MicroVMS Workstation Release Notes, Version 3.0

Order Number: AA–HR84A–TN

May 1986

This document provides supplemental information about Version 3.0 of MicroVMS Workstation graphics software. It describes changes between Version 2.0 and Version 3.0, lists problems and restrictions, includes notes to existing documentation, and provides an appendix for VMS data types.

Revision/Update Information: This manual supersedes the MicroVMS Workstation Release Notes, Version 2.0.

Software Version: VAX/VMS Version 4.4

digital equipment corporation
maynard, massachusetts

May 1986

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Copyright ©1986 by Digital Equipment Corporation

All Rights Reserved.
Printed in U.S.A.

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem-10	PDP	VT
DECSYSTEM-20	PDT	
DECUS	RSTS	
DECwriter	RSX	digital

ZK-3166

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA & PUERTO RICO*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire
03061

CANADA

Digital Equipment
of Canada Ltd.
100 Herzberg Road
Kanata, Ontario K2K 2A6
Attn: Direct Order Desk

INTERNATIONAL

Digital Equipment Corporation
PSG Business Manager
c/o Digital's local subsidiary
or approved distributor

In Continental USA and Puerto Rico call 800-258-1710.

In New Hampshire, Alaska, and Hawaii call 603-884-6660.

In Canada call 800-267-6215.

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminster, Massachusetts 01473.

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a trademark of the American Mathematical Society.

Contents

Preface vii

Chapter 1 Differences Between Version 2.0 and Version 3.0

1.1 Summary of New and Changed VAXstation Features 1-1
1.1.1 Changes to the User Interface 1-1
1.1.2 Changes to the Programming Interface 1-2
1.1.2.1 New UIS Routines 1-2
1.1.2.2 New UISDC Routines 1-4
1.1.2.3 New Chapters 1-5
1.1.2.4 New UIS Writing Modes 1-5
1.1.2.5 New Fonts in Technical Character Set 1-5
1.1.2.6 New Text Attributes 1-5
1.1.2.7 Changes to Existing UIS Routines 1-6
1.1.2.8 Display Lists and Segmentation 1-6
1.1.2.9 UIS Metafiles 1-6
1.1.2.10 Shrinking Viewports and Expanding Icons 1-6
1.1.2.11 Obsolete UIS Routines in Version 2.0 1-6
1.1.3 Changes to the Driver Interface 1-7

1.2 Fonts 1-7
1.2.1 New Font Utility 1-7

1.3 Demonstration Software 1-8

1.4 Hardcopy UIS 1-9

Chapter 2 Problems and Restrictions

2.1 User Interface 2-1
2.1.1 VAXstation II/GPX Boot Problem 2-1
2.1.2 Restriction with Autologin 2-1
2.1.3 Restriction to Print Screen Destination 2-1

2.2 Programming Interface 2-2
2.2.1 UIS$CIRCLE and UIS$ELLIPSE—Overflow Problem 2-2
2.2.2 UIS$DISABLE_KB and UIS$DISABLE_VIEWPORT_KB—AST Not Delivered 2-2
2.2.3 UIS$SET_KB_ATTRIBUTES—Spurious Data 2-2
2.2.4 UIS$SET_POINTER_AST—Late Execution of Exit AST Routines 2-2
2.2.5 Drawing Images That Use 8 Bits per Pixel 2-3
2.2.6 Text Problems 2-3
2.2.6.1 Extracting Transformed Control Lists 2-3
2.2.6.2 Tabs Within Control Lists 2-3
2.2.6.3 Text Placement and Display Lists 2-4
2.2.6.4 Text Formatting Problems 2-4
2.2.6.5 Sloped Text 2-4
2.2.6.6 Restrictions on Writing Modes That Change the Background 2-5

2.3 Device Driver Interface 2-6
2.3.1 MOVE/ROTATE DOP—Specifying Scaling 2-6

Chapter 3 Notes to Published Documentation

3.1 Directory Change for Font Utility 3-1

3.2 Notes to *MicroVMS Workstation Graphics Programming Guide* . . 3-1
3.2.1 UIS$GET_OBJECT_ATTRIBUTES—Missing Object Type . . . 3-1
3.2.2 Symbol Prefix Change 3-2

Appendix A VMS Data Types

A.1 VMS Data Types A-1
A.2 VAX BLISS Implementation A-21
A.3 VAX C Implementation A-24
A.4 VAX FORTRAN Implementation A-27
A.5 VAX MACRO Implementation A-32
A.6 VAX PASCAL Implementation A-35
A.7 VAX PL/I Implementation A-39

Index

Tables

A–1 VMS Data Types A-2
A–2 VAX BLISS Implementation A-21
A–3 VAX C Implementation A-24
A–4 VAX FORTRAN Implementation A-27
A–5 VAX MACRO Implementation A-32
A–6 VAX PASCAL Implementation A-35
A–7 VAX PL/I Implementation A-39

Preface

This document provides supplemental information about the Version 3.0 MicroVMS Workstation software. It describes all changes to the software since Version 2.0. If you have not already done so, please read the Read-Me First card included with your documentation.

Intended Audience

This manual is intended for use by graphic programmers and general users who should know about new features, problems and restrictions, and changes to existing documentation. All users should read this document before using the MicroVMS Workstation graphics software.

Structure of This Document

The *MicroVMS Workstation Release Notes, Version 3.0,* are arranged in four sections that cover the following topics:

- Differences Between Version 2.0 and Version 3.0
- Problems and Restrictions
- Notes to Published Documentation
- VMS Data Types—Appendix A

Associated Documents

The following manuals are related to this document:

- *VWS Installation Guide*
- *MicroVMS Workstation User's Guide*
- *MicroVMS Workstation Graphics Programming Guide*
- *MicroVMS Workstation Video Device Driver Manual*
- *MicroVMS Workstation Guide to Printing Graphics*

Conventions Used in This Document

Unless otherwise noted, the following conventions are used in this manual in displaying examples and the requirements of user input to the system.

Convention	Meaning
RET	A symbol with a one- to six-character abbreviation indicates that you press a key on the terminal, for example, RET .
CTRL/x	The phrase CTRL/x indicates that you must press the key labeled CTRL while you simultaneously press another key, for example, CTRL/C, CTRL/Y, CTRL/O.
$ SHOW TIME 05-JUN-1985 11:55:22	Command examples show all output lines or prompting characters that the system prints or displays in black letters. All user-entered commands are shown in red letters.
Ellipsis . . .	Vertical series of periods, or ellipsis, mean either that not all the data that the system would display in response to the particular command is shown or that not all the data a user would enter is shown.
file-spec, . . .	Horizontal ellipsis indicates that additional parameters, values, or information can be entered.

Convention	Meaning
[logical-name]	Square brackets indicate that the enclosed item is optional. (Square brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
quotation marks apostrophes	The term quotation marks is used to refer to double quotation marks ("). The term apostrophe is used to refer to a single quotation mark (').

Chapter 1

Differences Between Version 2.0 and Version 3.0

This chapter describes major changes in the MicroVMS Workstation software since Version 2.0. For additional changes, see Chapter 3, "Notes to Published Documentation."

1.1 Summary of New and Changed VAXstation Features

The following changes have been made for Version 3.0 of the MicroVMS Workstation software and are reflected in the corresponding documentation.

1.1.1 Changes to the User Interface

The following changes are reflected in the *MicroVMS Workstation User's Guide*:

- Automatic login—If automatic login is enabled, you need only log in to the system once. Any terminal-emulator windows you create subsequently will execute your login procedure automatically.
- Color display setup—On color and intensity systems, you can adjust the color shades using the color-setup options in the Workstation Setup menu. On a monochrome system, this setup item permits you to change between black, white, and grey.
- Mouse fallback mechanism—The pointer on the screen can be moved by using the CTRL and SHIFT keys in conjunction with the arrow keys. When used together with E4, E5, or E6, CTRL/SHIFT performs the same functions as the three mouse buttons.
- The Window Options menu includes new options:

  "Shrink to an icon," which provides a user interface for shrinking windows to icons.

"Additional options," which you may enable for your own use. The VT220 emulator uses this option to do a per-terminal setup.

- The banner on the terminal emulator window is always black, regardless of window background color. (In previous versions, the banner was always the reverse of the window background color.) On color and intensity workstations, the default is also black, but can be altered to any desired hue with the color-setup menu.
- The "Print (portion of) screen" option of the Workstation Options menu option includes a change in the SELECT button function on the mouse. Instead of two separate clicks of the SELECT button, a single *click and hold down* function is used to delineate a portion of the screen for printing.
- The VT100 terminal emulator has been replaced by a VT220 terminal emulator.
- The format and contents of the Workstation Setup menu have changed. Many new options have been added. The options "Window memory size" and "Text scrolling rate" have been removed.

1.1.2 Changes to the Programming Interface

The following sections describe changes made to the programming interface after UIS Version 2.0. These changes are documented in the *MicroVMS Workstation Graphics Programming Guide*.

1.1.2.1 New UIS Routines

The following UIS routines were added:

Function	Routine
AST-enabling	UIS$SET_ADDOPT_AST UIS$SET_EXPAND_ICON_AST UIS$SET_TB_AST UIS$SET_SHRINK_TO_ICON_AST

Function	Routine
Color	UIS$CREATE_COLOR_MAP UIS$CREATE_COLOR_MAP_SEG UIS$DELETE_COLOR_MAP UIS$DELETE_COLOR_MAP_SEG UIS$GET_COLORS UIS$GET_HW_COLOR_INFO UIS$GET_INTENSITIES UIS$GET_VCM_ID UIS$HLS_TO_RGB UIS$HSV_TO_RGB UIS$RESTORE_CMS_COLORS UIS$RGB_TO_HLS UIS$SET_COLORS UIS$RGB_TO_HSV UIS$SET_INTENSITIES
Display list	UIS$COPY_OBJECT UIS$DELETE_OBJECT UIS$DELETE_PRIVATE UIS$EXECUTE UIS$EXECUTE_DISPLAY UIS$EXTRACT_HEADER UIS$EXTRACT_OBJECT UIS$EXTRACT_PRIVATE UIS$EXTRACT_REGION UIS$EXTRACT_TRAILER UIS$FIND_PRIMITIVE UIS$FIND_SEGMENT UIS$GET_CURRENT_OBJECT UIS$GET_NEXT_OBJECT UIS$GET_OBJECT_ATTRIBUTES UIS$GET_PARENT_SEGMENT UIS$GET_PREVIOUS_OBJECT UIS$GET_ROOT_SEGMENT UIS$INSERT_OBJECT UIS$PRIVATE UIS$SET_INSERTION_POSITION UIS$TRANSFORM_OBJECT
Graphics	UIS$LINE UIS$LINE_ARRAY

Function	Routine
Keyboard and pointer	UIS$CREATE_TB UIS$DELETE_TB UIS$DISABLE_TB UIS$ENABLE_TB UIS$GET_TB_INFO UIS$GET_TB_POSITION
Text	UIS$GET_CHAR_ROTATION UIS$GET_CHAR_SIZE UIS$GET_CHAR_SLANT UIS$GET_FONT_ATTRIBUTES UIS$GET_TEXT_FORMATTING UIS$GET_TEXT_MARGINS UIS$GET_TEXT_PATH UIS$GET_TEXT_SLOPE UIS$SET_CHAR_ROTATION UIS$SET_CHAR_SIZE UIS$SET_CHAR_SLANT UIS$SET_TEXT_FORMATTING UIS$SET_TEXT_MARGINS UIS$SET_TEXT_PATH UIS$SET_TEXT_SLOPE
Windowing	UIS$EXPAND_ICON UIS$GET_VIEWPORT_ICON UIS$GET_WINDOW_SIZE UIS$SHRINK_TO_ICON

1.1.2.2 New UISDC Routines

The following UISDC routines are new for Version 3.0.

- UISDC$ALLOCATE_DOP
- UISDC$EXECUTE_DOP_ASYNCH
- UISDC$EXECUTE_DOP_SYNCH
- UISDC$GET_CHAR_SIZE
- UISDC$GET_TEXT_MARGINS
- UISDC$LINE
- UISDC$LINE_ARRAY
- UISDC$LOAD_BITMAP

- UISDC$QUEUE_DOP
- UISDC$SET_CHAR_SIZE
- UISDC$SET_TEXT_MARGINS

1.1.2.3 New Chapters

Three new chapters describing color concepts, transformations, and color programming have been added since Version 2.0.

- Color Concepts
- Geometric and Attribute Transformations
- Programming in Color

1.1.2.4 New UIS Writing Modes

Five new writing modes have been added since Version 2.0.

- UIS$C_MODE_BIC
- UIS$C_MODE_BICN
- UIS$C_MODE_BIS
- UIS$C_MODE_BISN
- UIS$C_MODE_COPYN

1.1.2.5 New Fonts in Technical Character Set

Twelve new fonts have been added to the technical character set since Version 2.0.

1.1.2.6 New Text Attributes

The following new text attributes have been added to the programming interface:

- Character rotation
- Character scaling
- Character slant
- Text formatting
- Text margins
- Text path
- Text slope

1.1.2.7 Changes to Existing UIS Routines

UIS$BEGIN_SEGMENT

UIS$BEGIN_SEGMENT now returns a segment identifier that can be referenced by other display list routines. For example, this allows traversing segments and segment paths.

UIS$MEASURE_TEXT and UIS$TEXT

You can now use control lists with UIS$TEXT and UIS$MEASURE_TEXT.

UIS$DISABLE_DISPLAY_LIST and UIS$ENABLE_DISPLAY_LIST

Additional arguments have been included that control updates to display screens and display lists.

UIS$SET_POINTER_PATTERN and UISDC$SET_POINTER_PATTERN

If you are using a color system, you can now specify a pointer pattern outline and flags to bind the pointer to a particular region.

1.1.2.8 Display Lists and Segmentation

The chapter on display lists and segmentation has been expanded with more examples.

1.1.2.9 UIS Metafiles

You can now create and store metafiles of generically encoded instructions as files and reexecute the file.

1.1.2.10 Shrinking Viewports and Expanding Icons

You can now shrink display viewports and expand icons.

1.1.2.11 Obsolete UIS Routines in Version 2.0

The following routines are now obsolete:

- UIS$GET_LEFT_MARGIN
- UIS$SET_LEFT_MARGIN
- UISDC$GET_LEFT_MARGIN
- UISDC$SET_LEFT_MARGIN

They have been replaced by the following routines:

- UIS$SET_TEXT_MARGINS
- UIS$GET_TEXT_MARGINS

1.1.3 Changes to the Driver Interface

The following changes are reflected in the *MicroVMS Workstation Video Device Driver Manual*:

- The QDSS driver is available (on systems with QDSS hardware). The QDSS driver permits you to draw multiplane (color) images through the use of the hardware-assisted Drawing Operation Primitive (DOP) interface. The QDSS system also uses a QIO interface. Read Chapters 1 and 2 for an overview of the driver.
- New QDSS-specific QIOs—see Chapter 4.
- New DOP interface—see Chapter 5.
- New UISDC routines for use with the DOP interface—see Chapter 5.

1.2 Fonts

The following sections describe new font features.

1.2.1 New Font Utility

A Font Utility has been added in Version 3.0. This utility permits you to add new user-defined fonts to the system. See Section 2.5 of the *VWS Installation Guide* for a description of how to install and use user-defined fonts.

Note that the documentation gives an incorrect file specification for this utility. Any references in documentation to SYS$FONT:UISFONTS should be changed to SYS$SYSTEM:UISFONTS.

Font File Types

In previous versions of UIS, the font files supplied in SYS$FONT had the file type FNT. Beginning with Version 3.0, the font files will have the file types shown in the following table.

System	File Type
VAXstation I	VWS$FONT
VAXstation II	VWS$FONT
VAXstation II/GPX	VWS$VAFONT

Loading UIS Fonts

In previous versions of UIS, a font was associated with an attribute block when a call to UIS$SET_FONT was made, but the font was not actually loaded until it was used in a call to UIS$TEXT or a fill routine. Beginning with Version 3.0, the font will be loaded during the call to UIS$SET_FONT. UIS$SET_FONT now signals all errors associated with loading the font—not UIS$TEXT or the fill routine. This may result in errors being signaled that would not have been signaled before. For example, if UIS$SET_FONT is called with a nonexistent font but that font is never used, UIS$SET_FONT will now signal an error when it would not have signaled before.

1.3 Demonstration Software

The MicroVMS Workstation Version 3.0 software kit includes a floppy disk containing a number of programs that demonstrate some of the capabilities of the workstation.

The floppy is labeled "MicroVMS Workstation Demos VWSDEMO030 1/1." Use VMSINSTAL to install the demonstration programs. It will place them in a directory that it creates called SYS$SYSDEVICE:[VWSDEMO]. The following files will be in the directory when VMSINSTAL completes:

banner.com—Procedure that produces interactive banner

banner.exe—Executable program invoked by banner.com

compile_all.com—Procedure that compiles and links quick and cube

cube.for—Graphics program that produces a rotating cube (source)

cube.exe—Executable version

declander.exe—Executable game

declander.help—Directions for playing DEClander

quick.for—Graphics program that produces swirling lines (source)

quick.exe—Executable version

setup_colors.pas—Linked with quick object module

sight.exe—Object-oriented graphics editor

sight.mem—Documentation for SIGHT

1.4 Hardcopy UIS

The MicroVMS Workstation Version 3.0 software kit contains media and documentation for Hardcopy UIS (HCUIS). HCUIS enables users and applications to translate UIS pictures to the formats needed for printing on a variety of hardcopy devices.

The kit for HCUIS consists of the following:

- The RENDER command, which translates and displays UIS picture files
- Four translators

  UIS to PostScript™
  UIS to sixel
  UIS to HPGL
  UIS to ReGIS

- HCUIS$ routines

See the *MicroVMS Workstation Guide to Printing Graphics* for additional information about HCUIS.

Chapter 2

Problems and Restrictions

This chapter describes problems and restrictions you may encounter when using Version 3.0 of the MicroVMS Workstation software. The chapter describes the problems and restrictions of the user interface, programming interface, and device driver interface in separate sections.

2.1 User Interface

The following sections describe problems and restrictions in the user interface.

2.1.1 VAXstation II/GPX Boot Problem

If your VAXstation II/GPX system appears to be hung during a boot or shutdown operation, press the F2 key. The system may have written a message to the operator console window and may be waiting for you to read the message before it continues.

2.1.2 Restriction with Autologin

If you log in to your first terminal emulator window while autologin is enabled and then quickly create another emulator, you may not be automatically logged in to the second window. This is because the process information for the first window has not yet been saved. A solution is to wait until the initial login procedure has completely executed before creating additional terminal emulators.

2.1.3 Restriction to Print Screen Destination

Do not set the print destination (using the Workstation setup) to be a WT device that is already displayed on the screen. Printing to such a device will cause the system to hang.

2.2 Programming Interface

The following sections describe problems and restrictions in the programming interface.

2.2.1 UIS$CIRCLE and UIS$ELLIPSE—Overflow Problem

On both VAXstation II and VAXstation II/GPX systems, UIS$CIRCLE or UIS$ELLIPSE may occasionally draw large circles or ellipses incorrectly, due to an overflow in the coordinate conversion.

2.2.2 UIS$DISABLE_KB and UIS$DISABLE_VIEWPORT_KB—AST Not Delivered

When a virtual keyboard is explicitly disabled by a UIS$DISABLE_KB or UIS$DISABLE_VIEWPORT_KB, the Lose Keyboard AST routine will not be delivered. If your application depends on this AST being received after the virtual keyboard has been detatched from the physical keyboard, you must explicitly call the AST routine after disabling the keyboard.

2.2.3 UIS$SET_KB_ATTRIBUTES—Spurious Data

When you are using the Up/Down key transitions enabled by UIS$SET_KB_ATTRIBUTES, you may get spurious data when the physical keyboard is attached to the window. A possible solution is to ignore incoming data for a short time after getting a GAIN_KB_AST. This will be fixed in a future release.

2.2.4 UIS$SET_POINTER_AST—Late Execution of Exit AST Routines

When two contiguous regions have been set up with UIS$SET_POINTER_AST, it is possible to execute an exit AST intended for the previous region after executing the first movement AST routine for the new region.

To clarify, when you exit from one region and enter the other, three actions occur in the following order:

1. The last movement in the first region
2. Exiting from the first region
3. The first movement in the new region

However, the ASTs asociated with these actions may be delivered out of order: last movement, first movement, exit.

The recommended solution is to test for the first movement on any contiguous region and emulate the exit AST before taking any other action. According to the application, you may wish to emulate only some essential portion of the exit AST routine and let the actual AST perform the remainder when it is executed.

2.2.5 Drawing Images That Use 8 Bits per Pixel

When drawing images that use 8 bits per pixel, use the COPY writing mode (UIS$C_MODE_COPY) to use the pixel values as direct indices into the color map. This writing mode will copy each pixel value from the image into the bit map without any changes to the data.

The default writing mode (UIS$C_MODE_OVER) will NOT work like copy mode.

2.2.6 Text Problems

The following sections describe problems and restrictions to be aware of when using text with the programming interface.

2.2.6.1 Extracting Transformed Control Lists

The result of extracting transformed control list text is undefined. That is, the following sequence of routine calls will produce a buffer containing unpredictable results:

```
UIS$TEXT(vd_id, atb, text_string, x, y, ctllist, ctllen)
obj_id = UIS$GET_CURRENT_OBJECT(vd_id)
UIS$TRANSFORM_OBJECT(obj_id, matrix, atb)
UIS$EXTRACT_OBJECT(obj_id, buflen, bufaddr, retlen)
```

Note, however, that the results on the screen and within UIS's internal display list will be correct.

2.2.6.2 Tabs Within Control Lists

Text that has a control list containing relative or absolute tabs may produce unexpected results if it falls under any of the following categories:

- Sloped
- Written with a nondefault major text path (for example, UIS$C_TEXT_PATH_LEFT)
- Transformed (using UIS$TRANSFORM_OBJECT or UIS$COPY_OBJECT) to be sloped

Slanted text that has a control list containing relative or absolute tabs may erase portions of characters when written with any writing mode that writes the background, such as overlay negate.

2.2.6.3 Text Placement and Display Lists

After calling UIS$TEXT and UIS$NEW_TEXT_LINE to create lines of text, you may wish to insert more text at the end of a line. Since current text position is undefined when you insert text into a display list, you should always explicitly position your inserted text.

2.2.6.4 Text Formatting Problems

The following sections describe problems and restrictions to be aware of when formatting text.

Enabling and Disabling Text Formatting

If the original input attribute block for a UIS$TEXT or UISDC$TEXT call with a control list does not have text formatting enabled and a subsequent ATB in the control list does format text, the results are undefined.

Formatted Text with Nondefault Attributes

Formatted text gives undefined results if the text or vertical major text path being written has nondefault attributes of slant, slope, rotation, or character size. The same is true for formatted text that is transformed to have nondefault attributes of slant, slope, rotation, or character size.

Full Text Justification of Nonstandard Fonts

For fully justified text to work correctly with fonts other than those supplied on the distribution kit, the glyph for the space character must be in the 33rd position in the font, which is the same position as the ASCII space character in the supplied fonts.

2.2.6.5 Sloped Text

The following sections describe the behavior of sloped text when it is viewed through a distorted viewport.

Text Slope Angles with Distorted Windows

If sloped text is displayed using UIS$TEXT and a distorted window/viewport mapping (that is, the aspect ratio of the window differs from the aspect ratio of the viewport), the results differ depending on whether character scaling is enabled. If character scaling is not enabled, the angle is displayed relative to the device. For example, at a slope of 45 degrees (with major path right) each character position will move up and right by the same number of pixels. If character scaling IS enabled, the slope is measured relative to world coordinates. For example, at a slope of 45 degrees (with major path right) each character position will move up and right by the same world-coordinate amount.

If sloped text is displayed using UISDC$TEXT, the slope angles are always measured based on device coordinates regardless of whether scaling is enabled.

This behavior will be permanent and is consistent with other uses of unscaled text and UISDC routines with distorted viewport/window mappings.

Text Slope Angles on VR100 Monitors

If sloped text is displayed using UIS$TEXT with character scaling disabled, the angles appear to be distorted, even if the viewport and window aspect ratios are the same. The reason for this behavior is that the angle is being drawn in device coordinates, and pixels on a VR100 are not square. To make the angle appear correct, you must enable character scaling using the UIS$SET_CHAR_SIZE routine.

If sloped text is displayed using UISDC$TEXT, angles will always appear distorted on a VR100 monitor.

This behavior will be permanent and is consistent with other uses of unscaled text and UISDC routines with VR100 monitors.

NOTE: The only supported hardware device that uses a VR100 monitor is a VAXstation I.

2.2.6.6 Restrictions on Writing Modes That Change the Background

The following sections describe restrictions that apply when writing text with writing modes that modify the background.

Scaled Text on GPX systems

The VAXstation II/GPX hardware has the characteristic that when it compresses text, it can write both the background and foreground colors into the same pixel on the screen. This means that if you are using a writing mode that changes background pixels (for example, REPL or REPLN), the foreground pixels can be overwritten. This can result in what appears to be missing pixels in scaled characters. Note that scaling is done implicitly if text is drawn at slope, rotation, or slant angles that are not 0 or multiples of 90 degrees.

This will be a permanent restriction.

Text Written at Angles

If text is written in a mode that causes the background of the cell to be written (for example, REPL or REPLN), there may be unwritten pixels between adjacent character cells. This effect only happens with slope, rotation, or slant angles that are not 0 or multiples of 90.

We believe that this is an unavoidable effect of rasterization, but we will continue to investigate possible future improvements.

2.3 Device Driver Interface

The following section describes a problem in the device driver interface.

2.3.1 MOVE/ROTATE DOP—Specifying Scaling

There is a problem in specifying scaling in the MOVE/ROTATE Drawing Operation Primitive (DOP).

If the **source_width** divided by the **vec1_length**, or the **source_height** divided by the **vec2_length**, cannot be represented exactly in 12 bits or fewer, a pixel may be dropped from the end of the source.

The recommended solution is to decrease the vector length (usually by a constant 1 or 2 pixels), without changing the Dx or Dy values, until the full source is drawn correctly.

Chapter 3

Notes to Published Documentation

This chapter describes omissions and errors in existing documentation.

3.1 Directory Change for Font Utility

Any references in documentation to SYS$FONT:UISFONTS (Font Utility) should be changed to SYS$SYSTEM:UISFONTS.

The *VWS Installation Guide,* Section 2.5, describes the procedures for adding user-defined fonts to the workstation. Some of these procedures provide examples that show how to invoke the Font Utility. For example:

```
$ RUN SYS$FONT:UISFONTS X
Font Utility>
```

The command in this example should be changed to the following:

```
$ RUN SYS$SYSTEM:UISFONTS X
Font Utility>
```

3.2 Notes to *MicroVMS Workstation Graphics Programming Guide*

The following sections describe corrections to *MicroVMS Workstation Graphics Programming Guide.*

3.2.1 UIS$GET_OBJECT_ATTRIBUTES—Missing Object Type

The routine UIS$GET_OBJECT_ATTRIBUTES returns a value that identifies an object. UIS$C_OBJECT_NEW_TEXT_LINE is the symbol of a value that is not listed in the UIS$GET_OBJECT_ATTRIBUTES routine description.

3.2.2 Symbol Prefix Change

In Section 15.2.1, the symbol UIS$C_LENGTH_DIFF should be GER$C_LENGTH_DIFF.

Appendix A

VMS Data Types

A.1 VMS Data Types

The VMS Usage entry in the documentation format for system routines indicates the VMS data type of the argument. Each VMS data type has only one storage representation. For example, the VMS data type access_mode is an unsigned byte. In addition, a VMS data type may or may not have a conceptual meaning.

Most VMS data types may be considered as conceptual types; that is, they carry meaning that is unique in the context of the VMS operating system. The **access_mode** is one of these. The storage representation of this VMS type is an unsigned byte, and the conceptual content of this unsigned byte is the fact that it designates a hardware access mode and has therefore only four valid values: *0*, designating kernel mode; *1*, executive mode; *2*, supervisor mode; and *3*, user mode. However, some VMS data types are not conceptual types; that is, they specify a storage representation but carry no other semantic content from the point of view of VAX/VMS. For example, the VMS data type **byte_signed** is not a conceptual type.

NOTE: The VMS Usage entry is NOT a traditional data type such as the VAX standard data types byte, word, longword and so on. It is significant only within the context of the VMS operating system environment and is intended solely to expedite data declarations within application programs.

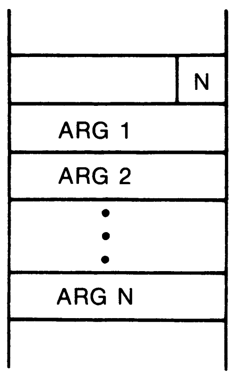
To use the VMS Usage entry, perform the following procedure:

1. Find the data type in Table A–1 and read its definition.
2. Find the same VMS data type in the appropriate VAX language implementation table (Tables A–2 through A–7) and its corresponding source language type declaration.
3. Use this code as your type declaration in your application program. Note that, in some instances, you may have to modify the declaration.

Table A-1 lists and describes the VMS data types.

Table A-1 VMS Data Types

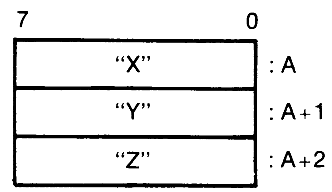
Data Type	Definition
access_bit_names	Homogeneous array of 32 quadword descriptors; each descriptor defines the name of one of the 32 bits in an access mask. The first descriptor names bit <0>, the second descriptor names bit <1>, and so on.
access_mode	Unsigned byte denoting a hardware access mode. This unsigned byte can take four values: *0* specifies kernel mode; *1*, executive mode; *2*, supervisor mode; and *3*, user mode.
address	Unsigned longword denoting the virtual memory address of either data or code, but not of a procedure entry mask (which is of type **procedure**).
address_range	Unsigned quadword denoting a range of virtual addresses, which identify an area of memory. The first longword specifies the beginning address in the range; the second longword specifies the ending address in the range.
arg_list	Procedure argument list consisting of one to 256 longwords. The first longword contains an unsigned integer count of the number of successive, contiguous longwords, each of which is an argument to be passed to a procedure by means of a VAX CALL instruction. The argument list has the following format:



ZK-4204-85

Table A-1 (Cont.) VMS Data Types

Data Type	Definition
ast_procedure	Unsigned longword integer denoting the entry mask to a procedure to be called at AST level. (Procedures that are not to be called at AST level are of type **procedure**.)
boolean	Unsigned longword denoting a Boolean truth value flag. This longword may have only two values: *1* (true) and *0* (false).
byte_signed	This VMS data type is the same as the data type **byte integer (signed)** in Table 6-1.
byte_unsigned	This VMS data type is the same as the data type **byte (unsigned)** in Table 6-1.
channel	Unsigned word integer that is an index to an I/O channel.
char_string	String of from 0 to 65,535 8-bit characters. This VMS data type is the same as the data type **character string** in Table 6-1. The following diagram shows the character string XYZ.



ZK-4202-85

Data Type	Definition
complex_number	One of the VAX standard complex floating-point data types. The three complex floating-point numbers are: F_floating complex, D_floating complex, and G_floating complex.

Table A–1 (Cont.) VMS Data Types

Data Type	Definition
	An F_floating complex number (r,i) is comprised of two F_floating point numbers. The first F_floating point number is the real part (r) of the complex number; the second F_floating point number is the imaginary part (i). The structure of an F_floating complex number is as follows:

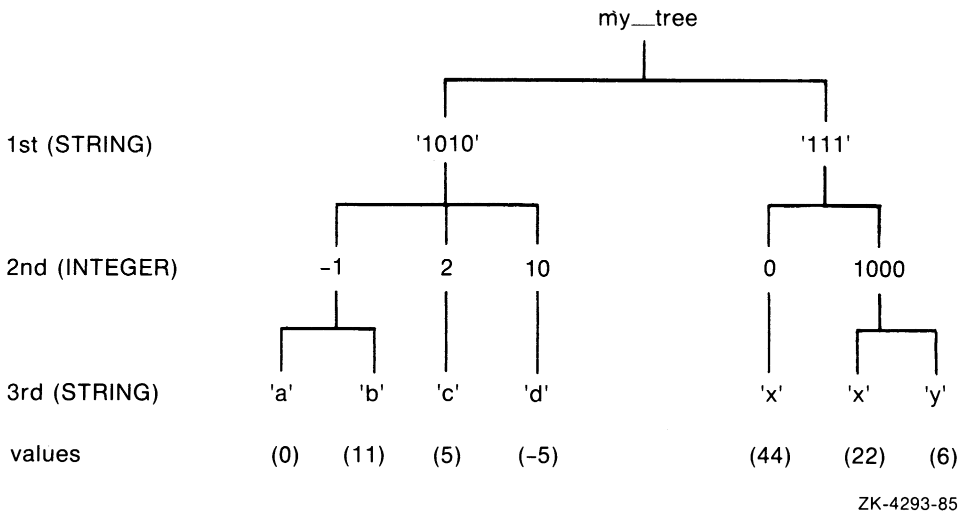
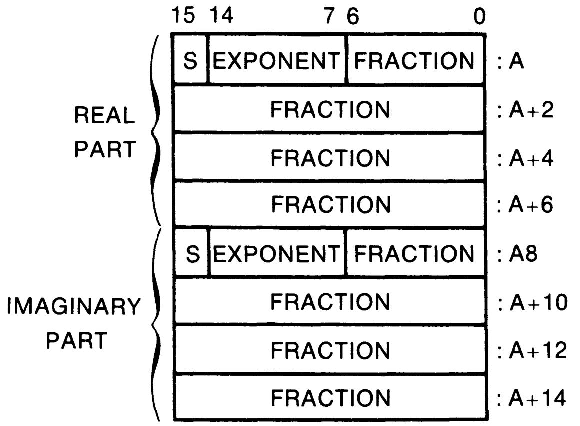


Table A–1 (Cont.) VMS Data Types

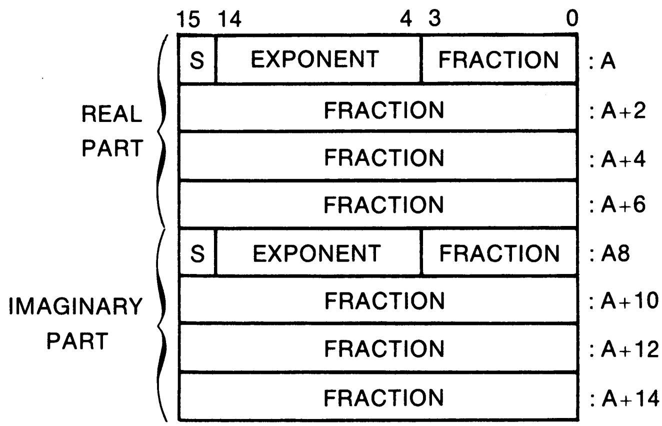
Data Type	Definition
	A D_floating complex number (r,i) is comprised of two D_floating point numbers. The first D_floating point number is the real part (r) of the complex number; the second D_floating point number is the imaginary part (i). The structure of a D_floating complex number is as follows:



ZK-4201-85

Table A-1 (Cont.) VMS Data Types

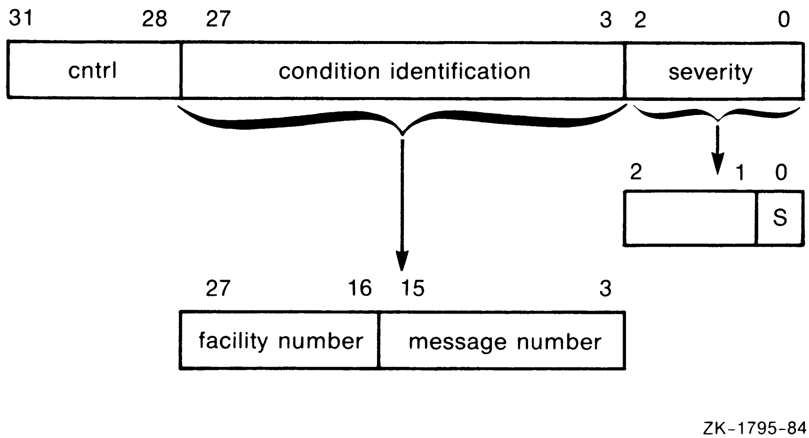
Data Type	Definition
	A G_floating complex number (r,i) is comprised of two G_floating point numbers. The first G_floating point number is the real part (r) of the complex number; the second G_floating point number is the imaginary part (i). The structure of a G_floating complex number is as follows:



ZK-4200-85

Table A-1 (Cont.) VMS Data Types

Data Type	Definition
cond_value	Unsigned longword integer denoting a condition value (that is, a return status or system condition code), which is typically returned by a procedure in R0. The structure of a condition value is as follows:



Depending on your specific needs, you can test just the low-order bit, the low-order three bits, or the entire value.

- The low-order bit indicates successful (1) or unsuccessful (0) completion of the service.
- The low-order three bits, taken together, represent the severity of the error.
- The remaining bits <31:3> classify the particular return condition and the operating system component that issued the condition value.

Each numeric condition value has a unique symbolic name in the following format, where code is a mnemonic describing the return condition.

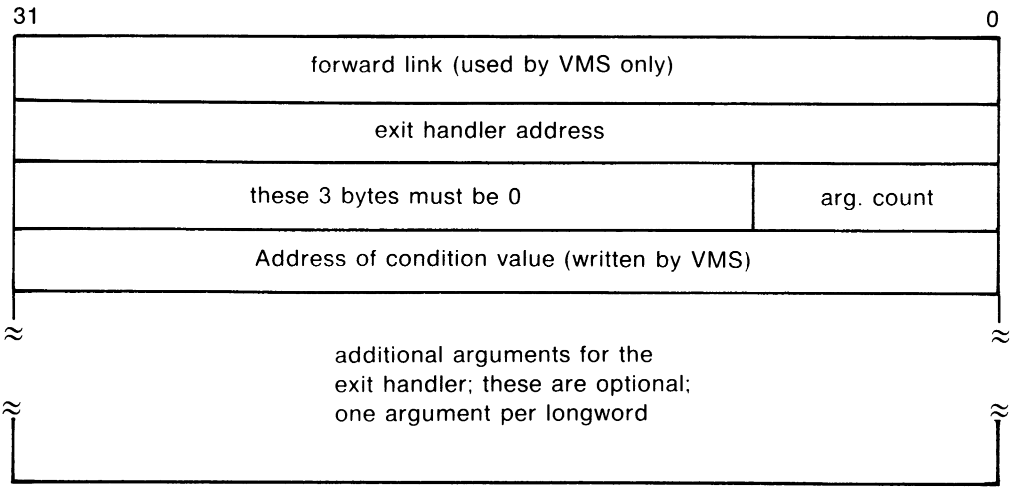
SS$_code

Table A-1 (Cont.) VMS Data Types

Data Type	Definition
context	Unsigned longword that is used by a called procedure to maintain position over an iterative sequence of calls. It is usually initialized by the caller, but thereafter manipulated by the called procedure.
date_time	64-bit unsigned, binary integer denoting a date and time as the number of elapsed 100-nanosecond units since 00:00 o'clock, November 17, 1858. This VMS data type is the same as the data type **absolute date and time** in Table 6-1.
device_name	Character string denoting the 1- to 15-character name of a device. It can be a logical name, but if it is, it must translate to a valid device name. If the device name contains a colon (:), the colon and the characters past it are ignored. When an underscore (_) precedes device name string, it indicates that the string is a physical device name.
ef_cluster_name	Character string denoting the 1- to 15-character name of an event flag cluster. It can be a logical name, but if it is, it must translate to a valid event flag cluster name.
ef_number	Unsigned longword integer denoting the number of an event flag. Local event flags numbered 32 to 63 are available to your programs.

Table A-1 (Cont.) VMS Data Types

Data Type	Definition
exit_handler_block	Variable-length structure denoting an exit handler control block. This control block, which describes the exit handler, is depicted in the following diagram.

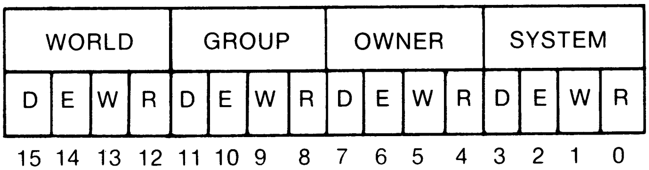


ZK-1714-84

Data Type	Definition
fab	Structure denoting an RMS file access block.
file_protection	Unsigned word that is a 16-bit mask that specifies file protection. The mask contains four 4-bit fields, each of which specifies the protection to be applied to file access attempts by one of the four categories of user: from the rightmost field to the leftmost field, (1) system users, (2) the file owner, (3) users in the same UIC group as the owner, and (4) all other users (the world). Each field specifies, from the rightmost bit to the leftmost bit: (1) read access, (2) write access, (3) execute access, (4) delete access. Set bits indicate that access is denied.

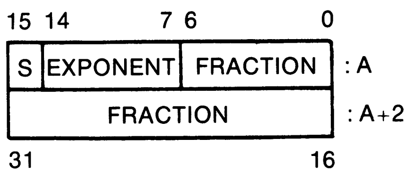
Table A-1 (Cont.) VMS Data Types

Data Type	Definition
	The following diagram depicts the 16-bit file-protection mask.



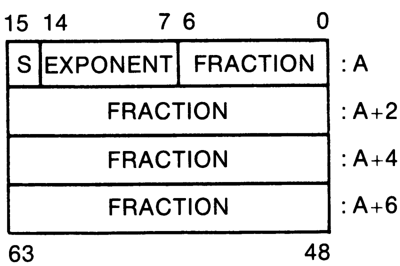
ZK-1706-84

Data Type	Definition
floating_point	One of the VAX standard floating-point data types. These types are F_floating, D_floating, G_floating, and H_floating. The structure of an F_floating number is as follows:



ZK-4197-85

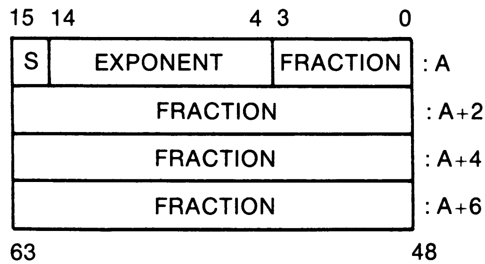
The structure of a D_floating number is as follows:



ZK-4198-85

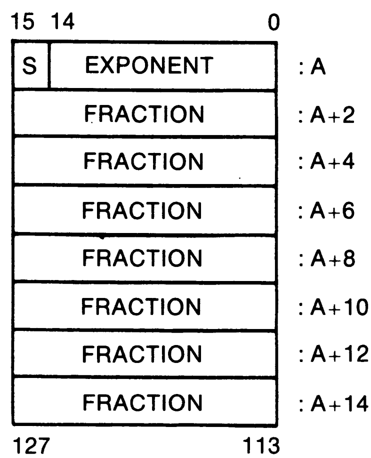
Table A-1 (Cont.) VMS Data Types

Data Type	Definition
	The structure of a G_floating number is as follows:



ZK-4199-85

The structure of an H_floating number is as follows:

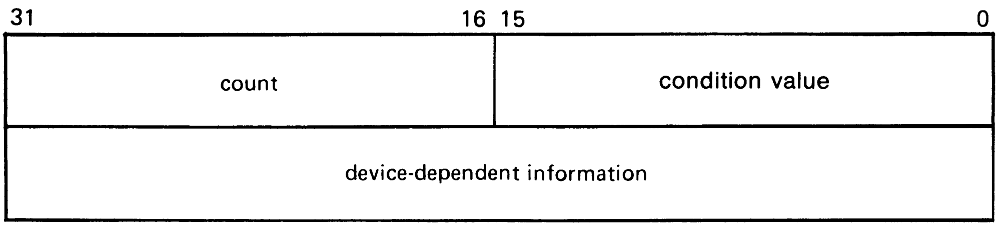


ZK-4196-85

function_code	Unsigned longword specifying the exact operations a procedure is to perform. This longword has two word-length fields: the first field is a number specifying the major operation; the second field is a mask or bit vector specifying various suboperations within the major operation.

Table A–1 (Cont.) VMS Data Types

Data Type	Definition
identifier	Unsigned longword that identifies an object returned by the system.
io_status_block	Quadword structure containing information returned by a procedure that completes asynchronously. The information returned varies depending on the procedure. The following figure illustrates the format of the information written in the IOSB for SYS$QIO.



ZK-856-82

The first word contains a condition value indicating the success or failure of the operation. The condition values used are the same as for all returns from system services; for example, SS$_NORMAL indicates successful completion.

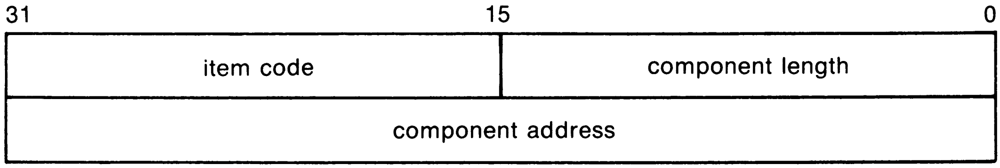
The second word contains the number of bytes actually transferred in the I/O operation. Note that for some devices this word contains only the low-order word of the count.

The second longword contains device-dependent return information.

To ensure successful I/O completion and the integrity of data transfers, the IOSB should be checked following I/O requests, particularly for device-dependent I/O functions.

Table A-1 (Cont.) VMS Data Types

Data Type	Definition
item_list_2	Structure that consists of one or more item descriptors and that is terminated by a longword containing *0*. Each item descriptor is a 2-longword structure that contains three fields. The following diagram depicts a single item descriptor:



ZK-1709-84

The first field is a word in which the service writes the length (in characters) of the requested component. If the service does not locate the component, it returns the value *0* in this field and in the component address field.

The second field contains a user-supplied, word-length symbolic code that specifies the component desired. The item codes are defined by the macros that are specific to the service.

The third field is a longword in which the service writes the starting address of the component. This address is within the input string itself.

Table A-1 (Cont.) VMS Data Types

Data Type	Definition
item_list_3	Structure that consists of one or more item descriptors and that is terminated by a longword containing *0*. Each item descriptor is a 3-longword structure that contains four fields. The following diagram depicts the format of a single item descriptor.

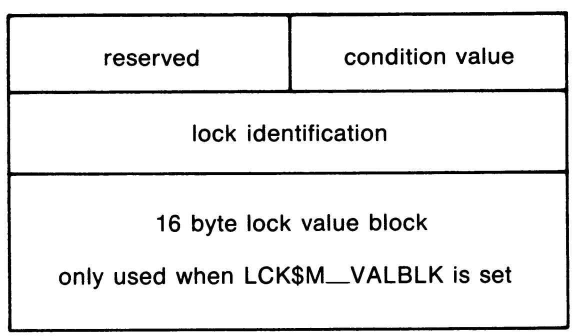
31	15	0
item code		buffer length
buffer address		
return length address		

ZK-1705-84

Data Type	Definition
	The first field is a word containing a user-supplied integer specifying the length (in bytes) of the buffer in which the service writes the information. The length of the buffer needed depends upon the item code specified in the item code field of the item descriptor. If the value of buffer length is too small, the service truncates the data.
	The second field is a word containing a user-supplied symbolic code specifying the item of information that the service is to return. These codes are defined by macros that are specific to the service.
	The third field is a longword containing the user-supplied address of the buffer in which the service writes the information.
	The fourth field is a longword containing the user-supplied address of a word in which the service writes the length in bytes of the information it actually returned.
item_list_pair	Structure that consists of one or more longword pairs, or *doublets* and is terminated by a longword containing *0*. Typically, the first longword contains an integer value such as a code. The second longword can contain a real or integer value.

Table A-1 (Cont.) VMS Data Types

Data Type	Definition
item_quota_list	Structure that consists of one or more quota descriptors and that is terminated by a byte containing a value defined by the symbolic name PQL$_LISTEND. Each quota descriptor consists of a 1-byte quota name followed by an unsigned longword containing the value for that quota.
lock_id	Unsigned longword integer denoting a lock identifier. This lock identifier is assigned by the lock manager facility to a lock when the lock is granted.
lock_status_block	Structure into which the lock manager facility writes status information about a lock. A lock status block always contains at least two longwords: the first word of the first longword contains a condition value; the second word of the first longword is reserved to DIGITAL; and the second longword contains the lock identifier. The lock status block receives the final condition value and the lock identification, and optionally contains a lock value block. When a request is queued, the lock identification is stored in the lock status block even if the lock has not been granted. This allows a procedure to dequeue locks that have not been granted. The condition value is placed in the lock status block only when the lock is granted (or when errors occur in granting the lock). The following diagram depicts a lock status block that includes the optional 16-byte lock value block.



ZK-376-81

Table A-1 (Cont.) VMS Data Types

Data Type	Definition
lock_value_block	16-byte block that the lock manager facility includes in a lock status block if the user requests it. The contents of the lock value block are user-defined and are not interpreted by the lock manager facility.
logical_name	Character string of from 1 to 255 characters that identifies a logical name or equivalence name to be manipulated by VMS logical name system services. Logical names that denote specific VMS objects have their own VMS types: for example, a logical name identifying a device has the VMS type **device_name**.
longword_signed	This VMS data type is the same as the data type **longword integer (signed)** in Table 6-1.
longword_unsigned	This VMS data type is the same as the data type **longword (unsigned)** in Table 6-1.
mask_byte	Unsigned byte wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of flags or as a bit mask.
mask_longword	Unsigned longword wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of flags or as a bit mask.
mask_quadword	Unsigned quadword wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of flags or as a bit mask.
mask_word	Unsigned word wherein each bit is interpreted by the called procedure. A mask is also referred to as a set of flags or bit mask.
null_arg	Unsigned longword denoting a null argument. A *null argument* is an argument whose only purpose is to hold a place in the argument list.
octaword_signed	This VMS data type is the same as the data type **octaword integer (signed)** in Table 6-1.
octaword_unsigned	This VMS data type is the same as the data type **octaword (unsigned)** in Table 6-1.
page_protection	Unsigned longword specifying page protection to be applied by the VAX hardware. Protection values are specified using bits <3:0> ; bits <31:4> are ignored.

Table A-1 (Cont.) VMS Data Types

Data Type	Definition
	The $PRTDEF macro defines the following symbolic names for the protection codes:

Symbol	Description
PRT$C_NA	No access
PRT$C_KR	Kernel read only
PRT$C_KW	Kernel write
PRT$C_ER	Executive read only
PRT$C_EW	Executive write
PRT$C_SR	Supervisor read only
PRT$C_SW	Supervisor write
PRT$C_UR	User read only
PRT$C_UW	User write
PRT$C_ERKW	Executive read; kernel write
PRT$C_SRKW	Supervisor read; kernel write
PRT$C_SREW	Supervisor read; executive write
PRT$C_URKW	User read; kernel write
PRT$C_UREW	User read; executive write
PRT$C_URSW	User read; supervisor write

Data Type	Definition
	If the protection is specified as *0*, the protection defaults to kernel read only.
procedure	Unsigned longword denoting the entry mask to a procedure that is not to be called at AST level. (Arguments specifying procedures to be called at AST level have the VMS type **ast_procedure**.)
process_id	Unsigned longword integer denoting a process identifier (PID). This process identifier is assigned by VMS to a process when the process is created.
process_name	Character string, containing 1 to 15 characters, that specifies the name of a process.
quadword_signed	This VMS data type is the same as the data type **quadword integer (signed)** in Table 6-1.
quadword_unsigned	This VMS data type is the same as the data type **quadword (unsigned)** in Table 6-1.

Table A-1 (Cont.) VMS Data Types

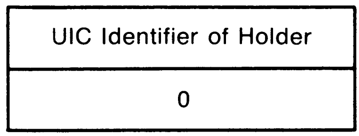
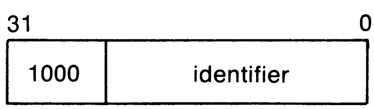
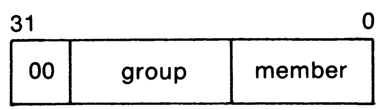
Data Type	Definition
rights_holder	Unsigned quadword specifying a user's access rights to a system object. This quadword consists of two fields: the first is an unsigned longword identifier (VMS type **rights_id**) and the second is a longword bit mask wherein each bit specifies an access right. The following diagram depicts the format of a rights holder.  ZK-1903-84
rights_id	Unsigned longword denoting a rights identifier, which identifies an interest group in the context of the VMS security environment. This rights environment may consist of all or part of a user's user identification code (UIC). Identifiers have two formats in the rights database: UIC format (VMS type **uic**) and ID format. The high order bits of the identifier value specify the format of the identifier. Two high order zero bits identify a UIC format identifier; bit <31> , set to *1*, identifies an ID format identifier. Bit <31> , set to *1*, specifies ID format. Bits <30:28> are reserved by DIGITAL. The remaining bits specify the identifier value. The following diagram depicts the ID format of a rights identifier.  ID Format ZK-1906-84

Table A-1 (Cont.) VMS Data Types

Data Type	Definition
	To the system, an identifier is a binary value; however, to make identifiers easy to use, the system translates the binary identifier value into an identifier name. The binary value and the identifier name are associated in the rights database. An identifier name consists of 1-31 alphanumeric characters and contains at least one nonnumeric character. An identifier name cannot consist entirely of numeric characters. It can include the characters A through Z, dollar signs ($) and underscores (_), as well as the numbers 0 through 9. Any lowercase characters are automatically converted to uppercase.
rab	Structure denoting an RMS record access block.
section_id	Unsigned quadword denoting a global section identifier. This identifier specifies the version of a global section and the criteria to be used in matching that global section.
section_name	Character string denoting a 1 to 43-character global-section name. This character string can be a logical name, but it must translate to a valid global-section name.
system_access_id	Unsigned quadword that denotes a system identification value that is to be associated with a rights database.
time_name	Character string specifying a time value in VMS format.
uic	Unsigned longword denoting a user identification code (UIC). Each UIC is unique and represents a system user. The UIC identifier contains two high order bits that designate format, a member field, and a group field. Member numbers range from 0 to 65,534; group numbers range from 1 to 16,382. The following diagram depicts the UIC format.



UIC Format

ZK-1905-84

Table A-1 (Cont.) VMS Data Types

Data Type	Definition
user_arg	Unsigned longword denoting a user-defined argument. This longword is passed to a procedure as an argument, but the contents of the longword are defined and interpreted by the user.
varying_arg	Unsigned longword denoting a variable argument. A variable argument can have variable types, depending on specifications made for other arguments in the call.
vector_byte_signed	A homogeneous array whose elements are all signed bytes.
vector_byte_unsigned	A homogeneous array whose elements are all unsigned bytes.
vector_longword_signed	A homogeneous array whose elements are all signed longwords.
vector_longword_unsigned	A homogeneous array whose elements are all unsigned longwords.
vector_quadword_signed	A homogeneous array whose elements are all signed quadwords.
vector_quadword_unsigned	A homogeneous array whose elements are all unsigned quadwords.
vector_word_signed	A homogeneous array whose elements are all signed words.
vector_word_unsigned	A homogeneous array whose elements are all unsigned words.
word_signed	This VMS data type is the same as the data type **word integer (signed)** in Table 6-1.
word_unsigned	This VMS data type is the same as the data type **word (unsigned)** in Table 6-1.

A.2 VAX BLISS Implementation

The following table lists VMS data types and their corresponding VAX BLISS data type declarations.

Table A–2 VAX BLISS Implementation

VMS Data Type	VAX BLISS Declaration
access_bit_names	BLOCKVECTOR[32,8,BYTE]
access_mode	UNSIGNED BYTE
address	UNSIGNED LONG
address_range	VECTOR[2,LONG,UNSIGNED]
arg_list	VECTOR[n,LONG,UNSIGNED] where *n* is the number of arguments + 1
ast_procedure	UNSIGNED LONG
boolean	UNSIGNED LONG
byte_signed	SIGNED BYTE
byte_unsigned	UNSIGNED BYTE
channel	UNSIGNED WORD
char_string	VECTOR[65536,BYTE,UNSIGNED]
complex_number	F_Complex: VECTOR[2,LONG] D_Complex: VECTOR[4,LONG] G_Complex: VECTOR[4,LONG] H_Complex: VECTOR[8,LONG]
cond_value	UNSIGNED LONG
context	UNSIGNED LONG
date_time	VECTOR[2,LONG,UNSIGNED]
device_name	VECTOR[n,BYTE,UNSIGNED] where n is the length of the device name
ef_cluster_name	VECTOR[n,BYTE,UNSIGNED] where n is the length of the event flag cluster name
ef_number	UNSIGNED LONG
exit_handler_block	BLOCK[n,BYTE] where n is the size of the exit handler control block
fab	$FAB_DECL (from STARLET.REQ)
file_protection	BLOCK[2,BYTE]

Table A-2 (Cont.) VAX BLISS Implementation

VMS Data Type	VAX BLISS Declaration
floating_point	F_Floating: VECTOR[1,LONG] D_Floating: VECTOR[2,LONG] G_Floating: VECTOR[2,LONG] H_Floating: VECTOR[4,LONG]
function_code	BLOCK[2,WORD]
identifier	UNSIGNED LONG
io_status_block	BLOCK[8,BYTE]
item_list_2	BLOCKVECTOR[n,8,BYTE] where n is the number of the item descriptors + 1
item_list_3	BLOCKVECTOR[n,12,BYTE] where n is the number of the item descriptors + 1 $ITMLST_DECL/$ITMLST_INIT from STARLET.REQ
item_list_pair	BLOCKVECTOR[n,2,LONG] where n is the number of the item descriptors + 1
item_quota_list	BLOCKVECTOR[n,5,BYTE] where n is the number of the quota descriptors + 1
lock_id	UNSIGNED_LONG
lock_status_block	BLOCK[n,BYTE] where n is the size of the lock_status_block -at least 8
lock_value_block	BLOCK[16,BYTE]
logical_name	VECTOR[255,BYTE,UNSIGNED]
longword_signed	SIGNED LONG
longword_unsigned	UNSIGNED LONG
mask_byte	BITVECTOR[8]
mask_longword	BITVECTOR[32]
mask_quadword	BITVECTOR[64]
mask_word	BITVECTOR[16]
null_arg	UNSIGNED LONG
octaword_signed	VECTOR[4,LONG,UNSIGNED]
octaword_unsigned	VECTOR[4,LONG,UNSIGNED]
page_protection	UNSIGNED LONG
procedure	UNSIGNED LONG
process_id	UNSIGNED LONG

Table A-2 (Cont.) VAX BLISS Implementation

VMS Data Type	VAX BLISS Declaration
process_name	VECTOR[n,BYTE,UNSIGNED] where n is the length of the process name
quadword_signed	VECTOR[2,LONG,UNSIGNED]
quadword_unsigned	VECTOR[2,LONG,UNSIGNED]
rights_holder	BLOCK[8,BYTE]
rights_id	UNSIGNED LONG
rab	$RAB_DECL from STARLET.REQ
section_id	VECTOR[2,LONG,UNSIGNED]
section_name	VECTOR[n,BYTE,UNSIGNED] where n is the length of the global section name
system_access_id	VECTOR[2,LONG,UNSIGNED]
time_name	VECTOR[n,BYTE,UNSIGNED] where n is the length of the time value in VMS format
uic	UNSIGNED LONG
user_arg	UNSIGNED LONG
varying_arg	UNSIGNED LONG
vector_byte_signed	VECTOR[n,BYTE,SIGNED] where n is the size of the array
vector_byte_unsigned	VECTOR[n,BYTE,UNSIGNED] where n is the size of the array
vector_longword_signed	VECTOR[n,LONG,SIGNED] where n is the size of the array
vector_longword_unsigned	VECTOR[n,LONG,UNSIGNED] where n is the size of the array
vector_quadword_signed	BLOCKVECTOR[n,2,LONG] where n is the size of the array
vector_quadword_unsigned	BLOCKVECTOR[n,2,LONG] where n is the size of the array
vector_word_signed	VECTOR[n,BYTE,SIGNED] where n is the size of the array
vector_word_unsigned	VECTOR[n,BYTE,UNSIGNED] where n is the size of the array
word_signed	SIGNED WORD
word_unsigned	UNSIGNED WORD

A.3 VAX C Implementation

The following table lists VMS data types and their corresponding VAX C data type declarations.

Table A-3 VAX C Implementation

VMS Data Type	VAX C Declaration
access_bit_names	User-defined [1]
access_mode	unsigned char
address	int *pointer [2,4]
address_range	int *array [2] [2,3,4]
arg_list	User-defined [1]
ast_procedure	Pointer to function. [2]
boolean	unsigned long int
byte_signed	char
byte_unsigned	unsigned char
channel	unsigned short int
char_string	char array[n] [3,5]
complex_number	User-defined [1]
cond_value	unsigned long int
context	unsigned long int
date_time	User-defined [1]
device_name	char array[n] [3,5]
ef_cluster_name	char array[n] [3,5]
ef_number	unsigned long int
exit_handler_block	User-defined [1]
fab	#include fab from text library struct FAB
file_protection	unsigned short int, or User-defined [1]
floating_point	float or double

[1]The declaration of a user-defined data structure depends on how the data will be used. Such data structures can be declared in a variety of ways, each of which is more suitable to specific applications.

[2]The term *pointer* refers to several declarations involving pointers. Pointers are declared with special syntax and associated with the data type of the object being pointed to. This object is often *user-defined.*

[3]The term *array* denotes the syntax of a VAX C array declaration.

[4]The data type specified can be changed to any valid VAX C data type.

[5]The size of the array must be substituted for n.

Table A-3 (Cont.) VAX C Implementation

VMS Data Type	VAX C Declaration
function_code	Unsigned long int or User-defined [1]
identifier	int *pointer [2,4]
io_status_block	User-defined [1]
item_list_2	User-defined [1]
item_list_3	User-defined [1]
item_list_pair	User-defined [1]
item_quota_list	User-defined [1]
lock_id	unsigned long int
lock_status_block	User-defined [1]
lock_value_block	User-defined [1]
logical_name	char array[n] [3,5]
longword_signed	long int
longword_unsigned	unsigned long int
mask_byte	unsigned char
mask_longword	unsigned long int
mask_quadword	User-defined [1]
mask_word	unsigned short int
null_arg	unsigned long int
octaword_signed	User-defined [1]
octaword_unsigned	User-defined [1]
page_protection	unsigned long int
procedure	Pointer to function [2]
process_id	unsigned long int
process_name	char array[n] [3,5]
quadword_signed	User-defined [1]
quadword_unsigned	User-defined [1]
rights_holder	User-defined [1]

[1] The declaration of a user-defined data structure depends on how the data will be used. Such data structures can be declared in a variety of ways, each of which is more suitable to specific applications.

[2] The term *pointer* refers to several declarations involving pointers. Pointers are declared with special syntax and associated with the data type of the object being pointed to. This object is often *user-defined*.

[3] The term *array* denotes the syntax of a VAX C array declaration.

[4] The data type specified can be changed to any valid VAX C data type.

[5] The size of the array must be substituted for n.

Table A-3 (Cont.) VAX C Implementation

VMS Data Type	VAX C Declaration
rights_id	unsigned long int
rab	#include rab from text library struct RAB
section_id	User-defined [1]
section_name	char array[n] [3,5]
system_access_id	User-defined [1]
time_name	char array[n] [3,5]
uic	unsigned long int
user_arg	User-defined [1]
varying_arg	User-defined [1]
vector_byte_signed	char array[n] [3,5]
vector_byte_unsigned	unsigned char array[n] [3,5]
vector_longword_signed	long int array[n] [3,5]
vector_longword_unsigned	unsigned long int array[n] [3,5]
vector_quadword_signed	User-defined [1]
vector_quadword_unsigned	User-defined [1]
vector_word_signed	short int array[n] [3,5]
vector_word_unsigned	unsigned short int array[n] [3,5]
word_signed	short int
word_unsigned	unsigned short int

[1] The declaration of a user-defined data structure depends on how the data will be used. Such data structures can be declared in a variety of ways, each of which is more suitable to specific applications.

[3] The term *array* denotes the syntax of a VAX C array declaration.

[5] The size of the array must be substituted for n.

A.4 VAX FORTRAN Implementation

The following table lists VMS data types and their corresponding VAX FORTRAN data type declarations.

Table A–4 VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration

access_bit_names	INTEGER*4(2,32)
	or
	STRUCTURE /access_bit_names/
	    INTEGER*4 access_name_len
	    INTEGER*4 access_name_buf
	END STRUCTURE !access_bit_names
	RECORD /access_bit_names/ my_names(32)

access_mode	BYTE
address	INTEGER*4

address_range	INTEGER*4(2)
	or
	STRUCTURE /address_range/
	    INTEGER*4 low_address
	    INTEGER*4 high_address
	END STRUCTURE

arg_list	INTEGER*4(n)
ast_procedure	EXTERNAL
boolean	LOGICAL*4
byte_signed	BYTE
byte_unsigned	BYTE [1]
channel	INTEGER*2
char_string	CHARACTER*n
complex_number	COMPLEX*8 COMPLEX*16
cond_value	INTEGER*4
context	INTEGER*4
date_time	INTEGER*4(2)
device_name	CHARACTER*n
ef_cluster_name	CHARACTER*n
ef_number	INTEGER*4

[1] Unsigned data types are not directly supported by VAX FORTRAN. However, in most cases you can substitute the signed equivalent so long as you do not exceed the range of the signed data structure.

Table A-4 (Cont.) VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
exit_handler_block	STRUCTURE /exhblock/ INTEGER*4 flink INTEGER*4 exit_handler_addr BYTE(3) /0/ BYTE arg_count INTEGER*4 cond_value ! . ! . (optional arguments ... ! . one argument per longword) ! END STRUCTURE !cntrlblk RECORD /exhblock/ myexh_block
fab	INCLUDE '($FABDEF)' RECORD /fabdef/ myfab
file_protection	INTEGER*4
floating_point	REAL*4 REAL*8 DOUBLE PRECISION REAL*16
function_code	INTEGER*4
identifier	INTEGER*4
io_status_block	STRUCTURE /iosb/ INTEGER*2 iostat, !return status 2 term_offset, !Loc. of line terminator 2 terminator, !value of terminator 2 term_size !size of terminator END STRUCTURE RECORD /iosb/ my_iosb

Table A-4 (Cont.) VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
item_list_2	STRUCTURE /itmlst/ UNION MAP INTEGER*2 buflen,code INTEGER*4 bufadr END MAP MAP INTEGER*4 end_list /0/ END MAP END UNION END STRUCTURE !itmlst RECORD /itmlst/ my_itmlst_2(n) (Allocate n records where n is the number item codes plus an extra element for the end-of-list item)
item_list_3	STRUCTURE /itmlst/ UNION MAP INTEGER*2 buflen,code INTEGER*4 bufadr,retlenadr END MAP MAP INTEGER*4 end_list /0/ END MAP END UNION END STRUCTURE !itmlst RECORD /itmlst/ my_itmlst_2(n)

Table A–4 (Cont.) VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
item_list_pair	STRUCTURE /itmlist_pair/ UNION MAP INTEGER*4 code INTEGER*4 value END MAP MAP INTEGER*4 end_list /0/ END MAP END UNION END STRUCTURE !itmlst_pair RECORD /itmlst_pair/ my_itmlst_pair(n) (Allocate n records where n is the number item codes plus an extra element for the end-of-list item)
item_quota_list	STRUCTURE /item_quota_list/ MAP BYTE quota_name INTEGER*4 quota_value END MAP MAP BYTE end_quota_list END MAP END STRUCTURE !item_quota_list
lock_id	INTEGER*4
lock_status_block	STRUCTURE/lksb/ INTEGER*2 cond_value INTEGER*2 unused INTEGER*4 lock_id BYTE(16) END STRUCTURE !lock_status_lock
lock_value_block	BYTE(16)
logical_name	CHARACTER*n
longword_signed	INTEGER*4
longword_unsigned	INTEGER*4 [1]
mask_byte	INTEGER*1
mask_longword	INTEGER*4

[1]Unsigned data types are not directly supported by VAX FORTRAN. However, in most cases you can substitute the signed equivalent so long as you do not exceed the range of the signed data structure.

Table A–4 (Cont.) VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
mask_quadword	INTEGER*4(2)
mask_word	INTEGER*2
null_arg	%VAL(0)
octaword_signed	INTEGER*4(4)
octaword_unsigned	INTEGER*4(4) [1]
page_protection	INTEGER*4
procedure	INTEGER*4
process_id	INTEGER*4
process_name	CHARACTER*n
quadword_signed	INTEGER*4(2)
quadword_unsigned	INTEGER*4(2) [1]

rights_holder	INTEGER*4(2)
	or
	STRUCTURE /rights_holder/
	    INTEGER*4 rights_id
	    INTEGER*4 rights_mask
	END STRUCTURE !rights_holder

rights_id	INTEGER*4
rab	INCLUDE '($RABDEF)' RECORD /rabdef/ myrab
section_id	INTEGER*4(2)
section_name	CHARACTER*n
system_access_id	INTEGER*4(2)
time_name	CHARACTER*23
uic	INTEGER*4
user_arg	Any longword quantity
varying_arg	INTEGER*4
vector_byte_signed	BYTE(n)
vector_byte_unsigned	BYTE(n) [1]
vector_longword_signed	INTEGER*4(n)
vector_longword_unsigned	INTEGER*4(n) [1]
vector_quadword_signed	INTEGER*4(2, n)
vector_quadword_unsigned	INTEGER*4(2,n) [1]

[1] Unsigned data types are not directly supported by VAX FORTRAN. However, in most cases you can substitute the signed equivalent so long as you do not exceed the range of the signed data structure.

Table A-4 (Cont.) VAX FORTRAN Implementation

VMS Data Type	VAX FORTRAN Declaration
vector_word_signed	INTEGER*2(n)
vector_word_unsigned	INTEGER*2(n) [1]
word_signed	INTEGER*2(n)
word_unsigned	INTEGER*2(n) [1]

[1] Unsigned data types are not directly supported by VAX FORTRAN. However, in most cases you can substitute the signed equivalent so long as you do not exceed the range of the signed data structure.

A.5 VAX MACRO Implementation

The following table lists VMS data types and their corresponding VAX MACRO data type declarations.

Table A-5 VAX MACRO Implementation

VMS Data Type	VAX MACRO Declaration
access_bit_names	.ASCID /name_for_bit0/ .ASCID /name_for_bit1/ ... .ASCID /name_for_bit31/
access_mode	.BYTE PSL$C_xxxx
address	.ADDRESSS virtual_address
address_range	.ADDRESS start_address,end_address
arg_list	.LONG n_args, arg1, arg2,...
ast_procedure	.ADDRESS ast_procedure
boolean	.LONG 1 or .LONG 0
byte_signed	.SIGNED_BYTE byte_value
byte_unsigned	.BYTE byte_value
channel	.WORD channel_number
char_string	.ASCID /string/
complex_number	NA
cond_value	.LONG cond_value
context	.LONG 0
date_time	.QUAD date_time

Table A–5 (Cont.) VAX MACRO Implementation

VMS Data Type	VAX MACRO Declaration
device_name	.ASCID /ddcu:/
ef_cluster_name	.ASCID /ef_cluster_name/
ef_number	.LONG ef_number
exit_handler_block	.LONG 0 .ADDRESS exit_handler_routine .LONG 1 .ADDRESS status STATUS: .BLKL 1
fab	MYFAB: $FAB
file_protection	.WORD prot_value
floating_point	.FLOAT, .G_FLOAT, or .H_FLOAT
function_code	.LONG code!mask
identifier	.ADDRESSS virtual_address
io_status_block	.QUAD 0
item_list_2	.WORD component_length .WORD item_code .ADDRESS component_address
item_list_3	.WORD buffer_length .WORD item_code .ADDRESS buffer_address .ADDRESS return_length_address
item_list_pair	.LONG item_code .LONG data
item_quota_list	.BYTE PQL$_xxxx .LONG value_for_quota .BYTE pql$_listend
lock_id	.LONG lock_id
lock_status_block	.QUAD 0
lock_value_block	.BLKB 16
logical_name	.ASCID /logical_name/
longword_signed	.LONG value
longword_unsigned	.LONG value
mask_byte	.BYTE mask_byte
mask_longword	.LONG mask_longword
mask_quadword	.QUAD mask_quadword
mask_word	.WORD mask_word

Table A-5 (Cont.) VAX MACRO Implementation

VMS Data Type	VAX MACRO Declaration
null_arg	.LONG 0
octaword_signed	NA
octaword_unsigned	.OCTA value
page_protection	.LONG page_protection
procedure	.ADDRESS procedure
process_id	.LONG process_id
process_name	.ASCID /process_name/
quadword_signed	NA
quadword_unsigned	.QUAD value
rights_holder	.LONG identifier, access_right_bitmask
rights_id	.LONG rights_id
rab	MYRAB: $RAB
section_id	.LONG sec$k_matXXX, version_number
section_name	.ASCID /section_name/
system_access_id	.QUAD system_access_id
time_name	.ASCID /dd-mmm-yyyy:hh:mm:ss.cc/
uic	.LONG uic
user_arg	.LONG data
varying_arg	Dependent upon application.
vector_byte_signed	.SIGNED_BYTE val1,val2,...valN
vector_byte_unsigned	.BYTE val1,val2,...valN
vector_longword_signed	.LONG val1,val2,...valN
vector_longword_unsigned	.LONG val1,val2,...valN
vector_quadword_signed	NA
vector_quadword_unsigned	.QUAD val1 .QUAD val2 ... .QUAD valN
vector_word_signed	.SIGNED_WORD val1,val2,...valN
vector_word_unsigned	.WORD val1,val2,...valN
word_signed	.SIGNED_WORD value
word_unsigned	.WORD value

A.6 VAX PASCAL Implementation

The following table lists VMS data types and their corresponding VAX PASCAL data type declarations.

Table A-6 VAX PASCAL Implementation

VMS Data Type	VAX PASCAL Declaration
access_bit_names	PACKED ARRAY [1..32] OF [QUAD] RECORD END; [1,6]
access_mode	[BYTE] 0..3; [6]
address	UNSIGNED;
address_range	PACKED ARRAY [1..2] OF UNSIGNED; [6]
arg_list	PACKED ARRAY [1..n] OF UNSIGNED; [6]
ast_procedure	UNSIGNED;
boolean	BOOLEAN; [3]
byte_signed	[BYTE] -128..127; [6]
byte_unsigned	[BYTE] 0..255; [6]
channel	[WORD] 0..65535; [6]
char_string	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR; [4]
complex_number	[LONG(2)] RECORD END; * F_Floating Complex * [1,6] [QUAD(2)] RECORD END; * D/G_Floating Complex * [OCTA(2)] RECORD END; * H_Floating Complex *
cond_value	UNSIGNED;
context	UNSIGNED;
date_time	[QUAD] RECORD END; [1,6]
device_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR; [4]
ef_cluster_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR; [4]
ef_number	UNSIGNED;
exit_handler_block	PACKED ARRAY [1..n] OF UNSIGNED; [6]

[1] This type is not available in VAX PASCAL and an empty record has been inserted. To manipulate the contents, declare with explicit field components. If you pass an empty record as a parameter to a PASCAL routine, you must use the VAR keyword.

[3] VAX PASCAL allocates a byte for a BOOLEAN variable. Use the [LONG] attribute when passing to routines that expect a longword.

[4] This parameter declaration accepts VARYING OF CHAR or PACKED ARRAY OF CHAR and produces the CLASS_S descriptor required by system services.

[6] VAX PASCAL expects either a type identifier or conformant schema. Declare this under the TYPE declaration and use the type identifier in the formal parameter declaration.

Table A-6 (Cont.) VAX PASCAL Implementation

VMS Data Type	VAX PASCAL Declaration
fab	FAB$TYPE; [5]
file_protection	[WORD] RECORD END; [1,6]
floating_point	REAL; { F_Floating } SINGLE; { F_Floating } DOUBLE; { D_Floating/G_Floating } [2] QUADRUPLE; { H_Floating }
function_code	UNSIGNED;
identifier	UNSIGNED;
io_status_block	[QUAD] RECORD END; [1,6]
item_list_2	PACKED ARRAY [1..n] OF PACKED RECORD [6] CASE INTEGER OF 1: (FIELD1 : [WORD] 0..65535; FIELD2 : [WORD] 0..65535; FIELD3 : UNSIGNED); 2: (TERMINATOR : UNSIGNED); END;
item_list_3	PACKED ARRAY [1..n] OF PACKED RECORD [6] CASE INTEGER OF 1: (FIELD1 : [WORD] 0..65535; FIELD2 : [WORD] 0..65535; FIELD3 : UNSIGNED; FIELD4 : UNSIGNED); 2: (TERMINATOR : UNSIGNED); END;

[1]This type is not available in VAX PASCAL and an empty record has been inserted. To manipulate the contents, declare with explicit field components. If you pass an empty record as a parameter to a PASCAL routine, you must use the VAR keyword.

[2]If the [G_FLOATING] attribute is used in compiling, double-precision variables and expressions are represented in G_floating format. The /G_FLOATING command line qualifier can also be used. Both methods default to no G_floating.

[5]The program must inherit the STARLET environment file located in SYS$LIBRARY:STARLET.PEN.

[6]VAX PASCAL expects either a type identifier or conformant schema. Declare this under the TYPE declaration and use the type identifier in the formal parameter declaration.

Table A-6 (Cont.) VAX PASCAL Implementation

VMS Data Type	VAX PASCAL Declaration
item_list_pair	PACKED ARRAY [1..n] OF PACKED RECORD [6] CASE INTEGER OF 1: (FIELD1 : INTEGER; FIELD2 : INTEGER); 2: (TERMINATOR : UNSIGNED); END;
item_quota_list	PACKED ARRAY [1..n] OF PACKED RECORD [6] CASE INTEGER OF 1: (QUOTA_NAME : [BYTE] 0..255; QUOTA_VALUE: UNSIGNED); 2: (QUOTA_TERM : [BYTE] 0..255); END;
lock_id	UNSIGNED;
lock_status_block	[BYTE(24)] RECORD END; [1,6]
lock_value_block	[BYTE(16)] RECORD END; [1,6]
logical_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR; [4]
longword_signed	INTEGER;
longword_unsigned	UNSIGNED;
mask_byte	[BYTE,UNSAFE] PACKED ARRAY [1..8] OF BOOLEAN; [6]
mask_longword	[LONG,UNSAFE] PACKED ARRAY [1..32] OF BOOLEAN; [6]
mask_quadword	[QUAD,UNSAFE] PACKED ARRAY [1..64] OF BOOLEAN; [6]
mask_word	[WORD,UNSAFE] PACKED ARRAY [1..16] OF BOOLEAN; [6]
null_arg	UNSIGNED;

[1] This type is not available in VAX PASCAL and an empty record has been inserted. To manipulate the contents, declare with explicit field components. If you pass an empty record as a parameter to a PASCAL routine, you must use the VAR keyword.

[4] This parameter declaration accepts VARYING OF CHAR or PACKED ARRAY OF CHAR and produces the CLASS_S descriptor required by system services.

[6] VAX PASCAL expects either a type identifier or conformant schema. Declare this under the TYPE declaration and use the type identifier in the formal parameter declaration.

Table A-6 (Cont.) VAX PASCAL Implementation

VMS Data Type	VAX PASCAL Declaration
octaword_signed	[OCTA] RECORD END; [1,6]
octaword_unsigned	[OCTA] RECORD END; [1,6]
page_protection	[LONG] 0..7; [6]
procedure	UNSIGNED;
process_id	UNSIGNED;
process_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR; [4]
quadword_signed	[QUAD] RECORD END; [1,6]
quadword_unsigned	[QUAD] RECORD END; [1,6]
rights_holder	[QUAD] RECORD END; [1,6]
rights_id	UNSIGNED;
rab	RAB$TYPE; [5]
section_id	[QUAD] RECORD END; [1,6]
section_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR; [4]
system_access_id	[QUAD] RECORD END; [1,6]
time_name	[CLASS_S] PACKED ARRAY [L..U:INTEGER] OF CHAR; [4]
uic	UNSIGNED;
user_arg	[UNSAFE] UNSIGNED;
varying_arg	[UNSAFE,REFERENCE] PACKED ARRAY [L..U:INTEGER] OF [BYTE] 0..255;
vector_byte_signed	PACKED ARRAY [1..n] OF [BYTE] -128..127; [6]
vector_byte_unsigned	PACKED ARRAY [1..n] OF [BYTE] 0..255; [6]
vector_longword_signed	PACKED ARRAY [1..n] OF INTEGER; [6]
vector_longword_unsigned	PACKED ARRAY [1..n] OF UNSIGNED; [6]
vector_quadword_signed	PACKED ARRAY [1..n] OF [QUAD] RECORD END; [1,6]
vector_quadword_unsigned	PACKED ARRAY [1..n] OF [QUAD] RECORD END; [1,6]

[1] This type is not available in VAX PASCAL and an empty record has been inserted. To manipulate the contents, declare with explicit field components. If you pass an empty record as a parameter to a PASCAL routine, you must use the VAR keyword.

[4] This parameter declaration accepts VARYING OF CHAR or PACKED ARRAY OF CHAR and produces the CLASS_S descriptor required by system services.

[5] The program must inherit the STARLET environment file located in SYS$LIBRARY:STARLET.PEN.

[6] VAX PASCAL expects either a type identifier or conformant schema. Declare this under the TYPE declaration and use the type identifier in the formal parameter declaration.

Table A-6 (Cont.) VAX PASCAL Implementation

VMS Data Type	VAX PASCAL Declaration
vector_word_signed	PACKED ARRAY [1..n] OF [WORD] -32768..32767; [6]
vector_word_unsigned	PACKED ARRAY [1..n] OF [WORD] 0..65535; [6]
word_signed	[WORD] -32768..32767; [6]
word_unsigned	[WORD] 0..65535; [6]

[6]VAX PASCAL expects either a type identifier or conformant schema. Declare this under the TYPE declaration and use the type identifier in the formal parameter declaration.

A.7 VAX PL/I Implementation

The following table lists VMS data types and their corresponding VAX PL/I data type declarations.

Table A-7 VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
access_bit_names	1 ACCESS_BIT_NAMES(32), 2 LENGTH FIXED BINARY(15), 2 DTYPE FIXED BINARY(7) INITIAL((32)DSC$K_DTYPE_T), 2 CLASS FIXED BINARY(7) INITIAL((32)DSC$K_CLASS_S), 2 CHAR_PTR POINTER; [6] The length of the LENGTH field in each element of the array should correspond to the length of a string of characters pointed to by the CHAR_PTR field. The constants DST$K_CLASS_S and DST$K_DTYPE_T can be used by including the module $DSCDEF from PLISTARLET or by declaring it GLOBALREF FIXED BINARY(31) VALUE.

[6]Routines declared in PLISTARLET often use ANY so the user is free to declare the data structure in the most convenient way for her application. ANY may be necessary in some cases since PL/I does not allow parameters declarations for some data types used by VMS. (In particular, PL/I parameters with arrays passed by reference may not be declared to have nonconstant bounds.)

Table A-7 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
access_mode	FIXED BINARY(7) (The constants for this type— PSLC_KERNEL, PSLC_EXEC, PSLC_SUPER, PSLC_USER—are declared in module $PSLDEF in PLISTARLET.)[1]
address	POINTER
address_range	(2) POINTER [6]
arg_list	1 ARG_LIST BASED, 2 ARGCOUNT FIXED BINARY(31), 2 ARGUMENT (X REFER (ARGCOUNT)) POINTER; [6] If the arguments are passed by value, it may be appropriate to change the type of the ARGUMENT field of the structure. Alternatively, you can use the POSINT, INT, or UNSPEC built-in functions/pseudovariables to access the data. X should be an expression with a value in the range 0-255 at the time the structure is allocated.
ast_procedure	PROCEDURE or ENTRY [2]
boolean	BIT ALIGNED [1]
byte_signed	FIXED BINARY(7)
byte_unsigned	FIXED BINARY(7) [3]
channel	FIXED BINARY(15)
char_string	CHARACTER(n) [4]

[1]System routines are often written so the parameter passed occupies more storage than the object requires. For example, some system services have parameters that return a bit value as a longword. These variables must be declared BIT(32) ALIGNED (not BIT(n) ALIGNED) so adjacent storage is not overwritten by return values or used incorrectly as input. (Longword parameters are always declared BIT(32) ALIGNED.)

[2]AST procedures and those passed as parameters of type ENTRY VALUE or ANY VALUE must be external procedures. This applies to all system routines which take procedure parameters.

[3]This is actually an unsigned integer. This declaration is interpreted as a signed number; use the POSINT function to determine the actual value.

[4]System services require CHARACTER string representation for parameters. Most other system routines allow either CHARACTER or CHARACTER VARYING. For parameter declarations, n should be an asterisk.

[6]Routines declared in PLISTARLET often use ANY so the user is free to declare the data structure in the most convenient way for her application. ANY may be necessary in some cases since PL/I does not allow parameters declarations for some data types used by VMS. (In particular, PL/I parameters with arrays passed by reference may not be declared to have nonconstant bounds.)

Table A-7 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
complex_number	(2) FLOAT BINARY(n) (See floating_point for values of n.)
cond_value	See module STS$VALUE in PLISTARLET [6]
context	FIXED BINARY(31)
date_time	BIT(64) ALIGNED [5]
device_name	CHARACTER(n) [4]
ef_cluster_name	CHARACTER(n) [4]
ef_number	FIXED BINARY(31)
exit_handler_block	1 EXIT_HANDLER_BLOCK BASED, 2 FORWARD_LINK POINTER, 2 HANDLER POINTER, 2 ARGCOUNT FIXED BINARY(31), 2 ARGUMENT (n REFER (ARGCOUNT)) POINTER; [6] Replace n with an expression that will yield a value between 0 and 255 at the time the structure is allocated.
fab	See module $FABDEF in PLISTARLET [6]
file_protection	BIT(16) ALIGNED [1]

[1]System routines are often written so the parameter passed occupies more storage than the object requires. For example, some system services have parameters that return a bit value as a longword. These variables must be declared BIT(32) ALIGNED (not BIT(n) ALIGNED) so adjacent storage is not overwritten by return values or used incorrectly as input. (Longword parameters are always declared BIT(32) ALIGNED.)

[4]System services require CHARACTER string representation for parameters. Most other system routines allow either CHARACTER or CHARACTER VARYING. For parameter declarations, n should be an asterisk.

[5]VAX PL/I does not support FIXED BINARY numbers with precisions greater than 32. To use larger values, declare variables to be BIT variables of the appropriate size and use the POSINT and SUBSTR bits as necessary to access the values, or declare the item as a structure. The RTL routines LIB$ADDX and LIB$SUBX may be useful if you need to perform arithmetic on these types.

[6]Routines declared in PLISTARLET often use ANY so the user is free to declare the data structure in the most convenient way for her application. ANY may be necessary in some cases since PL/I does not allow parameters declarations for some data types used by VMS. (In particular, PL/I parameters with arrays passed by reference may not be declared to have nonconstant bounds.)

Table A-7 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
floating_point	FLOAT BINARY(n) The values for n are as follows: 1 <= n <= 24 - F floating 25 <= n <= 53 - D floating 25 <= n <= 53 - G floating (with /G_FLOAT) 54 <= n <= 113 - H floating
function_code	BIT(32) ALIGNED
identifier	POINTER

io_status_block	Since there are different formats for I/O status blocks for various system services, different definitions will be appropriate for different uses. Some of the common formats are shown here. [6]
	
	/* See p. SYS-229 */
	1 IOSB_SYS$GETSYI,
	    2 STATUS FIXED BINARY(31),
	    2 RESERVED FIXED BINARY(31);
	
	/* See fig. 8-16 in Part I of the I/O User's Guide */
	1 IOSB_TTDRIVER_A,
	    2 STATUS FIXED BINARY(15),
	    2 BYTE_COUNT FIXED BINARY(15),
	    2 MBZ FIXED BINARY(31) INITIAL(0);
	
	/* See fig. 8-16 in Part I of the I/O User's Guide */
	1 IOSB_TTDRIVER_B,
	    2 STATUS FIXED BINARY(15),
	    2 TRANSMIT_SPEED FIXED BINARY(7),
	    2 RECEIVE_SPEED FIXED BINARY(7),
	    2 CR_FILL FIXED BINARY(7),
	    2 LF_FILL FIXED BINARY(7),
	    2 PARITY_FLAGS FIXED BINARY(7),
	    2 MBZ FIXED BINARY(7) INITIAL(0);

[6] Routines declared in PLISTARLET often use ANY so the user is free to declare the data structure in the most convenient way for her application. ANY may be necessary in some cases since PL/I does not allow parameters declarations for some data types used by VMS. (In particular, PL/I parameters with arrays passed by reference may not be declared to have nonconstant bounds.)

Table A-7 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
item_list_2	1 ITEM_LIST_2, 2 ITEM(SIZE), 3 COMPONENT_LENGTH FIXED BINARY(15), 3 ITEM_CODE FIXED BINARY(15), 3 COMPONENT_ADDRESS POINTER, 2 TERMINATOR FIXED BINARY(31) INITIAL(0);[6] Replace SIZE with the number of items you want.
item_list_3	1 ITEM_LIST_3, 2 ITEM(SIZE), 3 BUFFER_LENGTH FIXED BINARY(15), 3 ITEM_CODE FIXED BINARY(15), 3 BUFFER_ADDRESS POINTER, 3 RETURN_LENGTH POINTER, 2 TERMINATOR FIXED BINARY(31) INITIAL(0);[6] Replace SIZE with the number of items you want.
item_list_pair	1 ITEM_LIST_PAIR, 2 ITEM(SIZE), 3 ITEM_CODE FIXED BINARY(31), 3 ITEM UNION, 4 INTEGER FIXED BINARY(31), 0 REAL FLOAT BINARY(24), 2 TERMINATOR FIXED BINARY(31) INITIAL(0);[6] Replace SIZE with the number of items you want.
item_quota_list	1 ITEM_QUOTA_LIST, 2 QUOTA(SIZE), 3 NAME FIXED BINARY(7), 3 VALUE FIXED BINARY(31), 2 TERMINATOR FIXED BINARY(7) INITIAL(PQL$_LISTEND); [6] Replace SIZE with the number of quota entries that you want to use. The constant PQL$_LISTEND can be used by including the module $PQLDEF from PLISTARLET or by declaring it GLOBALREF FIXED BINARY(31) VALUE.
lock_id	FIXED BINARY(31)

[6]Routines declared in PLISTARLET often use ANY so the user is free to declare the data structure in the most convenient way for her application. ANY may be necessary in some cases since PL/I does not allow parameters declarations for some data types used by VMS. (In particular, PL/I parameters with arrays passed by reference may not be declared to have nonconstant bounds.)

Table A-7 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
lock_status_block	1 LOCK_STATUS_BLOCK, 2 STATUS_CODE FIXED BINARY(15), 2 RESERVED FIXED BINARY(15), 2 LOCK_ID FIXED BINARY(31); [6]
lock_value_block	The declaration of an item of this structure will depend on the use of the structure, since VMS does not interpret the value. [6]
logical_name	CHARACTER(n) [4]
longword_signed	FIXED BINARY(31)
longword_unsigned	FIXED BINARY(31) [3]
mask_byte	BIT(8) ALIGNED
mask_longword	BIT(32) ALIGNED
mask_quadword	BIT(64) ALIGNED
mask_word	BIT(16) ALIGNED
null_arg	Omit the corresponding parameter in the call. For example, FOO(A,,B) would omit the second parameter.
octaword_signed	BIT(128) ALIGNED [5]
octaword_unsigned	BIT(128) ALIGNED [3,5]
page_protection	FIXED BINARY(31) (The constants for this type are declared in module $PRTDEF in PLISTARLET.)
procedure	PROCEDURE or ENTRY [2]
process_id	FIXED BINARY(31)

[2] AST procedures and those passed as parameters of type ENTRY VALUE or ANY VALUE must be external procedures. This applies to all system routines which take procedure parameters.

[3] This is actually an unsigned integer. This declaration is interpreted as a signed number; use the POSINT function to determine the actual value.

[4] System services require CHARACTER string representation for parameters. Most other system routines allow either CHARACTER or CHARACTER VARYING. For parameter declarations, n should be an asterisk.

[5] VAX PL/I does not support FIXED BINARY numbers with precisions greater than 32. To use larger values, declare variables to be BIT variables of the appropriate size and use the POSINT and SUBSTR bits as necessary to access the values, or declare the item as a structure. The RTL routines LIB$ADDX and LIB$SUBX may be useful if you need to perform arithmetic on these types.

[6] Routines declared in PLISTARLET often use ANY so the user is free to declare the data structure in the most convenient way for her application. ANY may be necessary in some cases since PL/I does not allow parameters declarations for some data types used by VMS. (In particular, PL/I parameters with arrays passed by reference may not be declared to have nonconstant bounds.)

Table A-7 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
process_name	CHARACTER(n) [4]
quadword_signed	BIT(64) ALIGNED [5]
quadword_unsigned	BIT(64) ALIGNED [3,5]
rights_holder	1 RIGHTS_HOLDER, 2 RIGHTS_ID FIXED BINARY(31), 2 ACCESS_RIGHTS BIT(32) ALIGNED; [6]
rights_id	FIXED BINARY(31)
rab	See module $RABDEF in PLISTARLET [6]
section_id	BIT(64) ALIGNED
section_name	CHARACTER(n) [4]
system_access_id	BIT(64) ALIGNED
time_name	CHARACTER(n) [4]
uic	FIXED BINARY(31)
user_arg	ANY
varying_arg	ANY with OPTIONS(VARIABLE) on the routine declaration.
vector_byte_signed	(n) FIXED BINARY(7) [7]
vector_byte_unsigned	(n) FIXED BINARY(7) [3,7]
vector_longword_signed	(n) FIXED BINARY(31) [7]
vector_longword_unsigned	(n) FIXED BINARY(31) [3,7]
vector_quadword_signed	(n) BIT(64) ALIGNED [5,7]

[3] This is actually an unsigned integer. This declaration is interpreted as a signed number; use the POSINT function to determine the actual value.

[4] System services require CHARACTER string representation for parameters. Most other system routines allow either CHARACTER or CHARACTER VARYING. For parameter declarations, n should be an asterisk.

[5] VAX PL/I does not support FIXED BINARY numbers with precisions greater than 32. To use larger values, declare variables to be BIT variables of the appropriate size and use the POSINT and SUBSTR bits as necessary to access the values, or declare the item as a structure. The RTL routines LIB$ADDX and LIB$SUBX may be useful if you need to perform arithmetic on these types.

[6] Routines declared in PLISTARLET often use ANY so the user is free to declare the data structure in the most convenient way for her application. ANY may be necessary in some cases since PL/I does not allow parameters declarations for some data types used by VMS. (In particular, PL/I parameters with arrays passed by reference may not be declared to have nonconstant bounds.)

[7] For parameter declarations, the bounds must be constant for arrays passed by reference. For arrays passed by descriptor, *s should be used for the array extent instead. (VMS system routines almost always take arrays by reference.)

Table A-7 (Cont.) VAX PL/I Implementation

VMS Data Type	VAX PL/I Declaration
vector_quadword_unsigned	(n) BIT(64) ALIGNED [3,5,7]
vector_word_signed	(n) FIXED BINARY(15) [7]
vector_word_unsigned	(n) FIXED BINARY(15) [3,7]
word_signed	FIXED BINARY(15)
word_unsigned	FIXED BINARY(15) [3]

[3]This is actually an unsigned integer. This declaration is interpreted as a signed number; use the POSINT function to determine the actual value.

[5]VAX PL/I does not support FIXED BINARY numbers with precisions greater than 32. To use larger values, declare variables to be BIT variables of the appropriate size and use the POSINT and SUBSTR bits as necessary to access the values, or declare the item as a structure. The RTL routines LIB$ADDX and LIB$SUBX may be useful if you need to perform arithmetic on these types.

[7]For parameter declarations, the bounds must be constant for arrays passed by reference. For arrays passed by descriptor, *s should be used for the array extent instead. (VMS system routines almost always take arrays by reference.)

NOTE: All system services and many system constants and data structures are declared in PLISTARLET.TLB. For examples of using system services, see either the *VAX-11 PL/I User's Guide* or *Programming in VAX-11 PL/I.*

Important note: While the current version of VAX PL/I Version 2 does not support unsigned fixed binary numbers or fixed binary numbers with a precision greater than 31, it is possible that future versions may support these features. If VAX PL/I is extended to support these types, it is possible that declarations in PLISTARLET will change to use the new data types where appropriate.

Index

B

BLISS implementation table
  See Implementation table

C

C implementation table
  See Implementation table

D

Data type
  VMS
    definition of, A-1
    description of, A-2 to A-21

F

FORTRAN implementation table
  See Implementation table

I

Implementation table
  VAX BLISS, A-21
  VAX C, A-24
  VAX FORTRAN, A-27
  VAX MACRO, A-32
  VAX PASCAL, A-35
  VAX PL/I, A-39
  VMS Usage, A-2

M

MACRO implementation table
  See Implementation table

P

PASCAL implementation table
  See Implementation table
PL/I implementation table
  See Implementation table

V

VAX language implementation table
  See Implementation table
VMS Usage implementation table
  See Implementation table

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

____

____

____

____

____

____

Did you find errors in this manual? If so, specify the error and the page number.

____

____

____

____

____

____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ____________________

Name ____________________ Date ____________

Organization ____________________________

Street ________________________________

City __________________ State ________ Zip Code or Country ________

Do Not Tear - Fold Here and Tape

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here

Cut Along Dotted Line

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

____

____

____

____

____

____

Did you find errors in this manual? If so, specify the error and the page number.

____

____

____

____

____

____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ______________________

Name ______________________ Date ______________

Organization ______________________________

Street ______________________________

City ______________________ State ____________ Zip Code ________
or Country

Do Not Tear - Fold Here and Tape

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here

Cut Along Dotted Line