performed in a BATCH job.

* REVIEW mode to review compilation errors upon compile completion. The VAX Language-Sensitive Editor displays the compile errors in one window, with the corresponding source code displayed in a second window. For easy error correction, there is an EXAMINE ERROR command to position the cursor at the point in the source code where the compiler detected the error.

VAX MULTI-LANGUAGE SYMBOLIC DEBUGGER INTERFACE

The VAX Language-Sensitive Editor can be invoked from the VAX Multi-language Symbolic Debugger offering you the ability to make source code corrections as they are found during a debugging session.

FEATURES include:

* Notification if the file invoked by the editor is a different version than that displayed in the VAX Symbolic Debugger

* Ability to specify the file and line number from which to start the editing session with the default being the current source displayed in the VAX Symbolic Debugger

* Choice of terminating activity directly from the editing session or returing to the debugging session

USER INTERFACE

The LANGUAGE-SENSITIVE EDITOR offers an easy to use, flexible user interface designed to meet varying user needs and preferences.

FEATURES include:

* Default EDT-like keypad layout

* Multiple windows, multiple buffers

* Choice of editing mode: OVERSTRIKE or INSERT

* Ability to bind any command (User-defined or those supplied by the VAX Language-Sensitive Editor) to a key.

* On-line HELP facility for the VAX Language-Sensitive Editor

* VAX language-specific on-line HELP

USER-TAILORABLE FEATURES

With the VAX Language-Sensitive Editor, you can customize the editing environment to meet your indiviual needs. These user-defined environments can be saved for future editing sessions.

USER-TAILORABLE FEATURES include the ability to:

* Define a language name to represent a new set of user-defined syntactic models

* Define (or redefine) Syntactic Models, COMMANDS to complement those provided by the Editor

* Define ALIASes that are names that represent pieces of text

* Define (or redefine) keybindings

* Bind any command (including user-defined commands or combination of commands) to a key

VAX TPU (TEXT PROCESSING UTILITY)

For more unique editing requirements, the VAX Language-Sensitive Editor provides commands to call procedures written in VAX TPU (VAX Text Processing Utility) which is available as part of the VAX Language-Sensitive Editor. VAX TPU has an easy to use high-level procedural language which allows you to write functions not provided by the VAX Language-Sensitive Editor to further customize the editing environment. The VAX TPU language provides for looping and conditionals to allow you to perform more powerful editing tasks.

VAX TPU LANGUAGE STATEMENTS include:

* The assignment statement

* Procedural statements (PROCEDURE - ENDPROCEDURE)

* Looping statements (LOOP - ENDLOOP)

* Conditional statements (IF - THEN - ELSE - ENDIF)

* Case statements (CASE - ENDCASE)

* Error statements (ONERROR - ENDONERROR)

AVAILABILITY

The VAX Language-Sensitive Editor availability will begin in the Spring of 1985 for the full line of VAX and MicroVAX systems running the VMS Version 4.x or the MicroVMS Version 4.x operating system.

The Editor will run on any VT1xx, VT2xx, or ANSI terminal.

For more information, contact:

Celeste LaRock
Digital Equipment Corporation
110 Spit Brook Rd. (ZK02-3/Q08)
Nashua, N.H. 03062
Telephone: (603) 881-2336

VAX TPU

TEXT PROCESSING UTILITY

TUTORIAL

ADVANCED

TOPICS

THE TPU ENGINE

- HAS A COMPLETELY 'SOFT' HUMAN INTERFACE
  - NO DEFAULT KEY DEFINITIONS
- IS MORE THAN JUST AN EDITOR
  - PROVIDES EDITING CAPABILITIES SUCH AS TEXT INSERTION/DELETION
  - PROVIDES SCREEN MANAGEMENT OF BUFFERED TEXT
- TWO DIGITAL SUPPLIED INTERFACES AVAILABLE
  - EVE INTERFACE, designed by HUMAN FACTORS ENGINEERS
  - EDT KEYPAD EMULATOR INTERFACE

TPU BUFFERS

- o WORKSPACE IN WHICH FILES ARE EDITED
- o CAN BE CREATED WITH OR WITHOUT AN ASSOCIATED FILE
- o SPECIAL BUFFERS:
  - MESSAGE_BUFFER
    - * TPU MESSAGES
    - * BROADCAST MESSAGES
  - SHOW_BUFFER
- o CAN BE MAPPED TO MULTIPLE WINDOWS, WHERE EACH WINDOW CAN DISPLAY A DIFFERENT VIEW OF THE BUFFER
- o MODIFIABLE BUFFER ATTRIBUTES
  - TEXT FOR END OF BUFFER LABEL
  - DIRECTION (FORWARD OR REVERSE)
  - INSERT OR OVERSTRIKE EDITING
  - MARGINS FOR FILL BUILT-IN (LEFT AND RIGHT)
  - READ ONLY (NO_WRITE), PERMANENT, OR SYSTEM
  - TAB_STOPS
  - OUTPUT_FILE NAME

T P U W I N D O W S

- WINDOWS ARE AREAS OF THE SCREEN IN WHICH A BUFFER'S TEXT APPEARS

```
buffer_a := CREATE_BUFFER ('A', 'user$disk:[]test.doc');
window_a := CREATE_WINDOW (12, 10, ON);
MAP (window_a, buffer_a);
```

- MODIFIABLE WINDOW ATTRIBUTES
  - MULTIPLE VIDEO ATTRIBUTES IF YOUR TERMINAL SUPPORTS THEM
  - BLANK PADDING TO GIVE A 'BOXED' APPEARANCE
  - SCROLLING REGION BOUNDARIES AND AMOUNT
  - STATUS_LINE TEXT, VIDEO AND WHETHER OR NOT IT IS PRESENT
  - TEXT DISPLAY (BLANK_TABS, GRAPHIC_TABS OR NO_TRANSLATE)

- SPECIAL WINDOWS
  - INFO_WINDOW
    - * DISPLAYS 'SHOW' OR HELP INFORMATION
  - WINDOW(S) TO WHICH MESSAGE BUFFER IS MAPPED
    - * DISPLAYS TPU MESSAGES AND BROADCAST MESSAGES

T P U and T H E S C R E E N

- o THE TPU ENGINE CAN BE USED FROM MANY TYPES OF INPUT DEVICES

- o TPU SUPPORTS SCREEN-ORIENTED EDITING ON ANSI STANDARD CRT'S ONLY

- o INVISIBLE EDITING IS POSSIBLE

  - POSITION (window_a) NOT EQUAL TO POSITION (buffer_a)

- o SCREEN UPDATES DO NOT OCCUR AFTER EVERY BUILT-IN

  - FORCE UPDATING USING:

    UPDATE (window_a)

- o MODIFIABLE SCREEN ATTRIBUTES

  - TURN SCREEN_UPDATING ON OR OFF
  - TURN AUTO_REPEAT ON OR OFF (USED TO SLOW DOWN CURSOR, MINIMIZE FLYWHEELING)
  - DISPLAY A WORKING (TIMER) MESSAGE
  - RING BELL WHEN MESSAGES ARE DISPLAYED

T P U and T H E S C R E E N

- PROMPT_AREA IS A DEFINABLE REGION WHICH IS USED BY SEVERAL BUILT-INS TO DISPLAY PROMPTS

  (Example: READ_LINE)

  - A SINGLE VIDEO ATTRIBUTE CAN BE APPLIED
  - CAN OCCUPY MULTIPLE SCREEN LINES
  - CAN BE DEFINED TO OVERLAP A WINDOW OR EXIST AS A COMPLETELY SEPARATE SCREEN AREA
  - CAN BE EMULATED USING A BUFFER AND WINDOW

    * ADVANTAGES:

      - CAN SAVE A LOG OF THE COMMANDS
      - ALLOWS FOR EDITING OF PREVIOUS COMMANDS AND REPLAY

    * DISADVANTAGES:

      - SLIGHTLY SLOWER, ESPECIALLY WHEN USING VIDEO ATTRIBUTES

M A R K E R S

- REPRESENT A CHARACTER POSITION IN A BUFFER
- MOVE WITH THE ASSOCIATED CHARACTER
- CAN HAVE AN ASSOCIATED VIDEO ATTRIBUTE
- IF THE MARKER CHARACTER IS DELETED THEN THE MARKER BECOMES ASSOCIATED WITH THE NEXT CLOSEST CHARACTER
- MOST COMMON USES ARE AS BOOKMARKS OR TEMPORARY PLACEHOLDERS

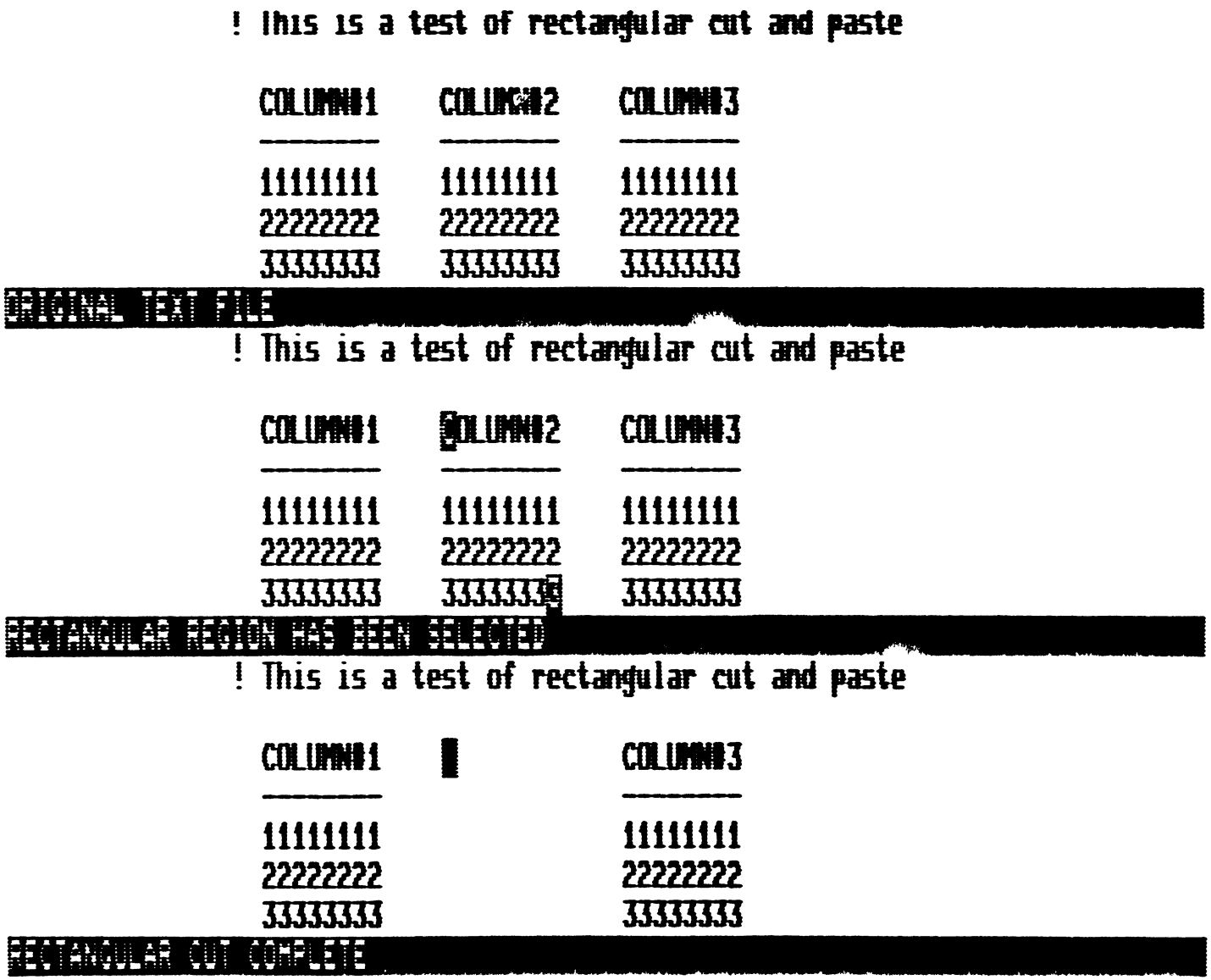
    * mark1 := MARK (NONE)
    * POSITION (mark1)
    * IF MARK (NONE) <> BEGINNING_OF (CURRENT_BUFFER)

RANGES

- REPRESENT THE TEXT BETWEEN AND INCLUDING TWO MARKERS
- ERASING A RANGE ERASES THE CHARACTERS OF A RANGE BUT DOES NOT REMOVE THE RANGE STRUCTURE
- CAN OVERLAP AND HAVE VIDEO ATTRIBUTES
- EXIST INDEPENDENTLY OF CURSOR MOTION
- MULTIPLE RANGES CAN BE ACTIVE AT ANY GIVEN TIME

SELECT RANGES

- SPECIAL RANGES WHICH :
  - BEGIN WITH A SELECT_MARK WITH AN ASSOCIATED VIDEO_ATTRIBUTE
  - INCLUDE OR EXCLUDE CHARACTERS BASED ON CURSOR POSITIONING
  - END BY CREATING A SELECT_RANGE
  - CAN OVERLAP OTHER RANGES
- ONLY ONE SELECT RANGE ACTIVE AT ANY GIVEN TIME


! This is a test of rectangular cut and paste
COLUMN#1 COLUMN#2 COLUMN#3
11111111 11111111 11111111
22222222 22222222 22222222
33333333 33333333 33333333
ORIGINAL TEXT FILE
! This is a test of rectangular cut and paste
COLUMN#1 COLUMN#2 COLUMN#3
11111111 11111111 11111111
22222222 22222222 22222222
33333333 33333333 33333333
RECTANGULAR REGION HAS BEEN SELECTED
! This is a test of rectangular cut and paste
COLUMN#1 COLUMN#3
11111111 11111111
22222222 22222222
33333333 33333333
RECTANGULAR CUT COMPLETE

P A T T E R N S

- A DATA TYPE WHICH USES THE |,&,@ OPERATORS TO CREATE A SIMPLE OR COMPLICATED TEXT 'STRUCTURE' USED AS AN ARGUMENT TO THE SEARCH BUILT-IN

- PATTERNS ARE RETURNED BY THE FOLLOWING BUILT-INS:

- ANY	- REMAIN
- ARB	- SCAN
- LINE_BEGIN	- SCANL
- LINE_END	- SPAN
- NOTANY	- SPANL

- PATTERNS ARE ALSO RETURNED BY ASSIGNMENT STATEMENTS WHICH USE THE |,&,@ OPERATORS

Example

```
patl := 'abc' & ('d' | 'e')
```

P A T T E R N S

- SEARCH IS THE ONLY BUILT-IN THAT ACCEPTS A PATTERN ARGUMENT
  - CAN BE MATCHED IN EITHER THE FORWARD OR REVERSE DIRECTION
- THREE TYPES OF SEARCH
  - ANCHOR
    - ANCHORS THE SEARCH AT THE CURRENT CHARACTER POSITION

('a'|'b'|'c') & 'd' looks for 'ad', 'bd', 'cd'

  - SEEK SEARCH (DEFAULT)
    - Looks for first possibility from current position to end of buffer, then looks for second possibility, and so on
    - First match in string 'cd bd ad' is 'ad'
  - INCREMENTAL SEARCH
    - Looks for all possibilities at current position then moves to next character and looks for all possibilities, and so on
    - First match in string 'cd bd ad' is 'cd'

PATTERN MATCHING EXAMPLE

o EDT WORD MOTION PATTERN

```
edt$v_forward_word :=

! Don't move off current character position
   (ANCHOR &

! If on eol, then match that
   ((LINE_END) |

! Leading spaces, on a word delimiter
   ((SPAN(' ') ) & (ANY (edt$v_word) | '') ) ) |

! No leading spaces
! On a word delimiter, move one past it
   (ANY (edt$v_word)) |

! No leading spaces
! On a real word, go one beyond it
   (SCAN (edt$v_word)) |

   ! No leading spaces
! On last real word of line, match rest of line
   REMAIN) &

! After matching, skip over trailing spaces if any
! except if match occurred at the eol.
! In this case, don't skip over blanks
   (LINE_BEGIN | SPAN(' ') | '');

search_range := SEARCH (edt$v_forward_word, FORWARD);
```

SUBPROCESSING

o CREATE A TPU PROCESS AND SEND COMMANDS TO IT

- Example

```
p1 := CREATE_PROCESS (process_buffer, '$ show quota');
MAP (process_window, process_buffer);
SEND ('$ show que sys$batch/all', p1);
p1 := 0;
UNMAP (process_window);
```

- USEFUL FOR

  * READING MAIL FROM WITHIN THE EDITOR
  * GETTING THE STATUS OF BATCH JOBS WITHOUT LEAVING THE EDITOR
  * SAVING A LOG OF YOUR DCL SESSION IN A BUFFER WHICH ALLOWS FOR SELECTION AND REPLAY OF PREVIOUS COMMANDS OR GROUPS OF COMMANDS

S U B P R O C E S S I N G

o CREATE A VMS PROCESS WITH SPAWN AND ATTACH

- SPAWN SUSPENDS THE TPU PROCESS, CREATES A NEW VMS SUBPROCESS AND ATTACHES YOU TO THE NEW PROCESS

- DCL ATTACH OR LOGOUT RETURNS YOU TO THE SUSPENDED TPU PROCESS

- ATTACH ('process_name') SUSPENDS THE CURRENT TPU PROCESS AND SWITCHES CONTEXT TO THE PREVIOUSLY CREATED PROCESS THAT YOU SPECIFY

- DCL SPAWN RETURNS YOU TO THE SUSPENDED TPU PROCESS

- USEFUL FOR

  * USING DCL FROM OUTSIDE OF THE EDITOR WHILE RETAINING THE CONTEXT OF THE EDITING SESSION

- WHEN CONTROL IS RETURNED

  * SCREEN REFRESHED
  * CURSOR POSITIONED TO PREVIOUSLY VISIBLE LOCATION

VARIABLES AND IDENTIFIERS

- GLOBAL AND LOCAL VARIABLES
  - THERE IS NO EXPLICIT GLOBAL DECLARATION
  - IF NOT DECLARED LOCAL, THEN BY DEFAULT THEY BECOME GLOBAL
- THERE IS NO DECLARATION STATEMENT FOR DECLARING THE 'TYPE' OF THE VARIABLE
- VARIABLES ON THE LHS OF AN ASSIGNMENT STATEMENT ASSUME A DATA TYPE BASED ON THE DATA TYPE OF THE RHS EXPRESSION
- MIXING OF DATA TYPES RESULTS IN AN ERROR MESSAGE (NO IMPLICIT TYPE CONVERSION)
- EXAMPLES
  - ! VALID: new_offset has the data type integer

```
new_offset := CURRENT_OFFSET + 20
```

  - ! INVALID: cannot mix strings with integers

```
bad_variable := 'hello' + 10;
```

TPU PROCEDURES

- APPROXIMATELY 100 BUILT-IN PRIMITIVES

- TPU'S DEFAULT COMMAND LINE INTERFACE IS PROCEDURAL
  - NO ABBREVIATIONS
  - MUST INCLUDE ALL PUNCTUATION (INCLUDING PARENTHESES AND COMMAS)

- PROCEDURES COMBINE TPU LANGUAGE STATEMENTS WITH CALLS TO BUILT-INS

```
PROCEDURE display_batch_stats
dcl_buffer := CREATE_BUFFER ('DCL BUFFER');
dcl_window := CREATE_WINDOW (1, 5, OFF);
SET (VIDEO, dcl_window, REVERSE);
SET (PAD, dcl_window, ON);
dcl_process := CREATE_PROCESS (dcl_buffer,
                  '$show que sys$batch/all');
MAP (dcl_window, dcl_buffer);
DEFINE_KEY ('UNMAP (dcl_window)', F11);
ENDPROCEDURE
```

```
PROCEDURE  identifier [(PARAMETER-LIST)]

[LOCAL-VARIABLE-LIST]

[ON_ERROR-STATEMENT]

        .
        .
        .
REPETITIVE-STATEMENTS; ----------------+
CONDITIONAL-STATEMENTS;                | Any Combination
CASE-STATEMENTS;                       | Any Order
MISCELLANEOUS-STATEMENTS; ---------+
        .
        .
        .

ENDPROCEDURE
```

ON_ERROR STATEMENTS

```
ON_ERROR
    STATEMENT-1;
        .
        .
        .
    STATEMENT-n;
ENDON_ERROR
```

- ON_ERROR STATEMENTS GAIN CONTROL WHENEVER ANY BUILT-IN USED WITHIN THE PROCEDURE RETURNS A WARNING OR ERROR STATUS
- IF THE STATUS IS WARNING, THEN YOU CAN USE THE ON_ERROR TO SUPPRESS THE DISPLAY OF A WARNING MESSAGE OR TO SUBSTITUTE YOUR OWN MESSAGE
- IF THE STATUS IS AN ERROR, AN ERROR MESSAGE IS DISPLAYED, BUT YOU CAN USE THE ON_ERROR TO DO ADDITIONAL PROCESSING
- COMMON STATEMENTS WHICH ARE USED TO CHANGE THE FLOW OF CONTROL ARE:
  - RETURN
  - ABORT
- AFTER THE ON_ERROR STATEMENT HAS FINISHED PROCESSING, EXECUTION CONTINUES FROM THE POINT IN THE CODE FOLLOWING THE STATEMENT WHICH CAUSED IT TO BE EXECUTED (unless RETURN or ABORT is used)

W R I T I N G T P U P R O C E D U R E S

- o IDENTATION (FORMATTING, SPACING) IS FREE STYLE
- o SEMICOLONS SEPARATE STATEMENTS
- o COMMENT STATEMENTS (which begin with !) CAN APPEAR ANYWHERE
  - WHEN ! IS SEEN, THE REST OF THE LINE IS IGNORED
- o TO RETURN A RESULT
  - RETURN value
  - procedure_name := value

REPETITIVE STATEMENTS

```
LOOP
    EXITIF boolean-expression;
    STATEMENT-1;
          .
          .
          .
    STATEMENT-n;
ENDLOOP;
```

- THE EXITIF STATEMENT CAN APPEAR ANYWHERE INSIDE THE LOOP

- THE ON ERROR STATEMENT CAN SOMETIMES BE USED TO EXIT FROM A LOOP PROVIDED THAT BUILT-INS WHICH RETURN EITHER WARNING OR ERROR STATUS ARE CALLED FROM INSIDE THE LOOP (Example: SEARCH)

C O N D I T I O N A L S T A T E M E N T S

```
IF boolean-expression
THEN
    STATEMENT-1;
          .
          .
          .
    STATEMENT-n;
[ELSE
    STATEMENT-1;
          .
          .
          .
    STATEMENT-n;]
 ENDIF;
```

- TRUE IS INTEGER ZERO (0) (NO RESERVED WORD TRUE)

- FALSE IS INTEGER ONE (1) (NO RESERVED WORD FALSE)

EXAMPLE

```
PROCEDURE global_search_replace (str_or_pat, str2)
!
! This procedure performs a search through the current
! buffer and replaces a string or pattern by a string

LOCAL src_range, replacement_count;

! Return to caller if string not found
___ ON_ERROR
|           msg_text := FAO ('Completed !UL  replacement!%S',
|                            replacement_count);
|           MESSAGE (msg_text);
|           RETURN;
|___ ENDON_ERROR

      replacement_count := 0;

! Loop until eob found or string not found
___ LOOP
|           src_range := SEARCH (str_or_pat, forward);
|           ! Search returns a range if found
|
|       ! Make sure it's not in a comment (call another routine
|       ! to avoid triggering the error handler)
|    __ IF is_a_comment <> 1
|    |  THEN
|    |       ! replace str1 with str2
|    |       ERASE (src_range);                ! Remove first string
|    |       POSITION (END_OF (src_range));    ! Move to right place
|    |       COPY_TEXT (str2);                 ! Replace with second string
|    |     replacement_count := replacement_count + 1;
|    |__ ENDIF;
|
|___ENDLOOP;
ENDPROCEDURE          ! global_search_replace
```

CASE STATEMENTS

```
CASE case-selector FROM case-lower TO case-upper

   [case-constant-1]: STATEMENT-1;... STATEMENT-n;
   [case-constant-2]: STATEMENT-1;... STATEMENT-n;
                      .
                      .
                      .
   [case-constant-n]: STATEMENT-1;... STATEMENT-n;

   [INRANGE]: STATEMENT-1;... STATEMENT-n;
   [OUTRANGE]: STATEMENT-1;... STATEMENT-n;

ENDCASE;
```

EXAMPLE:

```
menu_item := INT (READ_LINE (
                  'Enter Menu Selection (1-5): '));

CASE menu_item FROM 1 TO 5

   [1] : EXECUTE (proc_read_mail);
   [2] : SHOW (WINDOWS);
   [3] : UNMAP (INFO_WINDOW);
         POSITION (BEGINNING_OF (CURRENT BUFFER));
   [4] : EXECUTE (proc_show_batch);
   [5] : proc_help

   [OUTRANGE] : MESSAGE ('Invalid menu selection');

 ENDCASE;
```

T P U K E Y D E F I N I T I O N S

- KEY DEFINITIONS ALLOW YOU TO CUSTOMIZE THE KEYPAD INTERFACE

- COMBINE KEY MANIPULATION BUILT-INS TO DO USEFUL THINGS SUCH AS EMULATION OF THE EDT REPEAT COUNT

- KEY DEFINITION AND MANIPULATION COMMANDS
  - DEFINE_KEY (executable_code, key-name, 'comment')
  - LOOKUP_KEY (key-name, PROGRAM or COMMENT)
  - READ_KEY
  - UNDEFINE_KEY (key-name)
  - KEY_NAME (key-name, [SHIFT_KEY])
  - LAST_KEY

(See Handouts for examples)

WHAT IS A SECTION FILE

File mysection.tpu

```
! Comments
PROCEDURE my_proc_1
        .
        .                  ----------------> As many as needed
        .
ENDPROCEDURE

PROCEDURE my_key_definitions
DEFINE_KEY ('proc_rectangular_cut', F14);
        .
        .
        .
ENDPROCEDURE

PROCEDURE TPU$INIT_PROCEDURE
 !
 ! Initialize global variables
 ! Set up buffers and windows
 ! Look at the TPU command line to set up journaling,
 ! NO_WRITE buffers, and so on.
        .
        .
        .
ENDPROCEDURE
        .
        .
        .
! Include desired stand-alone executable statements
MY_KEY_DEFINITIONS;          ! Define the keys
SET (TIMER, ON, '...Working...');

SAVE ('sys$login:mysection.gbl');

QUIT;
```

WHAT IS A SECTION FILE

- ALL PROCEDURES MUST BE DEFINED BEFORE ANY EXECUTABLE STATEMENTS WHICH STAND ALONE

- THE SECTION IS CREATED BY USING THE SAVE BUILT-IN
  - STORES COMPILED PROCEDURES, KEY_DEFINITIONS, AND SYMBOL TABLE
  - MAKES START-UP SIGNIFICANTLY FASTER WHEN COMPARED TO USING A COMMAND FILE
    - * USING THE EDT EMULATOR INITIALIZATION FILE
      - AS A COMMAND FILE => approximately 50 seconds
      - AS A SECTION FILE => approximately 4 seconds

DIFFERENCES BETWEEN EVE AND THE EDT EMULATOR

	EDT EMULATOR	EVE INTERFACE
GOAL	MIGRATION PATH FROM EDT; USER EXPECTED TO DO OWN CUSTOMIZING	MEASUREABLY EASY TO LEARN AND FAST TO USE + USE ADVANCED EDITING FEATURES
KEYPAD	EDT KEYPAD	LK201 EDITING PLUS F10-F14
COMMAND LINE	PARSER FOR 6 CMDS USING CTRL-Z '*'; PF1/KP7->TPU Command: ENTER TPU COMMANDS USING TPU SYNTAX DIRECTLY (with parens, underscores)	74 COMMANDS; ALLOWS ACCESS TO TPU LANGUAGE WITH THE TPU COMMAND
ADVANCED FEATURES	ACCESS TO THEM USING THE TPU COMMAND LINE PROMPT OR BY USER WRITTEN PROCEDURES	PREPROGRAMMED WITH SOME BOUND TO KEYS AND OTHERS ACCESSIBLE USING AN ENGLISH LIKE LINE MODE
COMMAND PROCESSING	USES TPU PROMPT AREA	USES A BUFFER AND A WINDOW TO SAVE AN EDIT HISTORY; ALLOWS COMMAND RECALL

THE EVE INTERFACE TO TPU

o GOALS:

- MEASURABLY EASY TO USE AND EASY TO LEARN
- ITERATIVE DESIGN PROCESS, STARTING WITH EXISTING DATA ON TEXT EDITOR USAGE

o FEATURES:

- USES MULTIPLE FILES AND BUFFERS
- DEFINES TWO WINDOWS (ONE MAIN WINDOW IS THE DEFAULT)
- TEXT-ORIENTED FUNCTIONS SUCH AS AUTOMATIC WORD WRAPPING, SETTING MARGINS, FILLING A PARAGRAPH, AND CENTERING A LINE
- DEFINES A KEY FOR INSERT/OVERSTRIKE TOGGLE
- USES FREE CURSOR MOTION
- HAS THE ABILITY TO SET AND GO TO NAMED MARKERS IN MULTIPLE BUFFERS
- EXPLOITS VT200 KEYBOARD, BUT ALSO USABLE ON VT100
- PROVIDES FOR COMMAND-LINE EDITING AND ABBREVIATIONS USING A FANCY PARSING PROCEDURE
- PROVIDES ACCESS TO TPU AND DCL THROUGH SPECIALLY DEFINED COMMANDS
- ALLOWS EASY ACCESS TO TPU'S LEARN SEQUENCE CAPABILITIES

EXTENDING THE EVE INTERFACE

- START WITH AN EMPTY BUFFER.

- ENTER PROCEDURES AND EXECUTABLE STATEMENTS INTO THIS BUFFER (START THE PROCEDURE, ENDPROCEDURE STATEMENTS IN COLUMN 1).

- IF YOU WANT TO USE EVE'S COMMAND LINE TO ACCESS YOUR USER-WRITTEN PROCEDURES, PREFIX THEM WITH EVE_.

- PRESS 'Do', Command: EXTEND TPU *

- IF ERROR MESSAGES ARE DISPLAYED, USE 'Do',
  Command: BUFFER MESSAGES
  TO MAKE THE MESSAGES APPEAR IN THE TOP WINDOW.

- PRESS 'Do', Command: BUFFER MAIN, TO RETURN TO YOUR PREVIOUS EDITING CONTEXT AND CORRECT YOUR ERRORS.

- IF ERRORS ARE PRESENT, THE TPU COMMAND INTERPRETER WILL GIVE THE LINE NUMBER OF THE STATEMENTS IN ERROR.
  PRESS 'Do', Command: LINE n, TO MOVE THE CURSOR TO THE LINES TO BE CORRECTED.

- ONCE THE PROCEDURES ARE ERROR FREE YOU CAN TEST THEM BY ENTERING THE PROCEDURE NAMES USING 'Do'
  (omit EVE_ and other underscores from the procedure name).

- YOU CAN ALSO COMPILE ONE PROCEDURE AT A TIME FROM THE BUFFER BY ENTERING: EXTEND TPU complete_procedure_name,
  AT THE COMMAND LINE PROMPT.

- THERE ARE ADDITIONAL CONVENTIONS THAT MUST BE FOLLOWED WHEN PASSING ARGUMENTS TO EVE PROCEDURES

- CLEAR THE MAIN_BUFFER
- ENTER PROCEDURES AND EXECUTABLE STATEMENTS
- AFTER THE 'Tpu Command:' PROMPT, USE THE FOLLOWING TPU STATEMENTS:
  - SET (INFORMATIONALS, ON)
  - COMPILE (MAIN_BUFFER)
  - POSITION (MESSAGE_WINDOW)
  - POSITION (MAIN_WINDOW)
  - procedure_name ! For each procedure you want to test
  - EXECUTE (MAIN_BUFFER)
  - WRITE_FILE (MAIN_BUFFER, 'file_name')

WAYS TO ACCESS TPU

- AT DCL LEVEL

```
$ EDIT/TPU

$ EDIT/TPU/SECTION=EVESECINI file-name.ext

$ EDIT/TPU/NOSECTION/COMMAND=myini.tpu

$ EDIT/TPU/NOSECTION/COMMAND=BATCHINI.TPU/NODISPLAY

$ EVE :== EDIT/TPU/SECTION=EVESECINI.TPU
$ EVE file-name.ext
```

- FROM AN APPLICATION OR PROGRAMMING LANGUAGE

  - THROUGH THE CALLABLE INTERFACE

THE /NODISPLAY QUALIFIER

- ALLOWS YOU TO USE TPU IN BATCH MODE (SPECIAL INITIALIZATION FILES ARE RECOMMENDED)

- ALLOWS YOU TO PROGRAM A PSUEDO-LINE-MODE EDITING INTERFACE FOR 'UNSUPPORTED' DEVICES SUCH AS THE LA36

  - USES LIB$PUT_OUTPUT FOR READ_LINE AND MESSAGE BUILT-INS

- CAN BE USED IN A BATCH COMMAND FILE OR ENTERED INTERACTIVELY

(see Handouts for an example)

E X A M P L E

```
! Extract from a line mode editor for TPU
!
! Invoked from DCL via: TPU/NODISPLAY/NOSECTION/COM=linemode.tpu file-spec
!
! Set up the main editing buffer using the input file from the command line
!
input_file := GET_INFO (COMMAND_LINE, 'file_name');
main_buffer := CREATE_BUFFER ("MAIN", input_file);
POSITION (BEGINNING_OF (main_buffer));
!
LOOP        ! Continuously loop until QUIT
    cmd := READ_LINE ('*');

    IF cmd = ''
    THEN
        cmd_char := 'N';
    ELSE
        cmd_char := SUBSTR (cmd, 1, 1);
        CHANGE_CASE (cmd_char, UPPER);
    ENDIF;

    CASE cmd_char FROM 'I' TO 'T'     ! Only accepting I,L,N,Q,T

    !Top of buffer command
            ['T']:
                    POSITION (BEGINNING_OF (CURRENT_BUFFER));
                    MESSAGE (CURRENT_LINE);
    !Next line command
             ['N']:
                    MOVE_HORIZONTAL (-CURRENT_OFFSET);
                    MOVE_VERTICAL (1);
                    MESSAGE (CURRENT_LINE);
    !Insert text command
             ['I']:
                    SPLIT_LINE;
                    COPY_TEXT (SUBSTR (cmd, 2, 999));
                    MESSAGE (CURRENT_LINE);
    !List from here to end of file command
             ['L']:
                    m1 := MARK (NONE);
                    LOOP
                    MESSAGE (CURRENT_LINE);
                    MOVE_VERTICAL (1);
                    EXITIF MARK (NONE) = END_OF (CURRENT_BUFFER);
                    ENDLOOP;
                    POSITION (m1);
    !QUIT
             ['Q']:
                     QUIT;

             [INRANGE,OUTRANGE]:
                    MESSAGE ('Unrecognized command - enter I,L,N,Q or T');

    ENDCASE;

ENDLOOP;
```

Introduction to VAX ADA

VAX Ada (r)

In the April/May 1985 timeframe, Digital will begin shipping its newest language product for the entire VAX family -- VAX Ada.

WHAT IS ADA?

Ada is a modern higher order programming language which was designed as the result of a competition sponsored by the Department of Defense. The purpose of the competion was to design a language which would let the DoD reduce software costs by increasing software maintainability, evolvability, reliability, and portability. Ada is suitable for a variety of applications, including systems programming, computational programming, general programming, and especially real-time programming. Besides providing powerful programming language features, Ada is able to reduce software lifecycle costs by providing for modularization and separate compilation using packages, scope rules, and a compilation data base. Ada allows both bottom-up and top-down software development. Ada enhances software reliability through strong typing. The DoD was particularly interested in a suitable language for embedded real-time applications, for example, weapons systems. Ada provides language features for multi-tasking such as tasks, rendezvous, priorities, and entry calls.

Ada was approved as an ANSI standard in February 1983. The standard is rigorous; no subsets or extensions are allowed. Processor dependent features are permitted only in a very limited controlled context. Conformance to the standard is enforced by a large set of validation tests. The name Ada is a registered trademark of DoD and cannot be used without their permission.

VAX Ada fully conforms to the ANSI-83 Ada standard. As the newest member of the VAX/VMS and MicroVMS family of languages, the product also offers full integration with the Common Language Development Environment. The combination of the language features of VAX Ada and its integration with VAX/VMS and MicroVMS systems create a total environment for programmer productivity and reduced software lifecycle costs. And, as with other VAX languages, applications can be moved from the smallest MicroVAX system to the largest VAX system or cluster.

(r) Ada is a registered Trademark of the U.S. Government (Ada Joint Program Office)

MAJOR FEATURES OF THE ADA PROGRAMMING LANGUAGE

Although the initial interest in Ada results from the strong push which DoD is giving it in the mission critical area, the features of the language assure its acceptance in a much larger market:

- Strong typing -- Type checking is performed at compile time, detecting errors associated with data types and reducing program development time.

- Data abstraction -- Ada provides various levels of abstraction through data typing and the "package" mechanism.

- Concurrent processing -- Parallel activities are supported directly using constructs that are part of the language itself.

- Separate compilation -- Ada's separate (rather than independent) compilation feature lets a programmer divide a large program into compilation units that may be compiled at different times.

- Generic definitions -- A generic unit is a template of an algorithm. Specific instances of a generic unit can be defined at compilation time by supplying actual type, subprogram and/or object parameters to create a subprogram or package tailored to those parameters.

- Exception handling -- Ada includes predefined exception conditions and also allows the user to define exceptions.

- Machine Dependent Facilities -- A package called SYSTEM is provided which contains a collection of system defined constants to represent machine dependent information. Further, Ada provides the ability to refer to low-level machine-dependent features in high-level terms, using "representation specifications."

FEATURES OF VAX Ada

In addition to all standard Ada language features, Digital's VAX Ada provides a rich and robust programming support environment:

VAX Ada Features

FEATURE	DESCRIPTION
Government Validated Compiler	VAX Ada has successfully passed the latest set (V1.4) of over 2000 tests which compose the official government-mandated Ada Validation test suite; the test suite is a rigorous test of a product's conformance to the ANSI-MIL-STD-1815A-1983 for the Ada programming language.
Program Library Manager	Support is provided for teams of programmers through an Ada program library manager. In addition to providing the required support for Ada dependent language features, the library manager provides many useful features for managing Ada programming projects. This facility, for example, allows shared use of a program library by multiple programmers, the ability to share compiled Ada code either by reference or copy, the use of individual libraries as sublibraries of team libraries, and the automatic recompilation of obsolete units.
VMS integration	VAX Ada is integrated into the VAX/VMS common language environment. All utilities and system services are available to programs written in Ada. VAX Ada programs can also handle VMS asynchronous system traps (ASTs) and be linked with shared images and use shared global sections. VAX Ada supports VAX Record Management Services (RMS) including sequential, relative, and indexed file organizations and associated access methods.

VAX Ada Features (con't)

FEATURE	DESCRIPTION
Full Symbolic Debug Support	High level, fully symbolic debugging capability through the VMS debugger is provided for the Ada programmer. This capability includes support for multitasking, packages, and mixed Ada and non-Ada code.
Multilanguage Capability	VAX Ada conforms to the VAX Calling Standard, which provides the ability to call and be called by code written in other languages. VAX Ada is also able to handle exceptions from non-Ada code, generate exceptions to be handled by non-Ada code, and share data with non-Ada code through global variables and psects (common blocks).
Optional Ada Language Features	VAX Ada supports almost all the optional language features defined in the ANSI Ada Language Reference Manual: predefined PRAGMAs and attributes, unchecked conversion and unchecked deallocation, and representation specifications. Representation specifications allow programs to refer to low-level, machine dependent features in high-level terms. These specifications tell the compiler how an entity is to be implemented, or 'mapped' to the underlying machine.
Comprehensive Diagnostics	Diagnostic messages, including automatic syntax error correction, are geared to help the new Ada user.

VAX ADA Technical Summary

VAX[tm] Ada[r]*

Technical Summary
(Preliminary)

Digital Equipment Corporation
November 1984

PREFACE

This document provides technical information about Digital Equipment Corporation's implementation of Ada for VAX/VMS. Information is organized according to the "Ada-Europe Guidelines for Ada compiler specification and selection". Questions from the guidelines are not restated; rather, topics are discussed in a manner that makes it unnecessary to refer to the original questions. The presentation is intentionally terse to present the most information possible in the least space. Supplementary information is also provided as appropriate. An appendix shows listings from two small compilations to help answer many of the questions related to compilation listings and error messages.

The "Ada-Europe Guidelines for Ada compiler specification and selection" was developed in 1982 by J. C. D. Nissen, B. A. Wichmann and other members of Ada-Europe, with partial support from the Commission of the European Communities. It provides a detailed checklist of questions about an Ada implementation designed to assist users in evaluating Ada implementations and to assist vendors in describing Ada implementations.

The "Ada-Europe Guidelines" is available from the National Physical Laboratory as NPL Report DITC 10/82, ISSN 0262-5369. It was also reprinted in Ada Letters, Vol. III, No. 1 (July, August 1983), pp. 37-50. (Ada Letters is a bimonthly publication of the ACM Special Interest Group on Ada.)

---

tm VAX is a trademark of Digital Equipment Corporation.

r Ada is a registered trademark of the U.S. Government, Ada Joint Program Office.

* The information presented applies to the field test version of VAX Ada Version 1. The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation.

CONTENTS

1 INTRODUCTION

VAX Ada is a full, production-quality implementation of Ada that is well-integrated in the VAX/VMS environment, where the VAX/VMS operating system serves as both host and target. The main components of VAX Ada are:

- Ada Compiler
- Ada Program Library Manager (ACS)
- Ada Run-Time Library
- Ada Library of Predefined Units
- Symbolic Debugger Support
- Comprehensive Documentation

Each of these components is described in detail in the following sections.

2 HOST AND TARGET ENVIRONMENTS

The host hardware includes the VAX 8600, VAX-11/785, VAX-11/780, VAX-11/782, VAX-11/750, VAX-11/730,* VAX-11/725, MicroVAX I and future members of the VAX family. The host operating system is VAX/VMS, Version 4.0 or later, and MicroVMS, Version 1.0 or later (for the MicroVAX I). (MicroVMS is a repackaged variant of VAX/VMS intended for systems with limited disk space. It is not a subset and provides the full power and functionality of VAX/VMS.)

Target configurations are the same as the host configurations. Programs can be moved from any host to any target (when different) in source, object or executable image format, using any common media (for example, disk, magtape) or communication medium (for example, using DECnet). Complete Ada program libaries can also be moved in the same manner.

The VAX "Ada Programming Support Environment" (APSE) consists of the VAX common language environment and the Ada program library management utility (ACS). The VAX/VMS common language environment includes a variety of existing and planned languages and tools which all work together. Tools and utilities provided with VAX/VMS include the VAX Symbolic Debugger (DEBUG), Record Management Services (RMS), the common Run-Time Library (RTL), the VAX/VMS file system, and the DIGITAL Command Language (DCL).

---

* VAX Ada is not supported on the VAX-11/730 with the tailored dual RL02 system.

The ACS is part of the VAX Ada product and provides support for Ada program library requirements, as well as various project-oriented utility operations.

Optionally available tools and utilities include a source code management system (CMS), a module management system (MMS), a language sensitive editor (LSE, expected availability spring 1985), a test manager (DTM, available fall 1984), a performance and test coverage analyzer (PCA, expected availability spring 1985), and various VAX Information Architecture (VIA) products.

3 LANGUAGE-RELATED ISSUES

The Ada Standard (ANSI/MIL-STD-1815A-1983) explicitly allows variations among Ada processors in a number of dimensions. This section describes the language supported by VAX Ada and is organized according the applicable chapters of the LRM.*

3.1 Chapter 2: Lexical Elements

The VAX/VMS character set is an 8-bit superset of ISO standard 646, known as the DEC Multinational Character Set. The extended characters cannot, of course, be used in an Ada source program; similarly, the extended characters cannot be read or written as values of the Ada predefined type CHARACTER. However, the full character set can be used under the guise of an integer type.

The maximum number of characters in a source line is 120. This also limits the length of identifiers to 120 characters.

Input-output in executing Ada programs is limited only by the length of records supported by the VAX/VMS operating system: 32,767 bytes.

VAX Ada supports the language-defined pragmas CONTROLLED, ELABORATE, INLINE, INTERFACE, LIST, MEMORY_SIZE, OPTIMIZE, PACK, PAGE, PRIORITY, and STORAGE_UNIT. INLINE and INTERFACE are subject to certain restrictions (as discussed later).

The predefined pragmas SHARED and SUPPRESS are not supported. The VAX Ada pragma VOLATILE is similar to pragma SHARED but does not guarantee synchronization and can be used with objects of any type. The VAX Ada pragma SUPPRESS_ALL provides the ability to suppress all checking, but VAX Ada does not support selective disabling of checking by means of pragma SUPPRESS.

---

* LRM refers to the Ada Standard: "Reference Manual for the Ada Programming Language", ANSI/MIL-STD-1815A-1983.

VAX Ada also provides the following implementation-defined pragmas (their definitions are given in the VALRM* sections shown):

AST_ENTRY	(9.12a)
EXPORT_EXCEPTION	(13.9a.3.2)
EXPORT_FUNCTION	(13.9a.1.3)
EXPORT_OBJECT	(13.9a.2.2)
EXPORT_PROCEDURE	(13.9a.1.3)
IMPORT_EXCEPTION	(13.9a.3.1)
IMPORT_FUNCTION	(13.9a.1.1)
IMPORT_OBJECT	(13.9a.2.1)
IMPORT_PROCEDURE	(13.9a.1.1)
IMPORT_VALUED_PROCEDURE	(13.9a.1.1)
LONG_FLOAT	(3.5.7a)
PSECT_OBJECT	(13.9a.2.3)
SUPPRESS_ALL	(11.7)
TASK_STORAGE	(13.2a)
TIME_SLICE	(9.8a)
TITLE	(8)
VOLATILE	(9.11)

Each of these is briefly described in later sections.

3.2 Chapter 3: Declarations And Types

The maximum number of elements in an enumeration type is 65,535.

There are three predefined integer types: INTEGER, SHORT_INTEGER and SHORT_SHORT_INTEGER. Their first and last values are:

Type	T'FIRST	T'LAST
INTEGER	-(2**31)	(2**31)-1
SHORT_INTEGER	-(2**15)	(2**15)-1
SHORT_SHORT_INTEGER	-(2**7)	(2**7)-1

Type LONG_INTEGER is not supported in VAX Ada.

There are three predefined floating point types in package STANDARD: FLOAT, LONG_FLOAT and LONG_LONG_FLOAT. In addition, there are four predefined floating point types in package SYSTEM: F_FLOAT, D_FLOAT, G_FLOAT and H_FLOAT. All are implemented using one of the four VAX hardware floating point representations: F_floating (32 bits), D_floating (64 bits), G_floating (64 bits) and H_floating (128 bits). Type LONG_FLOAT uses the G_floating representation by default, but this can be changed to D_floating using pragma LONG_FLOAT.

---

* VALRM refers to DIGITAL's "VAX Ada Language Reference Manual", which includes both the full text of the LRM and VAX Ada specific supplements.

Type	T'DIGITS	Representation
FLOAT	6	F_floating
LONG_FLOAT		D or G_floating
using D_floating	9	
using G_floating (default)	15	
LONG_LONG_FLOAT	33	H_floating
F_FLOAT (in SYSTEM)	6	F_floating
D_FLOAT (in SYSTEM)	9	D_floating
G_FLOAT (in SYSTEM)	15	G_floating
H_FLOAT (in SYSTEM)	33	H_floating

Type SHORT_FLOAT is not supported for VAX Ada.

In addition to type DURATION, there are 63 anonymous predefined fixed point types. All fixed point types are implemented as 32-bit scaled binary values.

The compiler does not flag uninitialized variables, nor does it reject a program that depends on them.

3.3 Chapter 4: Names And Expressions

Static expressions of type universal_integer have no limit on the implemented range. Similarly, static expressions of type universal_real have no limit on the implemented accuracy or range. Evaluation of such expressions during compilation uses a general universal arithmetic package.

Nonstatic expresions of type universal_integer are evaluated during execution using 32-bit signed binary representation, and hence have a range limited to -2147483648 .. 2147483647. Nonstatic expressions of type universal_real are evaluated during execution using the H_floating representation, and hence are limited to an accuracy of approximately 33 digits and a range of approximately -5.9*(10**4931) .. 5.9*(10**4931).

3.4 Chapter 9: Tasks

The properties of type DURATION are as follows:

DURATION'DELTA	1.00000e-04	(10 milliseconds)
DURATION'SMALL	2.0**(-14)	(approximately 61 microseconds)
DURATION'FIRST	-131072.0000	(approximately
DURATION'LAST	131071.9999	36.4 hours)

Subtype PRIORITY has the following range:

```
PRIORITY'FIRST          0
PRIORITY'LAST          15
```

A task for which no pragma PRIORITY is specified has a default priority of 7.

VAX Ada does not support pragma SHARED. It does support the implementation-defined pragma VOLATILE, which guarantees that a variable will be in memory, but does not force synchronization. Pragma VOLATILE can be used with variables of any type, including composite variables.

3.5 Chapter 10: Program Structure And Compilation Issues

A VAX Ada program is initiated using the DCL RUN command.

Any library subprogram can be a main subprogram provided that it has no parameters, and, in the case of a function, returns a value of a discrete type.

Information can be communicated to an Ada program using command line parameters (using the system command line interface callback facilities), logical names, global sections, mailboxes and, of course, file and/or interactive input-output facilities. (Note: these are all general VAX/VMS capabilities and are in no way specific to VAX Ada.)

VAX Ada programs can be linked with and can call routines written in other VAX languages; conversely, Ada subprograms can be called from programs written in other VAX languages. The main program need not be written in the Ada language.

Tasks initiated in library packages follow the same rules for task termination as other tasks. In particular, they are not terminated simply because the main program terminates. Terminate alternatives in selective wait statements in library tasks are recommended.

3.6 Chapter 11: Exceptions

NUMERIC_ERROR is raised for integer or floating point overflow, and integer or floating point divide-by-zero; it is not raised for floating point underflow (the value zero results).

PROGRAM_ERROR is raised as required in the LRM; it is also raised in a few instances that are not required. PROGRAM_ERROR is not raised for incorrect order dependences.

STORAGE_ERROR is raised as required in the LRM.

VAX Ada provides (in package SYSTEM) the exception NON_ADA_ERROR, which allows an Ada exception handler to handle VAX conditions raised in imported code. Ada exceptions can also be handled by routines written in other languages. The continuation option of the VAX Condition Handling Facility (CHF) is not available to Ada subprograms, nor is it available for an exception that is reraised by an Ada subprogram.

3.7 Chapter 13: Representation Clauses And Implementation Dependent Features

Generally, VAX Ada supports all implementation-dependent facilities of Chapter 13 that have a useful interpretation in the VAX/VMS environment.

Pragma PACK is supported.

For a size specification for a discrete type, the given size must not exceed 32 bits; the given size becomes the default allocation for all objects and components of that type. For all other types, the given size must equal the size that would apply in the absence of a size specification.

For a collection size specification, the given size becomes the initial and maximum size of the collection. In the absence of a collection size specification, or for a size specification of zero, no storage is initially allocated for a collection, and the collection is extended as needed (until all virtual memory for the process is exhausted). If the value is less than zero, CONSTRAINT_ERROR is raised.

For a task activation specification, the given size becomes the initial and maximum size for the task activation (the task stack size). In the absence of a specification, or for a specification of zero, a default size is used. In either case, the task stack size is fixed at activation and is not extendable. If the value is less than zero, CONSTRAINT_ERROR is raised.

For the specification of small for a fixed point type, the given value must be a power of 2.0 (2.0**N, where -31 <= N <= 31) that is less than or equal to the delta of the type.

Enumeration representation clauses are supported.

Record representation clauses are supported. The value in an alignment clause must be a power of 2 (2**N, where 0 <= N <= 9). For stack objects, the alignment must not exceed 4 (longword alignment). For statically and collection-allocated (heap-allocated) objects, alignments up to 512 are supported.

VAX Ada distinguishes types that are bit-alignable and byte-alignable. Components of bit-alignable types can be allocated beginning at arbitrary bit offsets in component clauses, while components of byte-alignable types must be allocated at byte (addressable storage) boundaries. Generally, discrete types, arrays of discrete types and record types whose size is 32 bits or less are bit-alignable, while other types are not. (The exact rules are a little more complicated; see the "VAX Ada Programmer's Run-Time Reference Manual" for full details.)

VAX Ada generates no implementation-dependent components or names for record types.

VAX Ada does not support address specifications. (The implementation-defined attribute AST_ENTRY and pragma AST_ENTRY can be used to map a VAX/VMS asynchronous system trap (AST) to an entry call.)

VAX Ada provides various additional declarations in the predefined package SYSTEM to facilitate system programming in the VAX/VMS environment. These include the basic system-dependent named numbers, whose values are given as follows:

```
MIN_INT          -(2**31)
MAX_INT           (2**31)-1
MAX_DIGITS            33
MAX_MANTISSA          31
FINE_DELTA             2.0**(-30)
TICK                  10.0**(-2)
```

Type ADDRESS is a private type for which conventional "unsigned" addition, subtraction and relational operations are provided. Generic subprograms FETCH_FROM_ADDRESS and ASSIGN_TO_ADDRESS provide the ability to read from or write to memory at an arbitrary address according to the instantiated type.

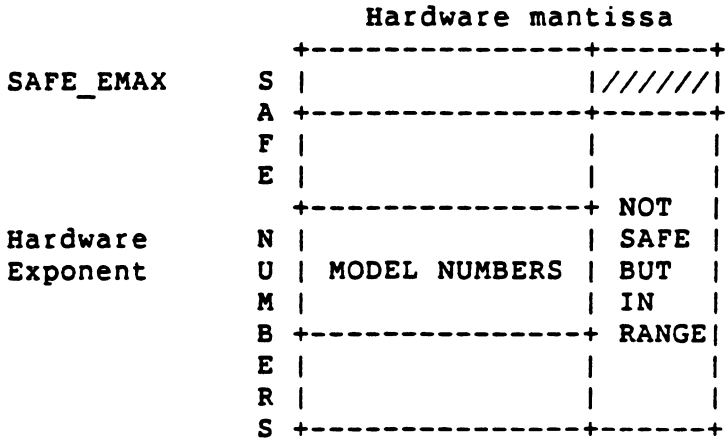
Other constituents of package SYSTEM are detailed in the VALRM.

The representation attributes ADDRESS, SIZE, POSITION, FIRST_BIT, LAST_BIT and STORAGE_SIZE are supported. The implementation-defined attribute BIT yields the bit offset within a storage unit of the first bit allocated to an object (a value from 0 to 7).

The values of the representation attributes for floating point types are summarized as follows:

Attribute	Floating Point Representation F	D	G	H
MACHINE_RADIX	2	2	2	2
MACHINE_MANTISSA	24	56	53	113
MACHINE_EMAX	127	127	1023	16383
MACHINE_EMIN	-127	-127	-1023	-16383
MACHINE_ROUNDS	TRUE	TRUE	TRUE	TRUE
MACHINE_OVERFLOWS	TRUE	TRUE	TRUE	TRUE

For each hardware floating point type, there are representable numbers that are not safe numbers, and there is a small set of numbers that are outside the range of safe numbers, as indicated in the shaded region of the following diagram. For F, D, G and H_floating respectively, the number of decimal digits in the safe numbers is 6, 9, 15 and 33, and the safe maximum exponents are 127, 127, 1023 and 16383.



Machine code insertions are not supported.

Pragma INTERFACE is supported and may be used with routines of any language that adheres to the VAX Procedure Calling Standard. (The language name given in this pragma is ignored.)

When pragma INTERFACE applies to more than one subprogram, it must be supplemented by implementation-defined IMPORT_FUNCTION, IMPORT_PROCEDURE and/or IMPORT_VALUED_PROCEDURE pragmas to identify the distinct foreign routines corresponding to each internal subprogram declaration. These pragmas may also be used to specify the the parameter passing mechanism to be used (by value, by reference or by descriptor) when the mechanisms required by the external routine differ from the default mechanisms that would be used by VAX Ada. Pragma IMPORT_VALUED_PROCEDURE allows "functions" in the environment that also update their parameters to be declared and called as procedures within Ada (functions in Ada are not allowed to have

'in out' or 'out' parameters).

The implementation-defined pragma IMPORT_OBJECT allows an externally declared variable to be associated with an Ada variable. Conversely, pragma EXPORT_OBJECT allows an Ada variable to be accessed in external routines. Finally, pragma PSECT_OBJECT allows an Ada variable to be associated with an external program section ("psect"), and thus made to correspond to a VAX FORTRAN common block or a PL/I external variable. These pragmas are generally restricted to use with variables declared in library packages (which are statically allocated).

The implementation-defined pragma IMPORT_EXCEPTION allows a specific VAX condition value (used for implementing exceptions in VAX/VMS) to be associated with an Ada exception. Conversely, pragma EXPORT_EXCEPTION allows an Ada exception to be exported as a VAX condition value.

For all of the import and export pragmas described above, additional rules and restrictions apply. See the VALRM for full details.

Unchecked deallocation is supported. (Pragma IMPORT_PROCEDURE cannot be used to substitute a user-defined procedure.)

Unchecked conversion is supported, subject to the following: The actual subtype corresponding to the formal type TARGET must not be an unconstrained array type or an unconstrained type with discriminants. Further, when the target type is a type with discriminants, the value resulting from a call of the conversion function resulting from an instantiation of UNCHECKED_CONVERSION is checked to assure that the discriminants satisfy the constraints of the actual subtype. For an unchecked conversion, when the size of the source value is not the same as the target subtype, the value is truncated (high-order bits are ignored) or zero-filled (high-order bits).

3.8 Chapter 14: Input-Output

The language-defined packages SEQUENTIAL_IO, DIRECT_IO, TEXT_IO and IO_EXCEPTIONS are supported. In addition, VAX Ada provides packages RELATIVE_IO, INDEXED_IO, SEQUENTIAL_MIXED_IO, DIRECT_MIXED_IO, RELATIVE_MIXED_IO, INDEXED_MIXED_IO and AUX_IO_EXCEPTIONS.

RELATIVE_IO is similar to DIRECT_IO, but adds the ability to detect "empty" records (as in COBOL). INDEXED_IO is also similar and adds the ability to locate records by a key value. Like SEQUENTIAL_IO and DIRECT_IO, both are instantiated with a type to create a package for external files whose elements are all of that same type.

SEQUENTIAL_MIXED_IO, DIRECT_MIXED_IO, RELATIVE_MIXED_IO and INDEXED_MIXED_IO are analogous to their "non-mixed" counterparts, but allow each external file record to contain multiple values of various types. These packages allow convenient interchange of data using files created or used by other VAX languages, such as FORTRAN, COBOL, PL/I and so on.

AUX_IO_EXCEPTIONS defines additional exceptions needed by the relative and indexed input-output packages, namely: LOCK_ERROR, EXISTENCE_ERROR and KEY_ERROR.

All input-output packages are implemented using VAX/VMS Record Management Services (RMS), which provides read sharing for the sequential and direct packages, and read/write sharing with automatic record locking for the relative and indexed packages.

All OPEN and CREATE routines in the input-output packages support FORM parameters using the DIGITAL File Definition Language (FDL). This allows full access to and use of all RMS facilities, including additional file sharing capabilites, allocation control, choice of representation options, and so on.

All input/output packages are integrated with the multitasking kernel so that operations on an internal file are indivisible (atomic) in the presence of concurrent calls from multiple tasks.

3.9 Annex A: Predefined Language Attributes

The values for implementation-defined values are detailed in the appropriate chapters of the VALRM and summarized in Appendix F. The complete list of VAX Ada implementation-defined attributes is:

```
AST_ENTRY
BIT
DICTIONARY_MAP
IMAGE_MAP
NULL_PARAMETER
TYPE_CLASS
VALUE_MAP
```

3.10 Annex C: Predefined Language Environment

Packages MACHINE_CODE and LOW_LEVEL_IO are not provided.

3.11 Appendix F: Implementation Dependent Characteristics

Implementation limits are summarized as follows:

Limit	Description
32	Maximum number of formal parameters in a subprogram or entry declaration that are of an unconstrained record type
120	Maximum identifier length (number of characters)
120	Maximum number of characters in a source line
245	Maximum number of discriminants in a record type
246	Maximum number of formal parameters in an entry or subprogram declaration
255	Maximum number of dimensions in an array type
1023	Maximum number of library units in the transitive closure of the with clauses of a compilation unit
32757	Maximum number of objects declared with PSECT_OBJECT pragmas
65535	Maximum number of enumeration literals in an enumeration type definition
65535	Maximum number of characters in a value of the predefined type STRING
65535	Maximum number of lines in a single source file
65535	Maximum number of frames that an exception can propogate
2**31-1	Maximum number of bits in any object or value

4 USER INTERFACE AND DEBUGGING FACILITIES

VAX Ada source files are compiled in the context of a program library. A program library is created with the ACS:

```
$ ACS CREATE LIBRARY [Mydir.Lib]
```

where $ is the VAX/VMS operating system prompt. This command creates a new VAX/VMS subdirectory named [Mydir.Lib] and initializes it for use as an Ada program library. By default, units from the predefined library, such as TEXT_IO, are automatically entered in the new library. Any number of program libraries are allowed.

The "current" program library is established with

```
$ ACS SET LIBRARY [Mydir.Lib]
```

Switching among program libraries can take place as often as desired.

The compiler is invoked with the ADA command, together with the name of the file to be compiled and any qualifiers needed to obtain desired compilation options:

```
$ ADA/LIST/MACHINE_CODE MYPROG
```

This command compiles the file MYPROG.ADA in the context of the current compilation library, and produces a listing that includes the generated machine code. If there are no errors, then the program library is updated with the results of the compilation, including the generated object module. If there are errors, then the program library is not updated. Debug symbol table (DST) information is included in the object module by default.

If MYPROG contains a main subprogram, say a procedure named MYMAIN, then that program may be linked for execution with:

```
$ ACS LINK MYMAIN
```

Any missing or obsolete units needed by MYMAIN are automatically detected and cause the link operation to be aborted.

The resulting executable image file may then be executed with:

```
$ RUN MYMAIN
```

If debugging is desired, it can be specified as part of the ACS LINK command:

```
$ ACS LINK/DEBUG MYMAIN
```

The RUN MYMAIN command then starts the program under control of DEBUG.

The sections that follow give details pertinent to the commands and facilities illustrated here.

4.1 Compiler Invocation And Listing Management

4.1.1 Invoking The Compiler - The compiler may be invoked with the ADA command or with the ACS COMPILE or ACS RECOMPILE commands. There are no differences in interactive or batch use.

4.1.2 Format And Content Of User Listings - The listings given in Appendix A illustrate most aspects of the compilation listings produced by VAX Ada. A brief summary of those aspects follows.

The header lines of each listing page identify the name of the unit being compiled, the date and time of the compilation, the compiler version, the full name of the source file being compiled and the date and time that the file was created or last modified.

When there is more than one input file, a separate listing file is produced for each.

Messages are embedded in the listing at the point where they apply. A total of each class of messages (errors, warnings and informationals) is given at the end of the listing.

When the /MACHINE_CODE option is specified, the generated machine code is included in the listing. The code appears following each complete subprogram or body. The machine code is correlated with the originating source line as illustrated in the listing for ACKERMANN in Appendix A.

No type map and data map display is available in the listing. Neither is a cross-reference option available, either for a single unit or across units.

The final page of the listing displays various compiler statistics. CPU time, elapsed time, page faults and input-output counts are provided for 25 phases (or subphases) of compilation. Total information is also provided including CPU time, elapsed time, lines/minute compilation rate (based on CPU time) and space (virtual pages) used. There are no fixed-size tables in the compiler.

4.2 Compilation Options

The options available for an Ada compilation are summarized in the following, where "(D)" indicates the default choice. Note that these options can be used with the ADA compilation command as well as with the ACS COMPILE and RECOMPILE commands.

/COPY_SOURCE (D)
/NOCOPY_SOURCE

> Controls whether a copied source file (.ADC) is created in the program library. The RECOMPILE command requires that a copied source file exist for any unit that is to be recompiled.

/DEBUG[=(option[,...])] (D)
/NODEBUG

Controls whether the compiler creates a symbol table for VAX DEBUG. You can request the following:

ALL	Provides both SYMBOLS and TRACEBACK
NONE	Provides neither SYMBOLS nor TRACEBACK; equivalent to /NODEBUG
[NO]SYMBOLS	Controls whether debug symbol records are included in the object file
[NO]TRACEBACK	Controls whether traceback information is included in the object file

By default, both debug symbol records and traceback information are included in the object file.

/DIAGNOSTICS[=file-spec]
/NODIAGNOSTICS (D)

Controls whether a diagnostics file is created for a language sensitive editor.

/ERROR_LIMIT[=n]
/NOERROR_LIMIT

Controls whether execution of the Ada compiler is terminated upon the occurrence of the n-th E-level error within a single compilation unit. The default is /ERROR_LIMIT=30.

/LIBRARY=directory-spec

Specifies the program library that is to be used for the compilation. By default, the library used is the program library last specified in an ACS SET LIBRARY command. The logical name ADA$LIB defines the current program library.

/LIST[=file-spec]
/NOLIST (D)

Controls whether a listing file is produced.

/MACHINE_CODE
/NOMACHINE_CODE (D)

Controls whether the listing produced by the compiler includes the machine language code generated by the compiler.

/MESSAGES[=(category:destination[,...])]

Controls which categories of informational (I-level) and warning (W-level) messages are displayed and where those messages are displayed. You can specify any of the following category keywords:

WARNINGS	Warning (W-level) diagnostics
WEAK_WARNINGS	Informational (I-level) diagnostics that indicate questionable usages
SUPPLEMENTAL	Additional information about a previous error
COMPILATION_NOTES	Notes on how the VAX Ada compiler translated the source program
STATUS	End-of-compilation statistics and updates to the program library

The destination code for a category can be ALL, NONE, or combinations of T (terminal), L (listing), or D (diagnostics file). If a destination code is not given for a category, the default is ALL. If you specify only some of the category keywords with the /MESSAGES qualifier, the default values for other categories are used. The default is:

```
/MESSAGES=(WARN:ALL,WEAK:ALL,SUPP:ALL,STATUS:L,COMP:NONE)
```

/NOTE_SOURCE (D)
/NONOTE_SOURCE

Controls whether the file specification of the current source file is noted in the program library. The COMPILE command uses that information to locate revised source files.

/OPTIMIZE[=option] (D)
/NOOPTIMIZE

Controls whether full optimization is applied in producing the compiled code, and controls whether the primary optimization criterion is the speed of compiled code or the size of compiled code.

You can specify one of the following options:

TIME	Provides full optimization with automatic inline expansion for small subprograms
SPACE	Provides full optimization without automatic inline expansion

/OPTIMIZE=TIME overrides any occurrences of pragma

OPTIMIZE(SPACE) in the source code, without having to edit the source code. Similarly, /OPTIMIZE=SPACE overrides any occurrences of pragma OPTIMIZE(TIME). /NOOPTIMIZE suppresses all optimizations and suppresses all inline expansion, including those specified by pragma INLINE; it also overrides any occurrences of pragma OPTIMIZE in the source code. The default is /OPTIMIZE=TIME.

/SHOW[=option]
/NOSHOW (D)

Controls optional sections of the listing file. You can specify one of the following options:

ALL	Provides all optional sections
NONE	Provides no optional sections
[NO]PORTABILITY	Controls whether a portability summary is included in the listing

/SUPPRESS_ALL
/NOSUPPRESS_ALL (D)

Controls whether run-time error checking is suppressed. /SUPPRESS_ALL is equivalent to including pragma SUPPRESS_ALL in every compilation unit. /NOSUPPRESS_ALL overrides any occurrences of pragma SUPPRESS_ALL in the source code, without having to edit the source code.

/SYNTAX_ONLY
/NOSYNTAX_ONLY (D)

Controls whether the source file is to be syntax-checked only, rather than being fully compiled.

No options are provided to control the width of a listing. The listing file assumes an available width of 132 characters. Terminal messages assume an available width of 80 characters.

The compiler /OPTIMIZE=NONE qualifier is available to suppress code optimization. While VAX DEBUG is designed to be used with optimized code, there may occasionally be situations where ease of debugging is aided by use of this option.

4.3 Other Features

The source file for each unit is fully identified in the object program; this information is used by both ACS and VAX DEBUG.

4.4 Errors And Warnings

No object code is produced for units containing errors. Object code is produced if the messages are warnings or informational.

4.4.1 Compiler Messages - There are four levels of messages: Fatal Errors, Errors (both classes suppress object file generation), Warnings and Informationals. Informational messages are further subdivided into Weak Warnings, Supplemental, Status and Compilation Notes.

Certain errors can produce multiple lines of informational text to aid in program correction. For example, the error message "Inconsistency detected during overload resolution" is followed by supplemental messages that provide information about how the overloading resolution was attempted. Terminal output of such supplemental messages is limited to six lines, provided a listing is also being produced (which will contain the full message); otherwise the full message is displayed. Error messages frequently provide information on what was "found", what was "expected" and give suggested corrections.

Warning messages and each class of informational messages can be selectively enabled or disabled to each of the listing, terminal and/or diagnostics file.

All messages are in English. Messages take the general form of VAX/VMS error messages with further informational text. The compiler design generally provides for the ability to provide messages in other natural languages, but this is not available in Version 1.

Terminal messages are available in the batch log file when the compiler is used from batch. Interactively, terminal error messages can also be directed to a file for later reference by redirecting the logical output file SYS$ERROR. More usefully, the /DIAGNOSTICS option can be used to create a specially formatted file that allows the optional language sensitive editor to simultaneously display errors and allow editing of the source file using "split screen" techniques.

The compiler attempts to pinpoint errors as accurately in the source code as possible. A dot and numeral (.....1) technique is used to show the correspondence of error messages to points in the source code.

4.4.2 Error Messages - Error messages provide a great deal of supplemental information; for example, how the compiler attempted to resolve overloading, what syntactic token the compiler expected at a particular point and what it actually found, etc.

The compiler uses an error-correcting parser. Attempts are made to diagnose as many errors as possible while minimizing cascading errors. There is no "restart at a position" ability in the compiler.

4.4.3 Warning Messages - An informational diagnostic is produced for

- an exception that is sure to be raised
- unreachable source code
- an object that is declared but not used (in which case no storage is allocated)

These diagnostics are produced only if the /MESSAGES=COMPILATION_NOTES option is given with the compilation command (not by default).

No diagnosis is provided for:

- use of an unusually expensive construct
- a real expression whose accuracy is inherently low
- erroneous usages
- use of an unassigned (uninitialized) variable
- an endless loop

A warning message is produced if a pragma is ignored and compilation proceeds.

No warning is given or needed for code that is moved or combined with other code; the debugger will correctly correlate source lines with generated code at run-time, even if code has been shifted.

4.5 Other Software Supplied

No facility for cross-referencing of identifiers and names is provided, nor is a facility provided to display the fully qualified name, except as part of error messages for overloading resolution errors.

No pretty printing or reformatting facility is provided. However, use of the optional language sensitive editor makes it easy to generate source in a consistent format.

4.6 Compilation (Program) Library Management

Management and utility functions for program libraries are provided by the ACS. The ACS program library management commands are summarized as follows:

CHECK	Forms the closure of one or more compiled units and checks the completeness and currency of the units in the closure.*
COPY FOREIGN	Copies a foreign (non-Ada) object file into a program library as a library body.
COPY UNIT	Copies a compiled unit from one program library to another.
CREATE LIBRARY	Creates a VAX Ada program library.
CREATE SUBLIBRARY	Creates a VAX Ada program sublibrary, to allow you to isolate the development of selected units.
DELETE LIBRARY	Deletes a program library and its contents.
DELETE SUBLIBRARY	Deletes a program sublibrary and its contents.
DELETE UNIT	Deletes one or more compiled units from a program library.
ENTER FOREIGN	Enters a reference (pointer) from a given program library to an external file as a foreign (non-Ada) library body.
ENTER UNIT	Enters a reference (pointer) from a given program library to a unit that has been compiled into another program library. The "entered" units can be used in the given program library as if they were actually in it.
EXPORT	Creates an object file that contains the object code for one or more units.
MERGE	Merges, into the parent library, new versions of one or more units from the sublibrary where they were modified. MERGE replaces the older obsolete version(s) in the parent library.
REENTER	Enters current references to units that were compiled after they were last entered with ENTER UNIT.

---

* In simple terms, "closure" is the complete set of units that a given unit depends on, plus any units needed for its execution.

SET LIBRARY	Defines a program library to be the current program library -- that is, the target program library for compiler output and ACS commands in general.
SET PRAGMA	Redefines specified values of the library characteristics SYSTEM_NAME, LONG_FLOAT, MEMORY_SIZE.
SHOW LIBRARY	Displays the name and characteristics of one or more program libraries.
SHOW PROGRAM	Displays key information about one or more units in a program library.
SHOW UNIT	Displays information, such as name and date and time of last compilation, about one or more units in a program library.

ACS and DCL commands for compiling, linking, and execution are summarized as follows:

DCL Commands

($) ADA	Invokes the VAX Ada compiler and compiles the specified Ada source file(s).
($) RUN	Executes the specified executable image file.

ACS Commands

COMPILE	Forms the closure of one or more compiled units; checks the completeness and currency of the units in the closure; identifies units that have revised source files; compiles units that have revised source files and units that are obsolete or will be made obsolete.
LINK	Forms the closure of an entire compiled program, checks the completeness and currency of the units in the closure, and links the set of units.
RECOMPILE	Forms the closure of one or more compiled units, checks the completeness and currency of the units in the closure, and recompiles any obsolete units in the appropriate order to make them current.
SET SOURCE	Defines a source file search list for the COMPILE command.

SHOW SOURCE	Displays the source file search list used by the COMPILE command.

Additional miscellaneous ACS commands are provided as follows:

ATTACH	Switches control of the terminal from the current process running ACS to another process in the job.
EXIT	Exits from ACS. CTRL/Z can also be used.
HELP	Invokes the ACS HELP facility to obtain information about ACS commands.
SPAWN	Creates a subprocess of the current process and suspends execution of the current process.

The ACS CHECK command identifies all units that a particular unit depends on that are either not in the library or are not current. The check can apply to a single unit, an explicit list of units (considered together) or all units of a program library.

The ACS RECOMPILE command provides the ability to recompile a unit and everything that it depends on that has become obsolete. It does not provide the ability to recompile automatically everything that depends on a particular unit; however, a similar effect can be achieved by using the ACS RECOMPILE command with the main subprogram and requesting a command file (/COMMAND option) for recompiling the main unit and any units it depends on and then editing out any unwanted compilations.

The ACS SHOW PROGRAM command displays structural dependences for a particular unit or set of units.

There is no facility for management control of low-level features, but the compiler can generate a cross-reference of potentially nonportable features (/SHOW=PORTABILITY option).

A set of flags indicating the presence of each kind of potentially nonportable feature is, however, always maintained for every unit, whether or not such information was requested in a listing. This information is optionally displayed by the ACS SHOW PROGRAM command.

4.7 Debugging Facilities

The VAX Symbolic Debugger (DEBUG) provides extensive facilities to aid the debugging of programs in VAX/VMS multilanguage programs.

4.7.1 General Facilities - The general facilities that apply to all supported languages allow users to:

- Stop execution (set breakpoints) at selected locations, specified either by routine name or by source module and line number, or when exceptions occur.
- Single step in increments of single hardware instructions or source lines. Stepping can selectively pass into called routines or treat the execution of a routine call as a single instruction.
- Watch the value of specified variables and interrupt execution when the value of a variable is changed.
- Trace execution (similar to breakpoints) at selected locations, reporting the location reached and continuing.
- Examine the contents of variables, displaying their contents according to their type.
- Evaluate expressions, including those that involve program variables.
- Deposit new values in variables.
- Display the current sequence of routine calls, giving the module and line numbers of each call in the source program.
- Display the source corresponding to any part of the program.
- Perform debug command procedures at breakpoints, tracepoints and watchpoints.
- Call procedures that are part of the program (functions cannot be called as part of expression evaluation, but can be called as a procedure and the result displayed).

DEBUG automatically establishes the language of discourse as the language in which the main program is written. Thereafter the language can be changed as appropriate to the parts of the program where debugging is in progress. The language setting adjusts the DEBUG syntax and interpretation of expressions as appropriate to that language.

DEBUG support for Ada language expressions differs in various respects from "pure" Ada; it does, however, provide extensive capabilities for interpreting Ada expressions in a useful manner.

Very briefly, DEBUG supports the following aspects of Ada expressions:

- Syntax of identifiers, except that user operator names must be given as, for example, %NAME '+'
- Reserved words
- Numeric, character and string literals
- Indexed component, slice, selected component and attribute notations
- Evaluation of (only) the following attributes: CONSTRAINED, FIRST and FIRST(n), LAST and LAST(n), LENGTH and LENGTH(n), POS, PRED, SIZE, SUCC and VAL
- Evaluation of operators, except that relational operators are supported only for scalar and string types, and concatenation operators are supported only for string and character types
- Qualified expressions, but only for the disambiguation of enumeration literals
- Use clauses

The debugger does not support:

- Overloading resolution (ambiguities are reported and special means are available to indicate how to interprete ambiguous names)
- Aggregates
- Operations on entire array or records
- Short circuit control forms: 'and then', 'or else'
- Membership tests: 'in', 'not in'
- Type conversions
- Allocators

4.7.2 Tasking Facilities - Debugging of programs involving Ada tasks requires additional facilities beyond those applicable to most languages. In addition to the general facilities described above, the facilities available in VAX Ada for task debugging allow users to:

- Display the state of given tasks or all tasks in a program. The possible states are:
  - created
  - running
  - ready
  - suspended
  - terminated
- Display more detailed substates of the main state:
  - abnormal

  - waiting at an accept statement
  - activating (elaborating its declarative part)
  - activating tasks (waiting for dependent tasks to finish activating)
  - completed: either abnormally, due to an exception or normally
  - delay
  - dependents: waiting for dependent tasks to terminate, either normally or due to an exception
  - entry call
  - I/O or AST
  - not yet activated
  - select: with or without delay or terminate alternatives
  - terminated: either abnormally, due to an exception or normally
  - timed entry call

- Place a task "on hold" so that it is not scheduled for execution

- Establish a given task to be the active task

- Change the priority of a task

- Display tasking run-time statistics: number of entry calls, tasks activated, accept and select operations, and so on

- Stop execution (set breakpoints) for various tasking events:

  - an exception is about to be handled: in any handler or just an 'others' choice
  - an exception is about to propagate from a rendezvous
  - an exception is waiting for dependents to terminate
  - a task is about to terminate: due to an exception, abort or normally
  - a task is about to execute: for the first time or any time
  - a task is preempted by a higher priority task
  - a task becomes activated
  - a task becomes suspended

4.7.3 VAX Ada Specific DEBUG Commands - A number of DEBUG commands have been implemented specifically to support the debugging of VAX Ada programs. These supplement the capabilities previously available. Some of the more important commands are summarized as follows.

CANCEL MODULE /[NO]RELATED

Enables automatic module cancelling and thereby cancels all those modules that are related to this one. /NORELATED specifies that only the module or modules specified in this command are to be cancelled.

SET BREAK /EVENT = event_name

Specifies that execution should be suspended whenever certain Ada tasking events occur. The event_names that can be specified are:

HANDLED	Triggers when an exception is about to be handled in some Ada exception handler, including an 'others' clause.
HANDLED_OTHERS	Triggers when an exception is about to be handled in an 'others' Ada. exception handler
RENDEZVOUS_EXCEPTION	Triggers when an exception is about to propogate out of a rendezvous.
DEPENDENTS_EXCEPTION	Triggers when an exception causes a task to wait for dependent tasks. in some frame
TERMINATED	Triggers when a task is terminating, whether normally, by abort or by an exception.
EXCEPTION_TERMINATED	Triggers when a task is terminating due to an exception.
ABORT_TERMINATED	Triggers when a task is terminating due to an abort.
RUN	Triggers when a task is about to run.
PREEMPTED	Triggers when a task is being preempted from the RUN state.
ACTIVATED	Triggers when a task is going to run for the first time.
SUSPENDED	Triggers when a task is about to be suspended.
READY	Triggers when a task becomes ready to run.

SET MODE [NO]DYNAMIC

DYNAMIC enables dynamic module setting and NODYNAMIC disables it. Dynamic module setting means that whenever you are at the DBG prompt, if your PC is in a module that is not set, then DEBUG will set the module for you. For example, if you step into a function in a module that is not set, DEBUG will set the module and issue an informational that it has done so. In the case of language Ada, additional related modules may be set as well.

SET MODULE /[NO]RELATED

/RELATED enables automatic module setting and thereby sets all those modules that are related to this one. This qualifier has no effect for those languages that do not require automatic module setting. /NORELATED disables setting of related modules. The default is /RELATED.

SET TASK taskname

Modifies certain characteristics of tasks in your program. Using this command, you can change a task's priority, put a task on hold, switch to a different active task, or make a task visible to the debugger.

A taskname parameter may be a pathname which designates a task declared in your program, for example, MY_MAIN\PROG\MONITOR. A taskname parameter may also be a task id. A task id is a unique number that is associated with a task at the time the task is created. A task id is of the form %TASK n. You can see what task ids were assigned to currently existing tasks by typing SHOW TASK/ALL.

If you omit all taskname parameters, you can specify the /ALL qualifier to select all tasks. If you omit both taskname parameters and the /ALL qualifier, the "visible task" is assumed. The visible task is typically the task that caused execution to enter the debugger (see also SET TASK/VISIBLE and SET TASK/ACTIVE).

The task properties that can be set are:

/ACTIVE

Causes the task specified by the taskname parameter to become the "active task". The named task must be in either the READY or the RUN state. The active task is that task which will execute instructions when the debugger is left (as when you type GO, STEP or CALL). It is marked with a '*' in the SHOW TASK display.

The SET TASK/ACTIVE command allows you to force a "task switch" to the named task without leaving the debugger.

In addition to making the named task become the active task, this command also makes it become the visible task, which is the task assumed by debugging commands when you do not specify a task.

/ALL

Specifies that all the tasks in your program are to be modified as specified by the other qualifiers given. When /ALL is specified, taskname parameters are not permitted.

/HOLD

Specifies that the selected tasks are to be placed on hold. Tasks which are on hold will not be allowed to run until the hold has been removed.

/NOHOLD

Specifies that the selected tasks are to be removed from hold status.

/PRIORITY = n

Specifies that the selected tasks are to have their priority changed to n.

/RESTORE

Specifies that the selected tasks are to have their natural priorities restored. This allows you to undo the effect of SET TASK/PRIORITY, without the need to remember the task's natural priority.

/VISIBLE

Causes the task specified by the taskname parameter to become the "visible task". The visible task is the task assumed by debugging commands when you do not (or cannot) specify a task. For example, the debugger command EXAMINE %R0 displays register 0 of the visible task.

The task whose execution entered the debugger is called the active task. Each time the debugger is entered, the visible task and the active task are one and the same. Using this qualifier, you can make the visible task be a task other than the active task.

SHOW MODULE /[NO]RELATED

This qualifier tells DEBUG to output information about those modules that are related to each module that is normally displayed. It lists the relationship of each of these subordinate modules to the main module along with the normal information you get with each module. This qualifier outputs rather uninteresting results for those languages that do not have automatic module setting. The default is /NORELATED.

SHOW TASK taskname

Displays information about tasks in your program. Using this command, you can obtain a list of tasks in existence and their current state. The form SHOW TASK/STATISTICS provides summary information on all tasks.

The options available with this command are:

/ALL

Specifies that all the tasks in your program are to be considered for display. When /ALL is specified, taskname parameters are not permitted.

/CALLS [= n]

For each task selected by other qualifiers, causes information to be displayed for the last n procedure calls.

/FULL

Causes detailed information to be displayed.

/HOLD

Specifies that only those tasks which have been placed on hold are to be considered for display (see SET TASK/HOLD). If you omit the taskname parameters, this qualifier selects all tasks that are on hold (that is, an /ALL qualifier is implied).

/NOHOLD

Specifies that only those tasks which have not been placed on hold are to be considered for display (see SET TASK/HOLD). If you omit the taskname parameters, this qualifier selects all tasks that are not on hold (that is, a /ALL qualifier is implied).

/PRIORITY = n

Specifies that only those tasks having the specified priority are to be considered for display. A list of priorities given in parentheses is also permitted, for example, SHOW TASK/PRIORITY = (3, 4). If you omit the taskname parameters, this qualifier selects all tasks that have the given priorities (that is, a /ALL qualifier is implied).

/STATE = s

Specifies that only those tasks in the specified state are to be considered for display. A list of states given in parentheses is also permitted, for example, SHOW TASK/STATE = (READY, RUN).

The debugger recognizes the following task states:

READY, RUNNING, SUSPENDED and TERMINATED

/STATISTICS

Displays summary statistics about the execution of all tasks in your program. If /STATISTICS is specified, the only other qualifier permitted is /FULL.

5 PERFORMANCE AND CAPACITY

5.1 Host Performance And Capacity

There is no simple way to indicate the VAX Ada compilation rate because there are no accepted standards for this measure. Package specifications, which typically involve little or no code generation, tend to compile very quickly -- rates exceeding 5000 lines per CPU-minute on a VAX-11/780 have been observed. On the other hand, extensive use of generics and/or inline expansion can greatly slow the compilation rate -- rates below 100 lines per CPU-minute have been observed (in contrived programs, the rate can be made arbitrarily low).

In programs that are representative of normal medium- to large-scale project development, measurements indicate that a compilation rate of 800 to 1000 lines per CPU-minute on a VAX-11/780 is representative for the field test version of VAX Ada. The compiler tends to perform better on larger compilation units than on small ones. One field test site is reporting compilation rates in excess of 1000 lines per CPU-minute for a large application of over 200,000 source lines.

There is a relatively small and fixed overhead associated with locating and opening the program library file for each unit named in a with clause (independent of the size of that library unit); moreover, for each library unit, it is also necessary to transitively locate and open the program library file for each unit named in its with clauses. The mere occurrence of a unit in a with clause, of itself, has only this small initial overhead. Thereafter, additional incremental cost is incurred whenever needed information is first loaded from that unit.

For subunits, there is an overhead for locating and loading information about each ancestor unit up to and including the parent library unit, after which compilation proceeds at a rate comparable to any other unit.

The VAX Ada compiler can run on a system with 1 megabyte of physical memory, but more memory is preferable: at least 4 megabytes for a VAX-11/780, at least 1-3/4 megabytes for a MicroVAX. A working set of 3/4 megabytes (1500 pages) or more is recommended to run the compiler.

If the host system does not have sufficient physical memory, the compiler will page excessively, and the compilation speed will be slowed considerably. But all VAX processors, down to and including the MicroVAX, can be configured with sufficient memory so that there is no practical limit on the size of program unit that can be compiled.

There are no specific restrictions on the numbers of items of various kinds in a unit (identifiers, strings, nesting and so on). All implementation limits are given in Section 3.11.

When a unit is recompiled, no use is made of the previous compilation to increase compiling speed. Similarly, if a package specification is changed by adding a declaration, the compilation of that package makes the units that use the package obsolete.

However, the "implicit recompilation" of package SYSTEM that occurs for one of the pragmas SYSTEM_NAME, MEMORY_SIZE or STORAGE_SIZE, and similarly, of package STANDARD for pragma LONG_FLOAT, is handled in a special way. A unit becomes obsolete as a result of compiling these pragmas only if it actually depends on the values (type representation in the case of LONG_FLOAT) changed by these pragmas. This applies also to the ACS SET PRAGMA command that provides the same capability.

5.2 Target Code Performance

Generally, VAX Ada compiled code is comparable in code quality and performance with other DIGITAL-supplied language products. A wide variety of standard optimization techniques are used throughout the compiler. The code generator used in VAX Ada is also used in DIGITAL's PL/I and C language products.

Most redundant constraint checks are eliminated. For example, in the following:

```
I: INTEGER range -2 .. 2;
J: INTEGER range 0 .. 10;

type AT  is access T;
V: AT;

I := 22 mod 3;     -- (1) no checks needed
I := J;            -- (2) check on top limit only
V := new T(...);
if V.L = ... then -- (3) no null access check
                   -- (4) current variant is correct
```

Contraint checks are eliminated at (1), (2) and (4). (Note: the field test compiler generally does not eliminate null access checks.)

Subprograms which are declared in a package but not used are loaded with the package. Note, however, that in a virtual memory system such as VAX/VMS, virtual address space that is never accessed has relatively little cost.

Static (sub)expressions are always evaluated by the compiler, even when when the LRM does not require it. Moreover, additional compile-time constant expressions are also evaluated by the compiler, even though they are not "static" expressions as defined in the LRM. Examples of such expressions are concatenation and some of the attributes, such as IMAGE.

Dead code is eliminated from if and case statements.

In statements such as

```
A(I) := A(I) + 1;
```

the address of A(I) is evaluated just once. Also, a special increment instruction is generated; however, the result of the addition may be assigned to a temporary if a constraint check must be performed before updating A(I).

With matrix computations such as

```
for I in 1 .. N loop
    for J in 1 .. M loop
        A(I,J) ...
    end loop;
end loop;
```

the address calculation for A(I,J) tends to be calculated each time using special VAX hardware addressing for this purpose (context indexing). Depending on various details, strength reduction may alternately be used.

When array index expressions are static, index bounds checking is performed during compilation.

Pragma INLINE is supported in most cases. Routines that contain nested routine declarations, exception declarations, generic declarations and/or instantiations, dependent tasks, access type declarations, or directly recursive calls are considered not to be inlinable; that is, they will not be expanded inline at the place of any calls to such routines. The result of instantiating a generic subprogram is also inlinable under the same conditions.

The compiler will expand local routines inline without pragma INLINE being specified where heuristics indicate that the size of the expanded call will be approximately the same as the call it replaces (or only a little larger). This can be suppressed with pragma OPTIMIZE(SPACE) or the equivalent /OPTIMIZE(SPACE) compiler option.

A subprogram that is called only once is not automatically expanded inline.

In a rendezvous, the rendezvous code is executed by the owning task, never by the calling task. (Experience using the so-called Habermann-Nassi optimization indicates that equivalent benefits can be achieved with less complexity.)

Tasks that are acting as monitors are not recognized as such, and do not result in special code generation.

Tasks are normally rescheduled only when the currently executing task is unable to proceed, or when a higher priority task becomes eligible for execution. Note that predefined input-output packages are implemented using asynchronous techniques so that other tasks can execute while one is waiting for input-output to complete.

Pragma TIME_SLICE may be used in a main subprogram to obtain timesharing-like scheduling on a round-robin basis for tasks with the same priority.

The VAX Ada run-time library is shareable, meaning that a single copy of it is used by all executing Ada programs on a particular system (it is also used by the Ada compiler). The shareable image is approximately 10,000 bytes. Because it is shareable, all of it is always available.

6 COMPILER AND RUN-TIME INTERFACING

6.1 Compiler Issues

6.1.1 Design Criteria - The design of the compiler is oriented toward medium- to large-scale software development. No particular application domain is envisioned consistent with VAX/VMS host and target applicability.

6.1.2 Compiler Phase And Pass Structure - The compiler can be considered to consist of the following phases:

- o Parser

  The parser is LALR(1) and uses an inhouse parser generator. (The parser generator is written in PASCAL and produces BLISS source control tables for use by the parser driver, which is written in BLISS.)

- o Static Semantics

  The static semantics phase performs declaration processing, name visibility and scope management, overloading resolution, generic and inline processing and all legality checking. (Generic and inline expansion does not actually occur during static semantics, however. Expansion is performed "on demand" as needed by backend processing.)

- o Dynamic Semantics

  The dynamic semantics phase performs size and shape determination for types and objects, inserts needed constraint and other checks (when not statically known to be satisfied), and performs literal and aggregate pooling. Subprogram, package and task bodies are converted to a linear n-tuple representation (innermost first) that is passed to the backend.

- o Backend

  The backend is a fairly classical code generator that performs flow analysis, code generation, register allocation, peephole optimization, jump/branch resolution and object module output. The body portion of the abstract syntax tree is discarded after code has been generated. (This code generator is also used in other DIGITAL language products, notably PL/I and C.)

- Program Library Output

  At the completion of code generation, the remaining abstract syntax tree consists of just those parts of the program that are potentially needed in the compilation of other units. This is written to the program library.

6.1.3 Compiler Module Structure - A number of the compiler components are common to or used with other DIGITAL-supplied language products. However, no provision is made for customers to take advantage of these components as independent entities.

6.1.4 Intermediate Program Representations - The intermediate representations used by the compiler are an abstract syntax tree and a code generator intermediate language. The abstract syntax tree representation is similar to DIANA in many respects because it evolved from the same formal semantics abstract syntax tree representation that was one of the major influences in the design of DIANA. However, there is no intention to establish or maintain any kind of compatibility with DIANA.

The compiler is written in several languages, namely: BLISS, PL/I, VAX MACRO and Ada. The only part of the compiler that is written in Ada is the universal arithmetic package. No attempt was made to avoid any particular language constructs.

BLISS, PL/I and VAX MACRO are all fully supported and sold by DIGITAL.

6.1.5 Final Program Representation - The format of the final object program is a standard VAX/VMS object file.

For an ACS LINK command, the ACS identifies the units that are needed and checks that they are all current. It then constructs an initialization object module that causes the elaboration of library units in an appropriate order. Finally, it creates a linker command (options) file and invokes the VAX linker. The link can be performed by a subprocess (with or without waiting for completion), or it can be optionally be submitted as a batch job.

6.1.6 Compiler Interfaces To Other Tools - The compiler generates a DEBUG symbol table (DST) as part of the object module for use with VAX DEBUG to allow full symbolic debugging. The same symbol table information is used by the Performance and Coverage Analyzer (PCA) for symbolic dynamic execution analysis.

The compiler also generates a diagnostic files for use with the Language Sensitive Editor (LSE).

The interfaces and formats used are generally not documented or supported for customer use.

6.1.7 Compiler Construction Tools - All compiler construction tools are proprietary to DIGITAL.

6.1.8 Target-Dependent Information - The compiler is not designed with the intent of retargeting by customers.

6.1.9 Installation - The VAX Ada product is supplied on typical VAX/VMS large layered product distribution media, including magtape and certain disks.

The VAX Ada product is installed using the normal automated VMS software installation procedures (VMSINSTAL). The process is easily performed by customers, and typically takes around 15 minutes on a VAX-11/780.

The "VAX Ada Installation Guide and Release Notes" provides all procedures and information necessary for customer installation.

6.2 Run-Time System Issues

6.2.1 Storage Management - A subheap is allocated for each collection. Fixed-size blocks with bit map allocation techniques are used for access types whose denoted type is fixed-size, and variable length blocks with first fit allocation techniques are used for access types whose denoted type is not fixed-sized.

Collections are deallocated when leaving the scope of the parent access type. UNCHECKED_DEALLOCATION is supported. Automatic garbage collection within a subheap is not provided.

6.2.2 Subprogam Call Parameter Handling - Scalar parameters are passed by reference with the copy-in/copy-back semantics of Ada implemented by prolog/epilog code in the called subprogram. Arrays and records are passed by reference except in circumstances where a byte-aligned copy is required (for example, for bit-aligned components of arrays and records). A copy may also be required on the calling side to provide the constraint check in the copy-back of 'in out' or 'out' parameters.

The allocation rules of VAX Ada are such that an array or record component can only be bit-aligned (not byte-aligned) when its size is static and less than 32 bits; thus, copying of array or record values for the purposes of parameter passing is never an expensive operation.

Function values with a size of 32 bits or less are returned in general register R0, and those with a size of between 33 and 64 bits are returned in registers R0 and R1. For other cases involving fixed-size function results, the caller allocates storage for the function value and passes the address or a descriptor to the function as an additional parameter. If the caller cannot allocate the storage, it is heap allocated by the function and freed by the caller.

Complete details are given in the "VAX Ada Programmer's Run-Time Reference Manual".

6.2.3 Data Representation - Signed and unsigned bytes (8 bits), words (16 bits), and longwords (32 bits) are used for integer and enumeration types. These can optionally be packed and can be bit-aligned within records and arrays. F (32 bits), D (64 bits), G (64 bits) and H_floating (128 bits) representations are used for real types. Signed longwords (32 bits) with an implicit binary scale factor are used for fixed point types.

Subtype descriptors do not exist as such, although temporaries may be used to hold the values of nonstatic constraints.

Storage allocation for array types is simply the number of components times the allocation of each component. Components can be packed and bit-aligned in some cases. A separate array (subtype) descriptor is allocated for parameter passing when necessary.

Storage allocation for records is the sum of the individual component allocations, which are byte-aligned by default. Components can be packed and bit-aligned in some cases.

For a constrained object of a record type with discriminants, the allocation is that required for the given variant (only). For an unconstrained object of a type with discriminants with defaults, the maximum size of each variant is used. Each object is contiguous, and heap allocation is never used (even for dynamically sized components).

Storage allocation for an access type is a (32-bit) longword pointer which contains the address of the designated object; in the case where the designated type is an unconstrained array type, the pointer contains the address of a descriptor which, in turn, includes a pointer to the array object together with index bounds.

6.2.4 Implementation Of Tasking - Stack storage for the main program is allocated as for a normal VAX/VMS program ("P1 space" in VAX terminology), and is automatically extendable. Storage for tasks is allocated in two parts: a task control block and a task stack. Both parts are heap allocated from the VAX/VMS process virtual address space ("P0 space" in VAX terminology).

When a task is created, a task control block and a user-specifiable fixed-size stack are allocated. The stack is deallocated when the task terminates, and the task control block is deallocated when execution leaves the scope of its master.

An Ada run-time kernel is used to implement multitasking within a single VAX/VMS process. The method of passing data parameters in a rendezvous is the same as for subprogram parameters. No copying of parameters or argument lists is performed by the run-time kernal.

The accuracy of the real-time clock and the delay statement is approximately 1/100th of a second.

External interrupts cannot be associated with task entries. However, VAX/VMS asynchronous system traps (ASTs) can be associated with entry calls.

An AST is an interrupt-like call initiated by the VAX/VMS operating system in response to certain conditions or events detected by the operating system (or hardware), such as completion of an input-output request. The identification of the "routine" to handle the AST is completely dynamic and is specified as one of the parameters to each system service call that provides this capability. Thus, the static association of an interrupt with an entry is not applicable to VAX/VMS.

Pragma AST_ENTRY must be used to identify an entry that will potentially be used to handle an AST. (This does not preclude its use in a normal entry call.) Attribute AST_ENTRY is then used to identify the "routine" to handle the AST as part of a system service call. Each evaluation of the AST_ENTRY attribute dynamically constructs a special code segment to transform the AST into an entry call, and returns the address of that segment to be passed to VAX/VMS. When the AST occurs, control passes to the special code segment in an interrupt-like fashion, which then queues a call to the intended entry and dismisses the AST. Handling of the AST by the task then occurs as for an ordinary entry call. In particular, the handling does not occur at AST-level (interrupt level).

6.2.5 Exception Handling System - Exception handling in Ada is implemented using the condition handling mechanism provided by VAX/VMS. Details are given in the "VAX Ada Progammer's Run-Time Reference Manual".

Provision of an Ada exception handler typically costs one instruction per call if no exceptions occur. When exceptions occur, there are increasing amounts of overhead as options in the following list are exercised: provision of an exception part, the number of handlers in that part, and use of "raise;".

Exceptions that are propagated out of a rendezvous are somewhat costlier than normal exceptions. A "copy" of the exception and its related parameters is created (which may entail some loss of information regarding non-Ada exceptions propagated from the VAX/VMS environment). This "copy" is propagated to the caller.

Distinct host exceptions are defined for each predefined exception in the Ada langauge and VAX Ada implementation. A single host exception serves for all user-defined exceptions with a parameter that identifies the text of the user exception name. Certain distinct host exceptions (for example, integer overflow) are automatically mapped to the predefined Ada exceptions (for example, NUMERIC_ERROR).

6.2.6 Input-Output Interfaces - TEXT_IO is fully supported. Many other VAX/VMS facilities are also available.

Packages are provided for a wide range of input-output options, including relative file input-output, indexed file input-output, and so on (see Section 3.8).

Execution of input-output for access types (including types that have subcomponents of an access type) is erroneous.

6.2.7 Generics - Generic instantiation is implemented using a macro-like expansion mechanism. Sharing of code among multiple instantiations is not provided in Version 1.

(Note: the design of VAX Ada does, however, anticipate support for generic code sharing. Many of the necessary mechanisms are already implemented and latent in the current version. Support for generic code sharing is a high priority after Version 1.)

In the absence of code sharing, the requirements for an instantiation are generally those required by the semantics of Ada, and costs over and above an equivalent nongeneric subprogram or package are quite minimal. For an 'in' formal object, a constant is created for each instantiation and initialized from the appropriate expression. For an 'in out' formal object, temporaries may be needed as for the equivalent renaming.

An instantiation is processed in a macro-like fashion with the consequence that the code for the resulting subprogram or package is compiled and optimized in the presence of full compile-time information on the actual parameters.

6.2.8 Documentation - The documentation provided with VAX Ada includes the following:

- "VAX Ada Language Reference Manual"

  Includes the full text of the Ada LRM together with VAX Ada specific supplements inserted where most appropriate. (Insertions are marked by change bars in the field test version of the manual.)

- "Developing Ada Programs on VAX/VMS"

  Describes use of the VAX Ada compiler, ACS and DEBUG to compile, link, run and debug VAX Ada programs.

- "VAX Ada Programmer's Run-Time Reference Manual"

  Gives system-related information, such as VAX Ada storage allocation and object representations, and explains how to use operating system components external to the language (for example, VAX/VMS system services), how to use operating system-related Ada features (such as multitasking and input-output), and how to use code written in other VAX languages in an Ada program.

- "The VAX Ada Language Summary"

  Summarizes the Ada language syntax as well as ACS and debugger commands.

  Note: not available during field test.

- "VAX Ada Installation Guide and Release Notes"

  Gives step-by-step instructions for installing the VAX Ada product, including information and recommendations on resource requirements. The release notes give errata, last-minute corrections, changes from earlier versions and other similar information.

Extensive online HELP is also provided.

7 RETARGETING AND REHOSTING

The VAX Ada product is not designed nor is it intended for retargeting

or rehosting by customers. The questions raised under this section by the "Ada-Europe Guidelines" are, therefore, not applicable.

8 CONTRACTUAL MATTERS

The following is a very brief summary of DIGITAL's standard terms and conditions for software, including some particulars for VAX Ada. For full details, contact a DIGITAL sales representative.

Several types of license are available for use of VAX Ada. For the VAX 8600, VAX-11/785, VAX-11/782, VAX-11/780 and VAX-11/750 processors there are single-use, cluster-use, and educational licenses. For the VAX-11/730 and VAX-11/725, there are single-use and educational licenses. For the MicroVAX I, there is a single-use license. Media, documentation and support services are sold separately from the license.

A separate license is required for each CPU on which the compiler runs. The VAX Ada license permits the run-time library to be moved to other systems. At its initial release, the Ada run-time library will be shipped with the compiler. However, the run-time library will be bundled with the base operating system in a future release of the operating system so that an application developed in Ada can be moved from one CPU to another. In general, there are no restrictions on use and/or distribution of software produced by the compiler.

In the United States, a full single-use license is $24,900. MicroVAX I and educational licenses are $4980 (educational licences may be purchased only by two- or four-year degree granting institutions). Cluster licenses are $14,940 per CPU (a full single-use license must be purchased for one member of a cluster). These prices are for the license only. Software media and documentation are an additional $2040 to $3600, depending on the media. Separate prices apply in other countries.

Standard support services are available which include periodic updates containing bug fixes. Supported customers can submit problem reports. Telephone support is also available. Support services are generally sold on an annual, renewable basis.

A variety of support services are available, as for other DIGITAL language products, including telephone support, maintenance updates, software problem report service, onsite services, and customized consulting services. Updates, when not covered by a support arrangement, are generally available to licensed customers for a charge to cover the cost of media and handling.

There are no plans to release detailed internal information on the compiler, ACS or Ada run-time library.

9 VALIDATION

VAX Ada was successfully validated as of September 17, 1984 using Version 1.4 of the Ada Compiler Validation Capability (ACVC).

The validation was conducted by the Federal Compiler Testing Center under the supervision of the Ada Validation Office (AVO). All applicable ACVC tests were run on the VAX-11/785. All applicable executable tests were run on the VAX-11/780, VAX-11/750, VAX-11/730 and MicroVAX I.

APPENDIX A

SAMPLE LISTINGS

The following pages show two listings that illustrate various aspects of the listings available from the VAX Ada compiler.

The first, from the compilation of a function that computes Ackermann's function, shows the format used for generated code.

The second, from the compilation of a procedure containing a variety of errors, shows the kinds of error reporting provided.

```
Ackermann                                                    15-Nov-1984 08:56:57    VAX Ada T1.0-2
01                                                           31-Oct-1984 10:26:34    ADAD$:[VATS.AD.PROJ]ACKERMANN.ADA;1       (1)

     1      -- Ackermann function as extracted from the LA Ada Fair '84 test
     2      -- Time_Ackermann to illustrate the VAX Ada compilation listing format
     3      --
     4      function Ackermann (M, N: Natural)  return Natural is
     5      begin
     6        if M = 0      then
     7          return N + 1;
     8        elsif N = 0  then
     9          return Ackermann (M - 1, 1);
    10        else
    11          return Ackermann (M - 1, Ackermann (M, N -1 ));
    12        end if;
    13      end Ackermann;

Line 00004
                                          0000  ACKERMANN:
                                     0C01C 0000          .entry  ACKERMANN,^m<dv,iv,r2,r3,r4>
                                  5E 20 C2 0002          subl2   #32,sp
                               52 04 BC D0 0005          movl    @4(ap),r2
                               53 08 BC D0 0009          movl    @8(ap),r3
                         F4 AD 00000004 EF 9E 000D       movab   $CONSTANT+4,-12(fp)
                           6D 00000000* EF 7E 0015       moval   ADA$HANDLER,(fp)

Line 00006
                                  52 D5 001C             tstl    r2
                                  06 12 001E             bneq    sym.1

Line 00007
                               54 53 01 C1 0020          addl3   #1,r3,r4
                                  59 11 0024             brb     ACKERMANN$RTN_LBL10
                                        0026  sym.1:

Line 00008
                                  53 D5 0026             tstl    r3
                                  1D 12 0028             bneq    sym.2

Line 00009
                            E0 AD 52 01 C3 002A          subl3   #1,r2,-32(fp)
                                  01 18 002F             bgeq    vcg.1
                                     00 0031             halt
                                        0032  vcg.1:
                               00000000 EF DF 0032       pushal  $CONSTANT
                                     E0 AD DF 0038       pushal  -32(fp)
                            00000000* EF 02 FB 003B      calls   #2,ACKERMANN
                                  54 50 D0 0042          movl    r0,r4
                                     38 11 0045          brb     ACKERMANN$RTN_LBL10
                                        0047  sym.2:

Line 00011
                            E4 AD 52 01 C3 0047          subl3   #1,r2,-28(fp)
                               EC AD 52 D0 004C          movl    r2,-20(fp)
                            F0 AD 53 01 C3 0050          subl3   #1,r3,-16(fp)
                                  01 18 0055             bgeq    vcg.2
                                     00 0057             halt
                                        0058  vcg.2:
                                  F0 AD DF 0058          pushal  -16(fp)
                                  EC AD DF 0058          pushal  -20(fp)
```

```
                    00000000* EF 02 FB     005E           calls   #2,ACKERMANN
                           E8 AD 50 D0     0065           movl    r0,-24(fp)
                              E4 AD D5     0069           tstl    -28(fp)
                                 01 18     006C           bgeq    vcg.3
                                    00     006E           halt
                                           006F   vcg.3:
                              E8 AD DF     006F           pushal  -24(fp)
                              E4 AD DF     0072           pushal  -28(fp)
                    00000000* EF 02 FB     0075           calls   #2,ACKERMANN
                              54 50 D0     007C           movl    r0,r4

Line 00013
                                           007F   ACKERMANN$RTN_LBL10:
                              50 54 D0     007F           movl    r4,r0
                                    04     0082           ret

PSECT MAP

Psect Hex Size     Dec Size     Name
    0 00000083          131     Ackermann.$CODE
    1 00000010           16     Ackermann.$CONSTANT


%I, Function body ACKERMANN added to library
    Replaces older version compiled 13-Nov-1984 07:14
```

COMMAND QUALIFIERS

```
  ADA/LIS/MACH ACKERMANN
```

QUALIFIERS USED

```
        /COPY_SOURCE/DEBUG=ALL/ERROR_LIMIT=30/LIST/MACHINE_CODE
        /NODIAGNOSTICS/LIBRARY=ADA$LIB
        /MESSAGES=(COMPILATION_NOTES=NONE,STATUS=L,SUPPLEMENTAL=DLT,WEAK_WARNINGS=DLT,WARNINGS=DLT)
        /NOTE_SOURCE/OPTIMIZE=TIME/NOSHOW/NOSUPPRESS_ALL/NOSYNTAX_ONLY
```

COMPILER INTERNAL TIMING

Phase	CPU seconds	Elapsed seconds	Page faults	I/O count
Initialization	0.48	2.54	158	23
Parser	0.18	0.74	32	1
Static semantics	0.21	1.33	32	0
IL generation	0.40	2.08	143	8
Segment tree	0.16	1.11	72	8
Annotate tree	0.01	0.22	3	0
Flow analysis	0.07	0.14	2	0
Linearize tree	0.14	0.58	60	0
Code generation	0.41	1.26	105	0
Optimizer	0.17	0.45	44	0
Data allocation	0.01	0.00	1	0
Generate code list	0.09	0.47	16	0
Register allocation	0.01	0.01	0	0
Peephole optimization	0.04	0.10	2	0
Branch/jump resolution	0.01	0.01	0	0
Write object module	0.07	0.19	33	0
DST generation	0.01	0.11	4	0
Listing generation	0.10	0.47	13	6
Compilation library	0.40	2.75	62	37
Compiler totals	2.21	11.34	552	75

COMPILATION STATISTICS

```
  Weak warnings:              0
  Warnings:                   0
  Errors:                     0

  Virtual pages used:      5418
  Virtual pages free:     58582
  Peak working set:        2000
  CPU Time:              00:02.21        (448 Lines/Minute + 0.48 seconds init.)
  Elapsed Time:          00:11.34
  Compilation Complete
```

```
SAMPLE_ERRORS                                                    15-Nov-1984 08:57:12    VAX Ada T1.0-2
01                                                               15-Nov-1984 08:55:53    ADAD$:[VATS.AD.PROJ]SAMPLE_ERRORS.ADA;3  (1)

     1   -- SAMPLE_ERRORS.ada
     2   --
     3   -- A simple program containing a sampling of errors to show the error
     4   -- reporting facilities of the VAX Ada compiler
     5   --
     6   --
     7   with NO_SUCH_UNIT; use NO_SUCH_UNIT;
.............1.................2
%E, (1) Unit NO_SUCH_UNIT not found in library
%E, (2) NO_SUCH_UNIT is not declared

     8   with SYSTEM;       use SYSTEM;
     9   procedure SAMPLE_ERRORS is
    10
    11       OBJ1 : NO_SUCH_TYPE := 1;
...................1..............2
%E, (1) NO_SUCH_TYPE is not declared
%I, (2) Type checking not completed -- the type required from context is unknown due to a prior error

    12       OBJ2 : STRING(1..4) := "a";
    13       OBJ3 : STRING := "abcd";
...................1
%E, (1) Array type STRING in predefined STANDARD is not constrained

    14
    15       type E is (E1, E2, E3);
    16       OBJE : E;
............1
%I, (1) The representation of type declaration E at line 15 is forced here

    17       for E use (-1, 5, 18);
................1
%E, (1) The representation of type declaration E at line 15 has already been forced at line 18

    18
    19       type     T is range 0..255;
............1
%I, (1) Parent type chosen for T is predefined SHORT_INTEGER (signed word)

    20       subtype ST is T range 0.0 .. 10.;
.................................1.....2..3
%E, (3) Missing digit -- inserted "0" after "."
%E, (1) Result type of expression is inconsistent with its context which requires (type for T) at line 19
%I, (1) For literal 0.0 the result type is any real   (discarded)
%E, (2) Result type of expression is inconsistent with its context which requires (type for T) at line 19
%I, (2) For literal 10.0 the result type is any real   (discarded)

    21
    22       OBJ4, OBJ5 : array(1..2) := (others => '*');
.....................................1.................2
%E, (1) Found ":=" when expecting "of"
%E, (2) Found ";" when expecting "of"
%I, (2) Declaration ignored due to syntactic errors

    23
```

```
SAMPLE_ERRORS                                                    15-Nov-1984 08:57:12    VAX Ada T1.0-2
01                                                               15-Nov-1984 08:55:53    ADAD$:[VATS.AD.PROJ]SAMPLE_ERRORS.ADA;3  (1)

   24       function F (X : E := E'FIRST) return E;
.............1
%E, (1) Function specification F has neither a corresponding body nor an importing pragma

   25       function F (Y : INTEGER := 0) return E is
   26       begin
   27           return E'VAL(Y);
   28       end;
   29
   30       pragma PACK(T);
.........................1
%W, (1) Subtype T at line 19 is not an array or record type -- pragma PACK ignored

   31       pragma UNKNOWN_PRAGMA(T, T+1, OPTION => MAGIC_NUMBER-2);
.............1
%W, (1) Pragma UNKNOWN_PRAGMA is not known to this implementation - pragma ignored

   32
   33   begin
........1
%I, (1) Function specification F at line 24 has no corresponding body

   34
   35       OBJ1 := -1
........................1
%E, (1) Inserted ";" at end of line

   36       OBJ2 := 'a';
...................1
%E, (1) Inconsistency detected during overloading resolution
%I, (1) For OBJ2 the meaning is variable OBJ2 at line 12 of type STRING in predefined STANDARD
        For literal 'a' the meaning is enumeral 'a' in predefined STANDARD of type CHARACTER in predefined STANDARD

   37       OBJ4 = OBJ5;
.............1....2.3
%E, (2) Replaced "=" with ":="
%E, (1) OBJ4 is not declared
%E, (3) OBJ5 is not declared

   38
   39       case OBJE is
.............1
%E, (1) Some values in the range of enumeration type E at line 15 are missing
%I, (1)     E3

   40           when E1     => null;
.................1
%I, (1) Choice E1 overlaps choice E1 .. E2 at line 42

   41           when E2     => E := 1;
.................1..............2
%E, (2) Type declaration E at line 15 is not a kind of entity that has a value
%I, (1) Choice E2 overlaps choice E1 .. E2 at line 42
```

```
SAMPLE_ERRORS                                                    15-Nov-1984 08:57:12    VAX Ada T1.0-2
01                                                               15-Nov-1984 08:55:53    ADAD$:[VATS.AD.PROJ]SAMPLE_ERRORS.ADA;3  (1)

   42           when E1..E2 => null;
................1
%E, (1) E1 from choice E1 .. E2 overlaps another choice for E1 at line 40
%E, (1) E2 from choice E1 .. E2 overlaps another choice for E2 at line 41

   43       end case;
   44
   45       OBJE := E'SUCC(F);
........................1
%E, (1) Ambiguous expression. The required type is E at line 15 but more than one possible meaning has this type
%I, (1) For F the meanings considered are:
            Call of F at line 24 with all default parameters returning E at line 15
            Call of F at line 25 with all default parameters returning E at line 15
            Function specification F at line 24 with result type E at line 15   (discarded)
            Function body F at line 25 with result type E at line 15   (discarded)

   46
   47   end;
%E, Error(s) compiling procedure body SAMPLE_ERRORS in file ADAD$:[VATS.AD.PROJ]SAMPLE_ERRORS.ADA;3


PORTABILITY SUMMARY

enumeration representation clause
                                    17
unknown pragma(s)                   31
```

```
SAMPLE_ERRORS                                                    15-Nov-1984 08:57:12    VAX Ada T1.0-2
01              Ada Compilation Statistics                       15-Nov-1984 08:55:53    ADAD$:[VATS.AD.PROJ]SAMPLE_ERRORS.ADA;3  (1)

COMMAND QUALIFIERS

  ADA/LIS/SHOW=ALL/MESS=COMP:L SAMPLE_ERRORS


QUALIFIERS USED
        /COPY_SOURCE/DEBUG=ALL/ERROR_LIMIT=30/LIST/NOMACHINE_CODE
        /NODIAGNOSTICS/LIBRARY=ADA$LIB
        /MESSAGES=(COMPILATION_NOTES=L,STATUS=L,SUPPLEMENTAL=DLT,WEAK_WARNINGS=DLT,WARNINGS=DLT)
        /NOTE_SOURCE/OPTIMIZE=TIME/SHOW=PORTABILITY/NOSUPPRESS_ALL/NOSYNTAX_ONLY


COMPILER INTERNAL TIMING

        Phase                          CPU     Elapsed    Page     I/O
                                     seconds   seconds   faults   count
  Initialization                       0.49      2.06      162      15
  Parser                               0.59      2.80      109       9
  Static semantics                     1.05      2.57      166      13
  Listing generation                   0.11      0.19        4       1
  Compilation library                  0.07      0.22        5       3
  Compiler totals                      2.40      8.03      467      41


COMPILATION STATISTICS

  Weak warnings:                0
  Warnings:                     2
  Errors:                      22

  Virtual pages used:        5400
  Virtual pages free:       58600
  Peak working set:          2000
  CPU Time:                  00:02.40        (1468 Lines/Minute + 0.49 seconds init.)
  Elapsed Time:              00:08.03
  Compilation Complete
```

LTSIG Wishlist

The following pages contain the latest LTSIG Wishlist. The wishlist is divided into two separate sections, the first containing seventy-one items for DEC to address, the second containing twenty-nine items for the SIG to address.

For each wishlist their is a maximum number of votes which can be cast per ballot. For the DEC directed list, you may cast forty votes, with no individual item getting more than five. for example, you may cast one vote for each of forty items, five votes for each of eight, or you may weight your ballot as desired within the range. The procedure for the SIG directed wishlist is the same, but since there are fewer items, the total vote count may not exceed twenty-five.

Following the wishlist items is a response sheet. Please indicate your votes on that sheet, by writing in the number of votes per item in the space following the item number. Do not write in yes/no answers, or cast all your votes for one item, or otherwise make life difficult for the people collating the responses. We are pressed for time with this vote, and will disgard illegal ballots rather than bend them to try to fit the system. If there are several interested people at your installation, however, please feel free to copy the answer sheet, and submit several ballots.

We hope to present the results of this wishlist at the next symposium, in New Orleans. This is a tight schedule, so it's important that you return your answer sheets as soon as possible. Please send them directly to me, at:

Alan L. Folsom, Jr.
Dept. 431
Fischer & Porter Co.
E. County Line Road
Warminster, Pa. 18974

The more responses we get, the more weight the wishlist will have with DEC, so please take the time to read the items carefully, and return the answer sheet.

DEC DIRECTED WISHLIST ITEMS

1 DEC should retarget their Ada Compiler to the Motorola 68000.

2 DEC should retarget their Ada Compiler to the Intel 8086 family.

3 DEC should retarget their Ada Compiler to the PDP-11 family.

4 DEC should retarget their Ada Compiler to the 1750A.

5 DEC should retarget their Ada Compiler to other micros.

6 DEC should provide an intermediate language for their Ada compiler (e.g. Diana) so individual companies can produce a backend to target to other computers.

7 DEC should host their Ada Compiler on Ultrix.

8 DEC should use their existing tools to implement an Ada Programming Support Environment (APSE) a la DOD-STD-1467(AR).

9 DEC should market an Ada workstation.

10 DEC should market a general programmer's workstation.

11 DEC should provide versions of RSX-targeted cross compilers that will run under native mode of the VAX, as opposed to the current compatibility mode.

12 DEC should provide a Modula-2 compiler for the VAX.

13 DEC should provide a Modula-2 compiler for the PDP-11's.

14 DEC should provide a "Lint" like tool for VAX C.

15 DEC should provide a more complete implementation of the 'C' run time library and closer compatibility of functions such as vfork() to the UNIX implementation.

16 DEC should provide support for 'C' for LSI-11 based systems.

17 DEC should make all VAX manual available on line, and provide a structured way to access them.

18 DEC should provide software tool support for requirements analysis.

19 DEC should provide software tool support for software design.

20 DEC should provide an automated test generator.

21 DEC should provide an automated Documentation Control Manager.

22 Dec should provide Documentation generation tools.

23 DEC should provide Documentation production tools.

24 DEC should provide a sophisticated text formatter, with things such as multiple fonts, proportional spacing, math/Greek, and macros, (e.g. Scribe, troff, TeX).

25 DEC should provide a Configuration Control Manager.

26 DEC should provide a Project Control Manager.

27 DEC should provide a problem report database tool that is integrated with DEC/CMS and the DEC/Test Manager.

28 DEC should provide a data-dictionary manager to support Structured Analysis (a la Tom Demarco).

29 DEC should provide a static analysis tool.

30 DEC should provide a global Data Flow analysis tool.

31 DEC should provide a dynamic memory usage monitor.

32 DEC should provide an automated package for graphic support for structured analysis and design. such a tool might allow the designer to specify the relaionships between design elements and experiment with alternative designs.

33 DEC should provide a tool to generate the dependancies file for MMS.

34 DEC should provide a Cross Reference Analysis Tool.

35 Compilers and Assemblers should show the names of all files referenced in the summary portion of the listing. This would make it easier to derive dependencies for MMS etc from existing code.

36 DEC should provide an automated MMS.

37 All DEC languages and (where appropriate) should be fully integrated with CDD.

38 DEC should provide a language with pattern matching capabilities, like the languages Snobol and Icon.

39 DEC should market the SCAN language.

40 DEC should document the debugger symbol tables so they can be used for other tools.

41 DEC should be more active in providing software tools and support for ULTRIX; and should promote ULTRIX more heavily.

42 DEC should provide a better Fortran on ULTRIX.

43 DEC should provide a better C on ULTRIX.

44 DEC should provide commercial languages for ULTRIX.

45 DEC should provide more VNX tools.

46 DEC should provide more emulation of the UNIX Environment on VMS.

47 DEC should integrate DTM v2.0 with FMS, so FMS applications can be tested for typos, etc.

48 DEC should provide a way in CMS to remove the last generation.

49 DEC should provide a mechanism to back off a software update to layered products.

50 DEC should market an Integrated CMS/MMS/TestManager type package. DEC should port graphics tools such as the GKS standard directly into the RTL or the terminal drivers.

51 DEC should support GKS to level 2.

52 DEC should increase the number of supported devices for GKS/ob. Alternatively, DEC could provide user driven tools for creating additional device support.

53 DEC should privode window management utilities that use GKS/ob.

54 DEC should provide low cost graphics workstations, or better resolution graphics terminals, and support them with GKS.

55 DEC should provide Graphics oriented MAIL on the VAX.

56 Fortran F/T data collection capability should be provided to users for their own use, possibly as an option at installation time.

57 DEC should provide a convenient mechanism for accessing SQRT, ALOG, and other Fortran Library functions from DEBUG control.

58 DEC should add the complete Block DO and Block CASE constructs to VAX-Fortran.

59 DEC should add the ability to place a .IDENT in Fortran.

60 DEC should provide tools such as PCA, LSE, and the various languages for the Real-time users of the PDP-11/RSX systems.

61 DEC should provide an option in the Linker to force a return to the first library in library list for each symbol being

searched for.

62 All VAX compilers should support a /Check=Argument_Count qualifier which would generate code at the entry point of routines to validate the number of arguments passed to the routine.

63 DEC should provide an efficient and easy to use high level language with data management capability integrated.

64 DEC should provide a PDL which could be used to rapidly expess the logic of a program module, and which could be automated to check syntax and completeness as well as produce reports.

65 DEC should provide the ability to turn language keywords off, so that existing languages can be used as PDL's.

66 DEC should provide on-line CAI courses for all new software tools.

67 DEC product brochures should contain more technical information about actual capabilities and requirements for a language/tool.

68 DEC should make videotapes explaining the technical aspects of its new products, available for rental/purchase at low cost.

69 DEC should provide product brochures that describe the benefits of new software products, in terms of productivity/money savings, aimed at upper management.

70 DEC should provide a description of all tools in on-line help, even if they are not licensed on the system, so people know what's available and what the general capabilities for each tool are.

71 DEC should provide formal seminars on the use of its new languages and tools through Educational Services.

SIG DIRECTED WISHLIST ITEMS

1 The SIG newsletter should be published more frequently.

2 The SIG should give "How To" tutorials on new Tools.

3 The SIG should provide more sessions on Software Requirements and Design issues.

4 The SIG should give more sessions on how to manage software projects.

5 The SIG should give more sessions on how DEC develops software internally.

6 The Sig should provide more sessions on software testing techniques.

7 The SIG should sponsor more user presentations on the experiences with or evaluations of various software development tools.

8 The SIG should give Pre Symposium Seminars on Ada.

9 The SIG should give Pre Symposium Seminars on CMS/MMS.

10 The SIG should give Pre Symposium Seminars on DEC/Test Manager.

11 The SIG should give Pre Symposium Seminars on Software Methodologies.

12 The SIG should give Pre Symposium Seminars on Software Project Management.

13 The SIG should give Pre Symposium Seminars on Software Development Tools.

14 The SIG should give Pre Symposium Seminars on Text Formatters.

15 The SIG should give Pre Symposium Seminars on Software Testing Techniques.

16 The SIG should give Pre Symposium Seminars on Pascal.

17 The SIG should give Pre Symposium Seminars on "C".

18 The SIG should give Pre Symposium Seminars on Modula-II.

19 The SIG should give Pre Symposium Seminars on Artifical Intelligence.

20 The SIG should give Pre Symposium Seminars concerning the selection and evaluation of programming languages for projects.

21 The SIG should establish a Configuration Control Working Group.

22 The SIG should survey database products evaluating their suitability to support a software development environment.

23 The SIG should provide DECUS "C" in RT readable format.

24 The SIG should research and publish the availability of Praxis.

25 The SIG should sponsor the conversion of the Software Tools from Ratfor to "C", perhaps by using a Fortran to C conversion utility.

26 The SIG should provide information about Program Design Languages, particularly the IEEE standard.

27 The SIG should publish a Contact List for SIG officers and Special Interest Area coordinators.

28 The SIG should sponsor a panel at the next symposium comparing Pascal, "C", Ada, and so forth.

29 The SIG should sponsor sessions concerning the selection and evaulation of programming languages for projects.

DEC DIRECTED WISHLIST ITEMS

In this section, each ballot may contain a total of forty votes, with no more than five votes per item. i.e., eight items may each receive five votes, forty items may each receive one vote, etc.

1 ____	16 ____	31 ____	46 ____	61 ____
2 ____	17 ____	32 ____	47 ____	62 ____
3 ____	18 ____	33 ____	48 ____	63 ____
4 ____	19 ____	34 ____	49 ____	64 ____
5 ____	20 ____	35 ____	50 ____	65 ____
6 ____	21 ____	36 ____	51 ____	66 ____
7 ____	22 ____	37 ____	52 ____	67 ____
8 ____	23 ____	38 ____	53 ____	68 ____
9 ____	24 ____	39 ____	54 ____	69 ____
10 ____	25 ____	40 ____	55 ____	70 ____
11 ____	26 ____	41 ____	56 ____	71 ____
12 ____	27 ____	42 ____	57 ____	
13 ____	28 ____	43 ____	58 ____	
14 ____	29 ____	44 ____	59 ____	
15 ____	30 ____	45 ____	60 ____	

SIG DIRECTED WISHLIST ITEMS

In this section, each ballot may contain a total of twenty-five votes, with no more than five votes per item. i.e., five items may each receive five votes, twenty-five items may each receive one vote, etc.

1 ____	7 ____	13 ____	19 ____	25 ____
2 ____	8 ____	14 ____	20 ____	26 ____
3 ____	9 ____	15 ____	21 ____	27 ____
4 ____	10 ____	16 ____	22 ____	28 ____
5 ____	11 ____	17 ____	23 ____	29 ____
6 ____	12 ____	18 ____	24 ____	

- POLICY NOTICE TO ALL ATTENDEES OR CONTRIBUTORS
"DECUS PRESENTATIONS, PUBLICATIONS, PROGRAMS, OR ANY OTHER PRODUCT WILL NOT CONTAIN TECHNICAL DATA/INFORMATION THAT IS PROPRIETARY, CLASSIFIED UNDER U.S. GOVERNMENT SECRECY LAWS, CONTROLLED BY NON-DISCLOSURE AGREEMENTS WITH THE U.S. GOVERNMENT OR THIRD PARTIES, OR GOVERNED BY THE U.S. DEPARTMENT OF STATE'S INTERNATIONAL TRAFFIC IN ARMS REGULATIONS (ITAR)."

DECUS SUBSCRIPTION SERVICE
DIGITAL EQUIPMENT COMPUTER SOCIETY
249 NORTHBORO ROAD, (BPO2)
MARLBORO, MA 01752

Bulk Rate
U.S. Postage
PAID
Permit No. 18
Leominster, MA
01453

MOVING OR REPLACING A DELEGATE?

Please notify us immediately to guarantee continuing receipt of DECUS literature. Allow up to six weeks for change to take effect.

( ) Change of Address
( ) Delegate Replacement

DECUS Membership No.: _____
Name: _____
Company: _____
Address: _____
_____
State/Country: _____
Zip/Postal Code: _____
Phone No.: _____

Mail to: **DECUS - Attn: Subscription Service**
249 Northboro Road, (BPO2)
Marlboro, MA 01752 USA

Affix mailing label