digital

OpenVMS Alpha Guide to 64-Bit Addressing

OpenVMS

OpenVMS

OpenVMS Alpha Guide to 64-Bit Addressing

Order Number: AA–QSBCA–TE

December 1995

This new manual describes the OpenVMS Alpha 64-bit virtual addressing support provided in OpenVMS Alpha Version 7.0.

DRAFT NOTICE

This document contains preliminary information.

Revision/Update Information: This is a new manual.

Software Version: OpenVMS Alpha Version 7.0

Digital Confidential

Digital Equipment Corporation
Maynard, Massachusetts

December 1995

ZK6467

This document is available on CD–ROM.

This document was prepared using VAX DOCUMENT Version 2.1.

Contents

Preface vii

1 Introduction

1.1 64-Bit Virtual Address Space Layout 1–1
1.1.1 Process Private Space 1–3
1.1.2 System Space 1–4
1.2 Page Table Space 1–4
1.2.1 Virtual Regions 1–5
1.2.2 Potential Uses for User-Defined Regions 1–5
1.2.3 Process Permanent Regions 1–5

2 System Services Support for 64-Bit Addressing

2.1 System Services Definitions 2–1
2.1.1 32-Bit System Service 2–1
2.1.2 64-Bit System Service 2–1
2.1.3 64-Bit Friendly Interface 2–1
2.2 64-Bit System Services 2–2

3 RMS Interface Enhancements for 64-Bit Addresses

3.1 The RAB64 Data Structure 3–2
3.2 Using the 64-Bit RAB Extension 3–3

4 OpenVMS Alpha Device Support for 64-Bit Addressing

4.1 $QIO Support for 64-Bit Addresses 4–1
4.2 OpenVMS Drivers Supporting 64-Bit Addresses 4–2
4.3 Function Codes that Support 64-Bit Addresses 4–4
4.4 64-Bit IO$_DIAGNOSE Function for SCSI class Drivers 4–5
4.4.1 64-bit S2DGB Example 4–10

5 OpenVMS Alpha 64-Bit API Guidelines

5.1 Quadword/Longword Argument Pointer Guidelines 5–1
5.2 Alpha/VAX Guidelines 5–6
5.3 Style Guidelines 5–7
5.4 Promoting an API from a 32-Bit API to a 64-Bit API 5–8
5.5 No new 64-bit MACRO-32 macros are available for system services. 5–9
5.6 Example of a 32-bit routine and a 64-bit routine 5–9

6 OpenVMS Alpha Tools and Utilities That Support 64-Bit Addressing

6.1 OpenVMS Debugger 6–1
6.2 OpenVMS Alpha System-Code Debugger 6–1
6.3 XDELTA 6–2
6.4 LIB$ and CVT$ Facilities of the OpenVMS Run-Time Library 6–2
6.5 Watchpoint Utility 6–2
6.6 SDA 6–3

7 DEC C RTL Support for 64-Bit Addressing

7.1 Using the DEC C Run-Time Library 7–1
7.2 Obtaining 64-bit Pointers to Memory 7–2
7.3 DEC C Header Files 7–2
7.4 Functions Affected 7–3
7.4.1 No Pointer-Size Impact 7–3
7.4.2 Functions Accepting Both Pointer Sizes 7–4
7.4.3 Functions with Two Implementations 7–4
7.4.4 Restricted to 32-Bit Pointers 7–5
7.5 Reading Header Files 7–6

8 MACRO–32 Programming Support for 64-Bit Addressing

8.1 Guidelines for 64-Bit Addressing 8–1
8.2 New and Changed Components for 64-Bit Addressing 8–1
8.3 Passing 64-Bit Values 8–2
8.3.1 Calls with a Fixed-Size Argument List 8–2
8.3.1.1 Usage Notes for $SETUP_CALL64, $PUSH_ARG64, and $CALL64 8–3
8.3.2 Calls with a Variable-Size Argument List 8–4
8.4 Declaring 64-Bit Arguments 8–4
8.4.1 Usage Notes for QUAD_ARGS 8–5
8.5 Specifying 64-Bit Address Arithmetic 8–5
8.5.1 Dependence on Wrapping Behavior of Longword Operations 8–6
8.6 Sign Extending and Checking 8–6
8.7 Alpha Instruction Built-ins 8–7
8.8 Calculating Page-Size Dependent Values 8–7
8.9 Creating and Using Buffers in 64-Bit Address Space 8–7
8.10 Coding for Moves Longer Than 64K Bytes 8–7

A 64-Bit Example Program

B MACRO-32 Macros for 64-Bit Addressing

B.1 Macros for Manipulating 64-Bit Addresses B–1
$SETUP_CALL64 . . . . . B–1
$PUSH_ARG64 . . . . . B–2
$CALL64 . . . . . B–3
B.2 Macro for Checking the Sign Extension B–4
$IS_32BITS B–4

Index

Examples

4–1	OpenVMS SCSI-2 Diagnose Buffer (S2DGB) 32-Bit Layout	4–6
4–2	OpenVMS SCSI-2 Diagnose Buffer (S2DGB) 64-Bit Layout	4–7

Figures

1–1	32-Bit Virtual Address Space Layout	1–2
1–2	64-Bit Virtual Address Space Layout	1–3

Tables

2–1	64-Bit System Services	2–2
4–1	Drivers Supporting 64-Bit Addresses	4–2
4–2	Drivers Restricted to 32-Bit Addresses	4–3
4–3	64-Bit Capable I/O Functions	4–5
7–1	Functions with Dual Implementations	7–5
7–2	Functions restricted to 32-bit pointers	7–5
7–3	Callbacks that Pass Only 32-Bit Pointers	7–6
8–1	New and Changed Components for 64-Bit Addressing	8–1
8–2	Passing 64-Bit Values with a Fixed-Size Argument List	8–2

Preface

This guide describes OpenVMS Alpha operating system support for 64-bit virtual addressing.

The information in this document applies only to applications on OpenVMS Alpha systems; applications on OpenVMS VAX systems are not affected.

Intended Audience

This information in this guide is intended for system and application programmers. It presumes that its readers are familiar with the OpenVMS Alpha programming environment and concepts.

Document Structure

Chapter 1 presents an overview of the OpenVMS Alpha 64-bit virtual address space layout and the operating system tools and languages that support 64-bit addressing. The following chapters contain more details about these topics.

Related Documents

This guide provides high-level descriptions of some of the topics covered in the following manuals; refer to these books for more detailed information:

- *OpenVMS Programming Concepts Manual*
- *OpenVMS Calling Standard*
- *OpenVMS System Services Reference Manual: A–GETMSG* and *OpenVMS System Services Reference Manual: GETQUI–Z*
- *OpenVMS Record Management Services Reference Manual*
- *OpenVMS RTL Library (LIB$) Manual*
- *OpenVMS Debugger Manual*
- *OpenVMS Alpha System Dump Analyzer Utility Manual*
- *OpenVMS Alpha Guide to Upgrading Privileged-Code Applications*

  If you have an application that links against the base system image SYS$BASE_IMAGE.EXE, you might need to relink, recompile, or make source-code changes for OpenVMS Alpha Version 7.0. Refer to this manual for more information about the changes that might affect privileged-code applications and device drivers.

For additional information about OpenVMS products and services, access the Digital OpenVMS World Wide Web site. Use the following URL:

```
http://www.openvms.digital.com
```

Reader's Comments

Digital welcomes your comments on this manual.

Print or edit the online form SYS$HELP:OPENVMSDOC_COMMENTS.TXT and send us your comments by:

Internet	**openvmsdoc@zko.mts.dec.com**
Fax	603 881-0120, Attention: OpenVMS Documentation, ZK03-4/U08
Mail	OpenVMS Documentation Group, ZKO3-4/U08 110 Spit Brook Rd. Nashua, NH 03062-2698

How To Order Additional Documentation

Use the following table to order additional documentation or information. If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Location	Call	Fax	Write
U.S.A.	DECdirect 800–DIGITAL 800–344–4825	Fax: 800–234–2298	Digital Equipment Corporation P.O. Box CS2008 Nashua, NH 03061
Puerto Rico	809–781–0505	Fax: 809–749–8300	Digital Equipment Caribbean, Inc. 3 Digital Plaza, 1st Street, Suite 200 P.O. Box 11038 Metro Office Park San Juan, Puerto Rico 00910–2138
Canada	800–267–6215	Fax: 613–592–1946	Digital Equipment of Canada, Ltd. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: DECdirect Sales
International	—	—	Local Digital subsidiary or approved distributor
Internal Orders	DTN: 264–4446 603–884–4446	Fax: 603–884–3960	U.S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063–1260

ZK–7654A–GE

Conventions

The name of the OpenVMS AXP operating system has been changed to OpenVMS Alpha. Any references to OpenVMS AXP or AXP in this document are synonymous with OpenVMS Alpha or Alpha.

1
Introduction

The OpenVMS Alpha operating system provides support for 64-bit virtual memory addressing, which makes the 64-bit virtual address space defined by the Alpha architecture available to the OpenVMS Alpha operating system and to application programs. In the 64-bit virtual address space, both process private and system virtual address space can extend beyond 2 GB. By using 64-bit addressing features, programmers can create images that map and access data beyond the limits of 32-bit virtual addresses.

Many OpenVMS Alpha tools and languages (including the Debugger, run-time library routines, and DEC C) support 64-bit virtual addressing. Input and output operations can be performed directly to and from the 64-bit addressable space by means of RMS services, the $QIO system service, and most of the device drivers supplied with OpenVMS Alpha systems.

Underlying this are new system services, which allow an application to allocate and manage the 64-bit virtual address space that is available for process private use.

Note that in order to take advantage of 64-bit addressing features, nonprivileged programs can optionally be modified. In other words, OpenVMS Alpha 64-bit virtual addressing does not affect nonprivileged programs that are not explicitly modified to exploit 64-bit support. Binary and source compatibility of existing 32-bit nonprivileged programs is guaranteed.

This chapter describes the layout of the OpenVMS Alpha 64-bit virtual memory address space. For more information about using specific programming features to take full advantage of 64-bit addressing support, refer to the remaining chapters in this guide.

1.1 64-Bit Virtual Address Space Layout

The 64-bit virtual address space layout is designed to accommodate the current and future needs of the OpenVMS Alpha operating system and its users. The 64-bit design ensures upward compatibility of programs that currently execute under OpenVMS, while providing a flexible framework within which 64-bit addresses can be used in many different ways to solve new problems.

The 64-bit address space layout is an extension of the traditional OpenVMS 32-bit address space layout. Figure 1–1 illustrates the 32-bit virtual address space layout design.

Figure 1–1 32-Bit Virtual Address Space Layout

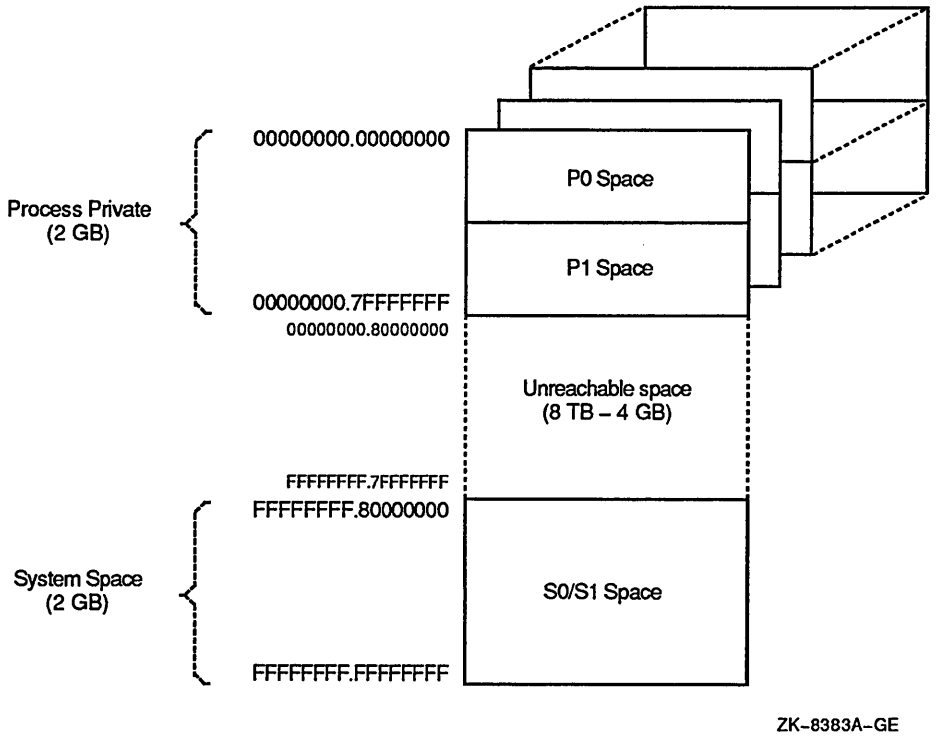
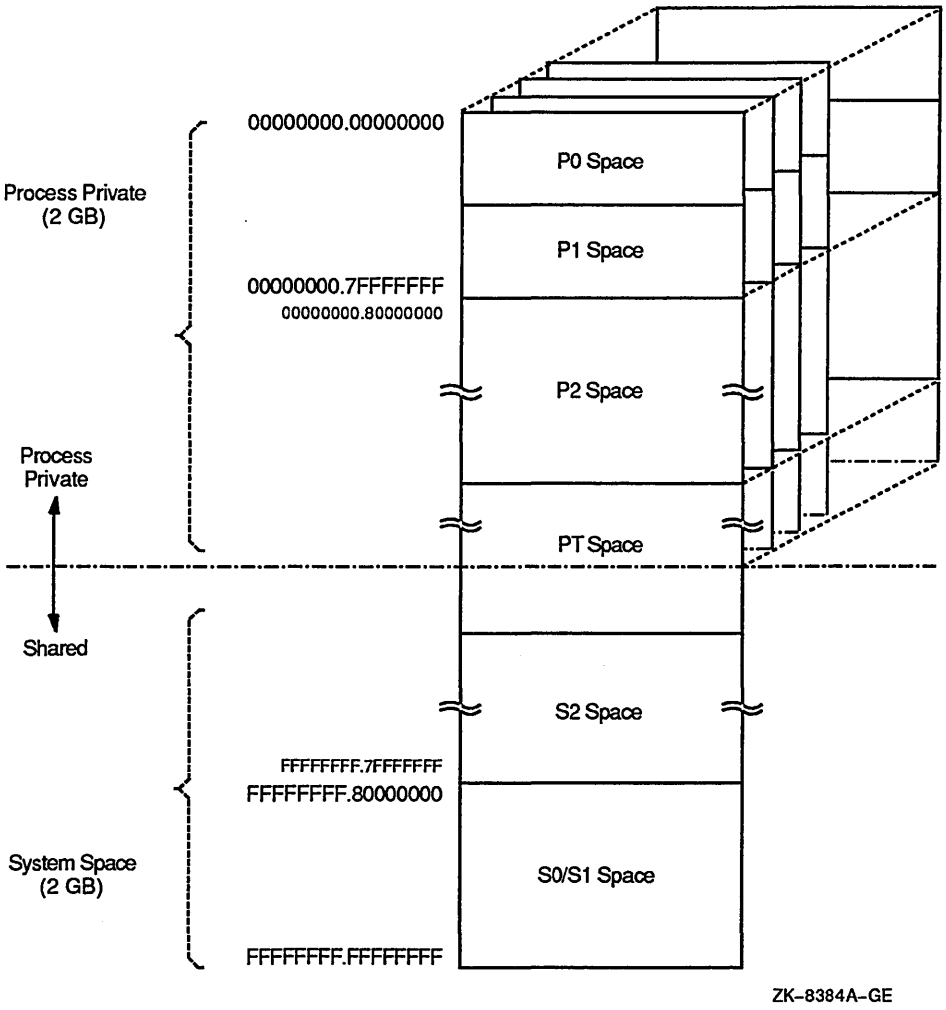


Figure 1–2 illustrates the 64-bit virtual address space layout design. The sections that follow briefly describe these areas.

Figure 1–2 64-Bit Virtual Address Space Layout



The new address space consists of the following fundamental areas:

- Process private space
- System space
- Page table space

1.1.1 Process Private Space

Supporting process private address space is a focus of much of the memory management design within the OpenVMS operating system.

Process private space, or process space, is the portion of the entire 64-bit virtual address range that is lower than that which contains PT Space. As shown in Figure 1–2, the layout of process space is further divided into the P0, P1, and P2 spaces. P0 space refers to the program region. P1 space refers to the control region. P2 space refers to the default 64-bit program region.

The **P0** and **P1 spaces** are formally defined to equate to the P0 and P1 regions defined by *DEC STD 032, VAX Architecture Standard.* Together, they encompass the traditional 32-bit process private region that ranges from 0.00000000_{16} to $0.7FFFFFFF_{16}$. **P2 space** encompasses all remaining process space that begins

just above P1 space, 0.80000000_{16}, and ends just below the lowest address of PT Space. In other words, P2 space is HUGE!

1.1.2 System Space

64-Bit system space refers to the portion of the entire 64-bit virtual address range that is higher than that which contains PT_Space. As shown in Figure 1–2, system space is further divided into the S0, S1, and S2 spaces.

The **S0** and **S1 spaces** are formally defined to equate to the S0 and S1 regions defined by *DEC STD 032, VAX Architecture Standard.* Together they encompass the traditional 32-bit system space region that ranges from $FFFFFFFF.80000000_{16}$ to $FFFFFFFF.FFFFFFFF_{16}$. **S2 space** encompasses all remaining system spaces between the highest address of PT_Space and the lowest address of the combined S0/S1 space.

S0, S1, and S2 are fully shared by all processes. S0/S1 space expands toward increasing virtual addresses. In order to maximize the use of shared system page tables, S2 space generally expands toward lower virtual addresses.

Addresses within system space can only be created and deleted from code that is executing in kernel mode. However, page protection for system space pages can be set up to allow any less privileged access mode read and/or write access.

System space base is controlled by a major SYSGEN parameter. The default value is based on the sizes required by expected consumers of 64-bit (S2) system space. These known consumers are the PFN database and the global page table.

The **global page table**, also known as the **GPT**, and the PFN database reside in the lowest-addressed portion of S2 space. Larger global sections expected in a 64-bit OpenVMS Alpha system require a larger global page table that may not fit easily into 32-bit system space.

Larger-memory systems expected in a 64-bit OpenVMS Alpha system also require a larger PFN database that may also not fit into 32-bit system space.

1.2 Page Table Space

In previous versions of the OpenVMS Alpha operating system, **page table space**, also known as **PT Space**, was addressable in more than one way. The PALcode TB miss handler used addresses starting at 2.00000000_{16} to read PTEs, while memory management code addressed the page tables primarily within the traditional 32-bit system space. (The only exception was when process page tables were paged out, and they had to be invalidated from both the address space used by the PALcode TB miss handler and their location in traditional 32-bit system space.) The process page tables were within the process header (PHD), and the system space page tables were located in the highest virtual addresses, all within the traditional 32-bit system space.

As of OpenVMS Alpha Version 7.0, page tables are addressed primarily within 64-bit PT_Space. Page table references are to this virtual address range, which is in process private address space—not in system shared address space. Process page tables have been removed from 32-bit system space.

The dotted line in Figure 1–2 marks the boundary between process private space and shared space. This boundary is in PT_space and marks the first PTE that maps shared system space.

1.2.1 Virtual Regions

Allows for reserving address space => "light weight" objects

Promotes application modularity

Allows address space to be used sparsely

Regions do not overlap

Addresses within regions expand in a "dense" manner (same as P0/P1)

Regions are created as either **ascending** (like P0) or **descending** (like P1)

Regions are created within P2 space

Region base VA is fixed at creation

Region size is fixed at creation

Protection of user defined regions, owner mode/create mode:

- Owner mode specified at creation
- Owner mode or more privileged can delete region
- Create mode or mode privileged can map files/create address space within a region
- Page protection applies to created address space

User-defined regions are deleted at image rundown.

1.2.2 Potential Uses for User-Defined Regions

Reserve address space: Guarantee virtually contiguous VAs

Regions for thread stacks

Memory mapped files may expand virtually contiguously

1.2.3 Process Permanent Regions

3 process permanent regions: Program, Control, 64-bit Program

- 64-bit Program Region = Default P2 Region
- Process permanent regions cannot be deleted.
- Process permanent regions encompass the entire address space up to first user-defined region within that address space (P0, P1 or P2).

  For example:

  When user-defined region is created in P2 space => 64-bit program region may be shrunk
  When user-defined region is deleted in P2 space => 64-bit program region may grow
  At image rundown, 64-bit program region (default P2) is reset to encompass all of P2 space.

2

System Services Support for 64-Bit Addressing

This chapter describes the OpenVMS Alpha system services that support 64-bit addressing. It explains the changes made to 32-bit services to support 64-bit addresses, and it lists the new 64-bit system services.

For more information about OpenVMS System Services that support OpenVMS Alpha 64-bit virtual addressing, see the *OpenVMS System Services Reference Manual: A–GETMSG* and *OpenVMS System Services Reference Manual: GETQUI–Z.*

2.1 System Services Definitions

The following system services definitions are used throughout this guide.

2.1.1 32-Bit System Service

A 32-bit system service is a system service that only supports 32-bit addresses on any of its arguments that specify addresses. If passed by value, on OpenVMS Alpha a 32-bit virtual address is actually a 64-bit address that is sign-extended from 32-bits.

2.1.2 64-Bit System Service

A 64-bit system service is a system service that is defined to accept all address arguments as 64-bit addresses (not necessarily 32-bit sign-extended values). Also, a 64-bit system service uses the entire 64 bits of all virtual addresses passed to it.

64-bit system services include the _64 suffix for services that accept 64-bit addresses by reference. For promoted services, this distinguishes the 64-bit capable version from its 32-bit counterpart. For new services, it is a visible reminder that a 64-bit wide address cell will be read/written. This is also used when a structure is passed which contains an embedded 64-bit address, IF the structure is not self-identifying as a 64-bit structure. Hence, a routine name need not include "_64" simply because it receives a 64-bit decriptor. Remember that passing an arbitrary value by reference does not mean the suffix is required; passing a 64-bit address by reference does.

2.1.3 64-Bit Friendly Interface

A 64-bit friendly interface is an interface that can be called with all 64-bit addresses. A 32-bit system service interface is 64-bit friendly if, without a change in the interface, it needs no modification to handle 64-bit addresses. The internal code that implements the system service might need modification, but not the system service interface.

The majority of OpenVMS Alpha system services prior to OpenVMS Alpha Version 7.0 have 64-bit friendly interfaces for the following reasons:

- The *OpenVMS Calling Standard* defines arguments to standard routines to be 64 bits wide. The caller of a routine sign-extends 32-bit arguments to be 64 bits.
- 64-bit string descriptors can be distinguished from 32-bit string descriptors at run time. (See the *OpenVMS Calling Standard* for more information about 64-bit descriptors.)
- User visible RMS data structures containing embeded type information such that the RMS routines can tell whether a non-32-bit form of a structure is being used. (See Chapter 3 for more details about RMS 64-bit addressing support.)

Examples of routines with 64-bit unfriendly interfaces are most of the memory management system services, such as $CRETVA, $DELTVA and $CRMPSC. The INADR and RETADR argument arrays do not promote easily to hold 64-bit addresses.

2.2 64-Bit System Services

Table 2–1 summarizes the OpenVMS Alpha system services that support 64-bit addresses. It includes system services from previous releases that have been enhanced to handle 64-bit addresses as well as new OpenVMS Alpha 64-bit system services.

Although RMS system services provide some 64-bit addressing capabilities, they are not listed in this table because they are not full 64-bit system services. See Chapter 3 for more details.

Table 2–1 64-Bit System Services

Service	Arguments
Alignment System Services	
$INIT_SYS_ALIGN_FAULT_REPORT	(match_table_64, buffer_size, flags)
$GET_ALIGN_FAULT_DATA	(buffer_64, buffer_size, return_size_64)
$GET_SYS_ALIGN_FAULT_DATA	(buffer_64, buffer_size, return_size_64)
$SAVE_SYS_ALIGN_FAULT_DATA	(fault_pc0_64, fault_pc1_64, fault_va0_64, fault_va1_64, fault_bit_mask, fault_ps_64)
$SAVE_ALIGN_FAULT_DATA	(fault_pc0_64, fault_pc1_64, fault_va0_64, fault_va1_64)
AST System Service	
$DCLAST	(astadr_64, astprm_64, acmode)

(continued on next page)

Table 2–1 (Cont.) 64-Bit System Services

Service	Arguments
Condition Handling System Services	
$FAO	(ctrstr_64, outlen_64, outbuf_64, p1_64...pn_64)
$FAOL	(ctrstr_64, outlen_64, outbuf_64, long_prmlst_64)
$FAOL_64	(ctrstr_64, outlen_64, outbuf_64, quad_prmlst_64)
$GETMSG	(msgid, msglen_64, bufadr_64, flags, outadr_64)
$PUTMSG	(msgvec_64, actrtn_64, facnam_64, actprm_64)
$SIGNAL_ARRAY_64	(mcharg, sigarg_64)
Event Flag System Services	
$READEF	(efn, state_64)
I/O System Services	
$QIO(W)	(efn, chan, func, iosb_64, astadr_64, astprm_64, p1_64, p2_64, p3_64, p4_64, p5_64, p6_64)
$SYNCH	(efn, iosb_64)
Locking System Services	
$ENQ(W)	(efn, lkmode, lksb_64, flags, resnam_64, parid, astadr_64, astprm_64, blkast_64, acmode)
$DEQ	(lkid, vablk_64, acmode, flags)

(continued on next page)

Table 2–1 (Cont.) 64-Bit System Services

Service	Arguments
Memory Management System Services	
$CREATE_REGION_64	(length_64, region_prot, flags, return_region_id_64, return_va_64, return_length_64, ...)
$DELETE_REGION_64	(region_id_64, acmode, return_va_64, return_length_64)
$GET_REGION_INFO	(function_code, region_id_64, start_va_64, ,buffer_length, buffer_address_64, return_length_64)
$EXPREG_64	(region_id_64, length_64, acmode, flags, return_va_64, return_length_64)
$CRETVA_64	(region_id_64, start_va_64, length_64, acmode, flags, return_va_64, return_length_64)
$CRMPSC_FILE_64	(region_id_64, file_offset_64, length_64, chan, acmode, flags, return_va_64, return_length_64, ...)
$CRMPSC_PFN_64	(region_id_64, start_pfn, page_count, acmode, flags, return_va_64, return_length_64, ...)
$UPDSEC_64(W)	(start_va_64, length_64, acmode, updflg, efn, iosa_64, return_va_64, return_length_64, ...)
$DELTVA_64	(region_id_64, start_va_64, length_64, acmode, return_va_64, return_length_64)
$CREATE_GFILE	(gsdnam_64, ident_64, file_offset_64, length_64, chan, acmode, flags, return_length_64, ...)
$CREATE_GPFILE	(gsdnam_64, ident_64, prot, length_64, acmode, flags,)
$CREATE_GPFN	(gsdnam_64, ident_64, prot, start_pfn, page_count, acmode, flags)
$DGBLSC	(flags, gsdnam_64, ident_64)
$MGBLSC_64	(gsdnam_64, ident_64, region_id_64, section_offset_64, length_64, acmode, flags, return_va_64, return_length_64, ...)
$MGBLSC_GPFN_64	(gsdnam_64, ident_64, region_id_64, relative_page, page_count, acmode, flags, return_va_64, return_length_64, ...)
$CRMPSC_GFILE_64	(gsdnam_64, ident_64, file_offset_64, length_64, chan, region_id_64, section_offset, acmode, flags, return_va_64, return_length_64, ...)
$CRMPSC_GPFILE_64	(gsdnam_64, ident_64, prot, length_64, region_id_64, section_offset_64, acmode, flags, return_va_64, return_length_64, ...)
$CRMPSC_GPFN_64	(gsdnam_64, ident_64, prot, start_pfn, page_count, region_id_64, relative_page, acmode, flags, return_va_64, return_length_64, ...)

(continued on next page)

Table 2–1 (Cont.) 64-Bit System Services

Service	Arguments
Memory Management System Services	
$ADJWSL	(pagcnt, wsetlm_64)
$LKWSET_64	(start_va_64, length_64, acmode, return_va_64, return_length_64)
$ULWSET_64	(start_va_64, length_64, acmode, return_va_64, return_length_64)
$PURGE_WS	(start_va_64, length_64)
$LCKPAG_64	(start_va_64, length_64, acmode, return_va_64, return_length_64)
$ULKPAG_64	(start_va_64, length_64, acmode, return_va_64, return_length_64)
$CREATE_BUFOBJ_64	(start_va_64, length_64, acmode, flags, return_va_64, return_length_64, return_buffer_handle_64)
$DELETE_BUFOBJ	(buffer_handle_64)
$SETPRT_64	(start_va_64, length_64, acmode, prot, return_va_64, return_length_64, return_prot_64)
Time System Services	
$GETTIM	(timadr_64)
$SETIMR	(efn, daytim_64, astadr_64, reqidt_64, flags)
$CANTIM	(reqidt_64, acmode)
Other System Services	
$CMEXEC_64	(routine_64, quad_arglst_64)
$CMKRNL_64	(routine_64, quad_arglst_64)

3

RMS Interface Enhancements for 64-Bit Addresses

The RMS user interface consists of a number of control data structures (FAB, RAB, NAM, XABs) that are linked together with 32-bit wide pointers and contain embedded pointers to user data buffers of various kinds, including file name strings and item lists, as well as I/O buffers. RMS support for 64-bit addressable regions allows 64-bit addresses for all of the current user I/O buffers, except for the prompt buffer (pointed to by RAB$L_PBF), as follows:

- UBF - user record buffer
- RBF - record buffer
- RHB - fixed-length record header buffer (fixed portion of VFC record format)
- KBF - key buffer containing the key value for random access

Specific enhancements to the RMS interface for 64-bit addressing are as follows:

- Data buffers in the P2 or S2 space exist for the following services:
  - Record I/O: $GET, $FIND, $PUT, $UPDATE
  - Block I/O: $READ, $WRITE
- The RAB structure points to the record and data buffers used by these services.
- An extension of the existing RAB structure is used for 64-bit buffer pointers and sizes.
- Changes to buffer size maximums:
  - Buffer size maximum for RMS block I/O services ($READ and $WRITE) has been increased from 65535 bytes to 2**31-1 bytes, with two exceptions:

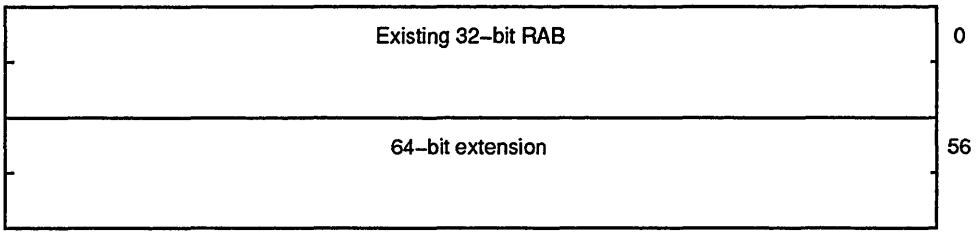
    For RMS journaling, a journaled $WRITE is restricted to the current maximum (65535 minus 99 bytes of journaling overhead); an RSZ error is returned (RAB$L_STS) if the maximum is exceeded.

    Magnetic tape continues to be limited at the device driver level to 65535 bytes.
  - Buffer size maximums for RMS record I/O services ($GET, $PUT, $UPDATE) have not changed. Prior RMS on-disk record size limits still apply.
- The rest of the RMS interface is restricted to 32-bits at this time:
  - FAB, RAB, NAM, XABs, must continue to be allocated in 32-bit space.
  - Any descriptors or embedded pointers to file names, item lists, etc. must continue to be 32-bit pointers.

- Any arguments passed to the RMS system services remain 32-bit arguments. If a user attempts to pass a 64-bit argument, the error SS$_ARG_GTR_32_BITS is returned.

3.1 The RAB64 Data Structure

The RAB data structure has been extended as follows:



ZK–8202A–GE

The extended RAB or RAB64 structure contains the following new fields:

RAB64$PQ_UBF	User buffer address ($GET, $READ). Extension counterpart for RAB64$L_UBF
RAB64$Q_USZ	User buffer size. Extension counterpart for RAB64$W_USZ
RAB64$PQ_RBF	Record buffer address ($PUT, $UPDATE, $WRITE). Extension counterpart for RAB64$L_RBF
RAB64$Q_RSZ	Record buffer size. Extension counterpart for RAB64$W_RSZ
RAB64$PQ_KBF	Key buffer address containing the key value for random access ($GET, $FIND). Extension counterpart for RAB64$L_KBF
RAB64$PQ_RHB	Record header buffer address (fixed portion of VFC record format). Extension counterpart for RAB64$L_RHB
RAB64$Q_CTX	User context (extension counterpart for RAB64$L_CTX). This cell is not used by RMS, but is available to the user. It is analogous to the current RAB$L_CTX cell, which is often used to contain a pointer. For asynchronous I/O, it provides the user with the equivalent of an AST parameter.

Note

The symbolic prefix RAB64 may be either RAB or RAB64 for MACRO or BLISS and must be RAB64 for DEC C. The DEC C structure is `rab64def`.

Buffered PQ fields can hold either 32-bit (sign-extended to 64-bits) or 64-bit addresses. Therefore, you can use the new fields in all applications whether or not you are using 64-bit addresses.

For most record I/O service requests, there is an RMS internal buffer between the device and the user's data buffer. The one exception is the RMS service $PUT in the case of a unit record device. If the device is a unit record device and it is not a network file access, RMS passes the address of the user record buffer (RBF) to the $QIO system service. If a user inappropriately specifies a record buffer (RBF) allocated in 64-bit address space for a $PUT to a unit record device that does not support 64-bit address space, the $QIO service will return SS$_NOT64DEVFUNC. (See <REFERENCE>(drivers_chapt) for more information about $QIO.) RMS will return an RMS error status (RMS-F-SYS, QIO System Service Request Failed) and the SS$_NOT64DEVFUNC as the secondary status value (RAB$L_STV).

3.2 Using the 64-Bit RAB Extension

Because RAB64 is an extension of the existing RAB, applications that do not include the source changes required to use RAB64 can continue to use the same RAB fields in the current RAB structure.

Only minimal source code changes are required for applications to use 64-bit RMS support.

The 64-bit buffer address counterpart is used only if the following two conditions are met:

- The extension is present; that is, the RAB$B_BLN (or RAB64$B_BLN) field has been initialized to RAB$C_BLN64 (or RAB64$C_BLN64).
- The 32-bit address cell in the 32-bit portion of the RAB contains a value of -1.

The new quadword size cell is used if use of the new address cell is designated in the 32-bit address cell. For example:

RAB64$L_UBF	-1
RAB64$W_USZ	0
RAB64$PQ_UBF	64-bit address
RAB64$Q_USZ	Buffer size

Note

The symbolic prefix RAB64 may be either RAB or RAB64 for MACRO or BLISS and must be RAB64 for DEC C. The DEC C structure is `rab64def`.

The 64-bit extension to the user RAB structure includes the following new macros:

- MACRO-32 macros: $RAB64 and $RAB64_STORE
  - $RAB64 (counterpart to $RAB)
  - $RAB64_STORE (counterpart to $RAB_STORE)

    The original longword I/O buffers are initialized to -1 and the USZ and RSZ word sizes to 0. RAB$B_BLN is assigned the constant of RAB$C_BLN64.

Values specified using the UBF, USZ, RBF, RSZ, RHB, or KBF keywords will be moved into the quadword cells for these keywords. (The $RAB and $RAB_STORE macros move them into the longword (or word) cells for these keywords.)

- BLISS macros: $RAB64, $RAB64_INIT, $RAB64_DECL

  The following Bliss macros are available only in the STARLET.R64 library because they use the QUAD keyword, which is available only to Bliss-64. Thus, any Bliss routines referencing them must be compiled with the Bliss-64 compiler.

  - $RAB64 (counterpart to $RAB)
  - $RAB64_INIT (counterpart to $RAB_INIT)

    The original longword I/O buffers are initialized to -1, and the USZ and RSZ word sizes are initialized to 0. RAB$B_BLN is assigned the constant of RAB$C_BLN64.

    Values assigned to the keywords UBF, USZ, RBF, RSZ, RHB or KBF will be moved into the quadword cells for these keywords. (The $RAB and $RAB_INIT macros move them into the longword (or word) cells for these keywords.)
  - $RAB64_DECL (counterpart to $RAB_DECL)

    Allocation of block structure of bytes with length RAB$C_BLN64.

4

OpenVMS Alpha Device Support for 64-Bit Addressing

Input and output operations can be performed directly to and from the 64-bit addressable space by means of RMS services, the $QIO system service, and most of the device drivers supplied with OpenVMS Alpha systems.

This chapter explain the $QIO system service support for 64-bit addresses, describes the OpenVMS Alpha device drivers that support 64-bit addresses and those that do not, and lists the OpenVMS Alpha disk and tape driver function codes that support 64-bit addresses.

Device drivers can be modified to support 64-bit addresses, but this support is not required in all drivers. For information about how to modify a customer-written device driver to support 64-bit addressing, refer to the *OpenVMS Alpha Guide to Upgrading Privileged-Code Applications*. To see an example of a device driver that supports 64-bit addressing, refer to the LRDRIVER device driver that is available in the SYS$EXAMPLES directory. This device driver has been modified to support a 64-bit buffer address in all of its functions.

Important

OpenVMS Alpha Version 7.0 includes significant changes to OpenVMS Alpha privileged interfaces and data structures. As a result of these changes, all device drivers from previous versions of OpenVMS Alpha (including field test versions of OpenVMS Alpha Version 7.0) must be recompiled and relinked to run correctly on OpenVMS Alpha Version 7.0.

For more details about OpenVMS Alpha Version 7.0 changes that may require source changes to customer-written drivers, see the *OpenVMS Alpha Guide to Upgrading Privileged-Code Applications*.

4.1 $QIO Support for 64-Bit Addresses

The $QIO service takes the following parameters:

```
$QIO[W] efn,chan,func,iosb,astadr,astprm,
        p1,p2,p3,p4,p5,p6
```

- Existing $QIO and $SYNCH system services have been enhanced by taking advantage of the Alpha architecture and OpenVMS Alpha calling standard.
- $QIO uses 64-bits of the iosb, astadr, and astprm parameters.
- $QIO checks 64-bit p1 to determine whether the caller needs 64-bit support in the driver.
- Drivers can declare support for 64 bit addressing on a function-by-function basis using ini_fdt_act (C), FDT_64 (MACRO), and FDTAB FDT_64 (BLISS).

- The maximum size of an I/O initiated by $QIO is still limited by a 32-bit byte length parameter.

4.2 OpenVMS Drivers Supporting 64-Bit Addresses

A device driver declares support for 64-bit addresses individually by I/O function code. Disk and tape device drivers support 64-bit addresses for data transfers to and from disk and tape devices on the virtual, logical, and physical read and write functions. For example, the OpenVMS SCSI disk class driver, SYS$DKDRIVER, supports 64-bit addresses on the IO$_READVBLK and IO$_WRITEVBLK functions, but not on the IO$_AUDIO function.

OpenVMS Alpha device drivers that support 64-bit adresses include the following:

- All disk and tape drivers
- All port drivers below disk and tape drivers
- LAN drivers
- Mailbox driver
- ISA parallel port driver (LRDRIVER.C)

Table 4–1 lists the OpenVMS Alpha device drivers that support for 64-bit addresses on at least some functions.

Table 4–1 Drivers Supporting 64-Bit Addresses

Driver	Description
SYS$DADDRIVER	Local area client disk
SYS$DKDRIVER	SCSI disk class driver
SYS$DQDRIVER	Mannequin simulator disk
SYS$DUDRIVER	DSA disk class driver
SYS$DVDRIVER	Intel 83077AA Floppy disk
SYS$ECDRIVER	LANCE PMAD LAN, TC
SYS$ERDRIVER	DE422 LAN, EISA
SYS$ESDRIVER	LANCE LAN
SYS$EWDRIVER	TULIP LAN, PCI
SYS$EXDRIVER	DEMNA LAN, XMI
SYS$EZDRIVER	SGEC/INEC/TGEC LAN
SYS$FADRIVER	FDDI for Futurebus
SYS$FCDRIVER	DEFZA, DEFTA LAN, TC
SYS$FRDRIVER	DEFEA LAN, EISA
SYS$FXDRIVER	DEMFA LAN, XMI
SYS$GKDRIVER	SCSI generic class driver
SYS$HCDRIVER	OTTO class ATM
SYS$ICDRIVER	TMS380 LAN, TC
SYS$IRDRIVER	TMS380 EISA Token Ring

(continued on next page)

Table 4–1 (Cont.) Drivers Supporting 64-Bit Addresses

Driver	Description
SYS$LADDRIVER	Local Area Disk
SYS$LASTDRIVER	Local Area System Trans
SYS$LRDRIVER	VL82C106 parallel printer driver
SYS$MADDRIVER	Local area client tape
MBDRIVER	Mailbox driver
SYS$MKDRIVER	SCSI tape class driver
NLDRIVER	Null device driver
SYS$PADRIVER	SHAC CI and DSSI port driver
SYS$PDDRIVER	RAM disk
SYS$PEDRIVER	NI SCS port driver
SYS$PIDRIVER	NCR 53C710 DSSI port
SYS$PKCDRIVER	SCSI NCR 53C94 Port
SYS$PKEDRIVER	NCR 53C810 SCSI port
SYS$PKJDRIVER	ADAPTEC 1742A SCSI port
SYS$PKSDRIVER	SIMport TC-SCSI port
SYS$PKTDRIVER	NCR 53C710 SCSI port
SYS$PKZDRIVER	XZA SCSI Port
SYS$PNDRIVER	NPORT SCS port
SYS$PUDRIVER	Laser KDM DSA/SCS port
SYS$SHDRIVER	Volume shadowing
SYS$TUDRIVER	MSCP/DSA tape class
SYS$WPDRIVER	Watchpoint driver

Table 4–2 lists the OpenVMS Alpha device drivers that do not support 64-bit addresses in OpenVMS Alpha Version 7.0.

Table 4–2 Drivers Restricted to 32-Bit Addresses

Driver	Description
SYS$CTDRIVER	CTERM driver
SYS$FBDRIVER	Terminal fallback driver
SYS$FTDRIVER	Psuedo terminal driver
SYS$FYDRIVER	DUP DSA protocol class driver
SYS$GQADRIVER	QVISION driver
SYS$GTADRIVER	DECwindows TX driver for Flamingo
SYS$GXADRIVER	Flamingo CXTurbo (aka SFB, aka HX) driver
SYS$GYADRIVER	SFB+ aka HX+, aka FFB driver
SYS$GYBDRIVER	Driver for TGA graphics on the PCI bus
SYS$IEDRIVER	DECwindows extension

(continued on next page)

Table 4–2 (Cont.) Drivers Restricted to 32-Bit Addresses

Driver	Description
SYS$IKBDRIVER	DECwindows PCXAL keyboard
SYS$IKDRIVER	DECwindows LKxxx keyboard
SYS$IMBDRIVER	DECwindows PCXAS (PS2) mouse
SYS$IMDRIVER	DECwindows VSxxx mouse
SYS$INDRIVER	DECwindows input driver
SYS$LTDRIVER	LAT terminal driver
NDDRIVER	DECnet Phase IV DLE (MOP support)
NETDRIVER	DECnet Phase IV
SYS$RTTDRIVER	Remote DECnet terminal driver
SYS$SODRIVER	AMD79C30A Audio/ISDN driver
SYS$TTDRIVER	Terminal class driver
DECW$XTDRIVER	X Terminal class driver
SYS$YRDRIVER	Z85C30 SCC terminal port driver
SYS$YSDRIVER	PC87312 terminal port driver

Some notable points about the drivers that are restricted to 32-bit buffer addresses include the following:

- The terminal drivers do not support 64-bit addresses.
- The drivers that are used by the DECwindows MOTIF software do not support 64-bit addresses.
- The DECnet Phase IV drivers do not support 64-bit addresses. NETDRIVER uses chained CXBs to perform buffered I/O. The code in NETDRIVER is complex and depends significantly on the 32-bit addressability of data via CXBs.

4.3 Function Codes that Support 64-Bit Addresses

Table 4–3 lists the OpenVMS Alpha I/O function codes that support 64-bit addresses. Table 4–3

Table 4–3 64-Bit Capable I/O Functions

Driver Type	Function Code	64-Bit Addresses
Disk		
	IO$_READLBLK	P1
	IO$_READPBLK	P1
	IO$_READVBLK	P1
	IO$_WRITECHECK	P1
	IO$_WRITELBLK	P1
	IO$_WRITEPBLK	P1
	IO$_WRITEVBLK	P1
Magnetic Tape		
	IO$_READLBLK	P1
	IO$_READPBLK	P1
	IO$_READVBLK	P1
	IO$_WRITELBLK	P1
	IO$_WRITEOF	P1
	IO$_WRITEPBLK	P1
	IO$_WRITEVBLK	P1
Mailbox		
	IO$_READLBLK	P1
	IO$_READPBLK	P1
	IO$_READVBLK	P1
	IO$_WRITELBLK	P1
	IO$_WRITEPBLK	P1
	IO$_WRITEVBLK	P1
Local Area Network (LAN)		
	IO$_READLBLK	P1,P5
	IO$_READPBLK	P1,P5
	IO$_READVBLK	P1,P5
	IO$_WRITELBLK	P1,P4,P5
	IO$_WRITEPBLK	P1,P4,P5
	IO$_WRITEVBLK	P1,P4,P5

4.4 64-Bit IO$_DIAGNOSE Function for SCSI class Drivers

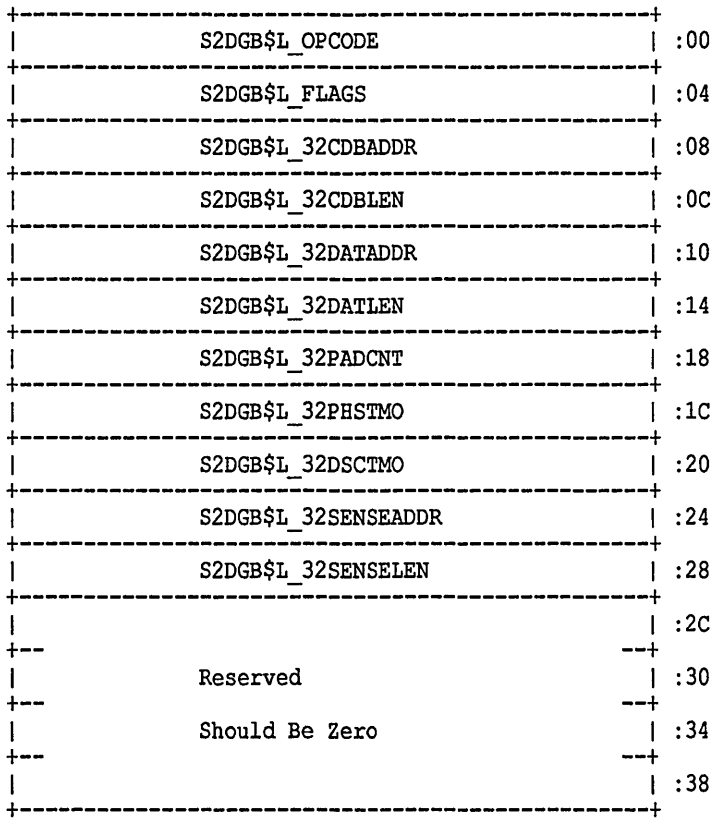
The $QIO IO$_DIAGNOSE function has been enhanced to support 64-bit addressing for the following SCSI class drivers: GKDRIVER, DKDRIVER, and MKDRIVER. This means that the virtual addresses specified within the S2DGB may now be 64-bit virtual addresses if the user application so requests it.

The $QIO IO$_DIAGNOSE parameters are still as follows:

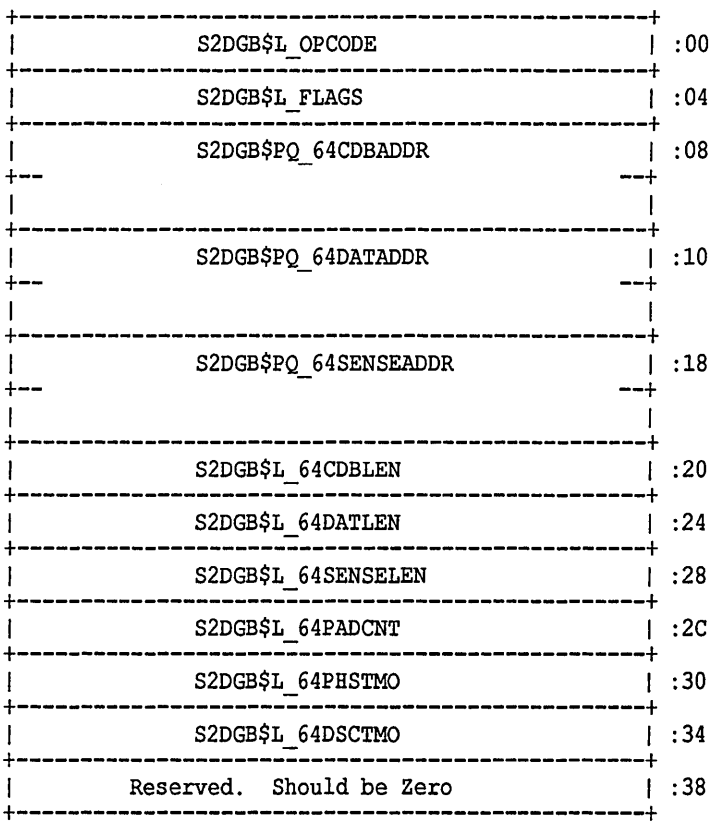
Parameter	Use
P1	S2DGB base address
P2	S2DGB length
P3	Reserved, should be zero
P4	Reserved, should be zero
P5	Reserved, should be zero
P6	Reserved, should be zero

The SCSI Diagnose Buffer (S2DGB) defined in STARLET now allows two formats, one for 32-bit addressing and one one for 64-bit addressing. The 32-bit format is identical to the one supported on OpenVMS Alpha Version 6.2. Example 4–1 shows the 32-bit S2DGB format. Example 4–2 shows the 64-bit S2DGB format.

Example 4–1 OpenVMS SCSI-2 Diagnose Buffer (S2DGB) 32-Bit Layout



Example 4–2 OpenVMS SCSI-2 Diagnose Buffer (S2DGB) 64-Bit Layout



A user application must specify which one of the two S2DGB formats is to be used by passing a format value in S2DGB$L_OPCODE. Specifically, S2DGB$L_OPCODE must be assigned a value of either OP_XCDB32 (= 1) to request 32-bit format, or OP_XCDB64 (= 2) to request 64-bit format. Once the value of OP_XCDB64 has been specified, the user application is obligated to use the 64-bit S2DGB format and, in particular, to use the 64-bit names for S2DGB fields as described below. Likewise, an opcode value of OP_XCDB32 obligates the user application to use the 32-bit names for the fields.

The correct length of the structure is defined by the constant S2DGB$K_XCDB32_LENGTH (value: 60-decimal), as well as by the constant S2DGB$K_XCDB64_LENGTH (value: 60-decimal).

The fields in the S2DGB are in the sections that follow. Whenever a field has two different names for the 32-bit and 64-bit cases, the 32-bit name is given first, and the 64-bit name is given after it in parentheses. Also, except for fields which contain addresses, all fields are unsigned longwords.

S2DGB$L_OPCODE

This field should contain either S2DGB$K_OP_XCDB32 or S2DGB$K_OP_XCDB64, depending on whether the user application intends to supply 32-bit virtual addresses or 64-bit virtual addresses, respectively, in the other fields of the S2DGB.

S2DGB$L_FLAGS

This field should contain the bit fields shown in the following table. Note that these bit definitions start at bit 0 and omit no bits. This is required for compatibility with the IO$_DIAGNOSE interface available in OpenVMS Alpha Version 6.1 and earlier.

S2DGB$V_READ	This bit should be 1 if the operation being performed is a read. If the operation is a write, this bit should be 0.
S2DGB$V_DISCPRIV	This bit should contain the DiscPriv bit value to be used in the IDENTIFY message sent with this operation. If S2DGB$V_TAGGED_REQ is 1, then this bit should be ignored. Note that this bit may be ignored by some ports.
S2DGB$V_SYNCHRONOUS	This bit is ignored since its value is beyond the control of the user in SCSI-2 drivers.
S2DGB$V_OBSOLETE1	This bit is ignored. In previous releases, it represented the disabling of command retries, which is now beyond the control of the user in SCSI-2 drivers.
S2DGB$V_TAGGED_REQ	When this bit is 1, the operation is processed as using tagged command queuing and S2DGB$V_ TAG should define the tag value to be used. When this bit is 0, the operation is processed without benefit of tagged command queuing. Ports that do not support tagged command queuing always behave as if this bit is 0. Note that some ports simulate untagged operations using appropriately tagged operations. one of the following coded constant values: S2DGB$K_SIMPLE indicates that the command is to be sent with the SIMPLE queue tag. S2DGB$K_ORDERED indicates that the command is to be sent with the ORDERED queue tag. S2DGB$K_EXPRESS indicates that the command is to be sent with the HEAD OF QUEUE queue tag. If S2DGB$V_TAGGED_REQ is 0, then this field is ignored. Ports that do not support tagged command queuing always ignore the S2DGB$V_TAG field and send all commands as untagged operations. Note that automatic contingent allegiance processing is not accessible through the IO$_DIAGNOSE function. Also, even though this is a 3-bit field, only 2 bits are currently being utilized. That is, the 3 constants above represent values, not bit positions.

S2DGB$V_AUTOSENSE	When this bit is 1, S2DGB$L_32SENSEADDR and S2DGB$L_32SENSELEN should contain a valid sense buffer address and length. If a CHECK CONDITION or COMMAND TERMINATED status is returned, REQUEST SENSE data will be returned in the buffer defined by S2DGB$L_ 32SENSEADDR and S2DGB$L_32SENSELEN. When S2DGB$V_AUTOSENSE is 0, the buffer described by S2DGB$L_32SENSEADDR and S2DGB$L_32SENSELEN is ignored. In such case, the class driver saves the autosense data in pool and returns it to the next IO$_DIAGNOSE, if and only if that IO$_DIAGNOSE has a REQUEST SENSE CDB. All other bits in S2DGB$L_FLAGS should be zero.

S2DGB$L_32CDBADDR (S2DGB$PQ_64CDBADDR)

This field should contain the 32-bit (or 64-bit) virtual address of the SCSI Command Data Block (CDB) to be sent to the target by this IO$_DIAGNOSE operation.

Note that S2DGB$L_32CDBADDR is a pointer to a longword, while S2DGB$PQ_ 64CDBADDR is a pointer to a quadword.

S2DGB$L_32CDBLEN (S2DGB$L_64CDBLEN)

This field should contain the number of bytes in the SCSI Command Data Block (CDB) to be sent to the target by this IO$_DIAGNOSE operation. (Legal values: 2 to 248. However, some ports may restrict CDBs to smaller lengths. Recommended values: 2 to 16.)

S2DGB$L_32DATADDR (S2DGB$PQ_64DATADDR)

This field should contain the 32-bit (or 64-bit) virtual address of the DATAIN or DATAOUT buffer to be used with this SCSI operation. If the CDB being sent to the target does not use a DATAIN or DATAOUT buffer, then this field should be zero.

Note that S2DGB$L_32DATADDR is a pointer to a longword, while S2DGB$PQ_ 64DATADDR is a pointer to a quadword.

S2DGB$L_32DATLEN (S2DGB$L_64DATLEN)

This field should contain the number of bytes in the DATAIN or DATAOUT buffer associated with this operation. If the CDB being sent to the target does not use a DATAIN or DATAOUT buffer, then this field should be zero. (Legal values: 0 to UCB$L_MAXBCNT. Recommended values: 0 to 65,536. All ports are required to support at least 65,536 byte data transfers.)

S2DGB$L_32PADCNT (S2DGB$L_64PADCNT)

This field should contain the number of padding DATAIN or DATAOUT bytes required by this operation. (Legal values: 0 to the maximum number of bytes in a disk block on this system minus one. Current legal values: 0 to 511.))

S2DGB$L_32PHSTMO (S2DGB$L_64PHSTMO)

This field should contain the number of seconds that the port driver should wait for a phase transition to occur or for delivery of an expected interrupt. If S2DGB$V_ TAGGED_REQ is 1 or this field contains a 0 or 1, then the current phase transition timeout setting will not be changed. (Legal values: 0 to 300 {5 minutes}.)

S2DGB$L_32DSCTMO (S2DGB$L_64DSCTMO)

This field should contain the number of seconds that the port driver should wait for a disconnected transaction to reconnect. If S2DGB$V_TAGGED_REQ is 1 or this field contains a 0 or 1, then the current disconnect timeout setting will not be changed. (Legal values: 0 to 65,535 {about 18 hours}.)

S2DGB$L_32SENSEADDR (S2DGB$PQ_64SENSEADDR)

If S2DGB$V_AUTOSENSE is 1, then this field should contain the 32-bit (or 64-bit) virtual address of the sense buffer to be used by this SCSI operation. If S2DGB$V_AUTOSENSE is 0, this field will be ignored.

Note that S2DGB$L_32SENSEADDR is a pointer to a longword, while S2DGB$PQ_64SENSEADDR is a pointer to a quadword.

S2DGB$L_32SENSELEN (S2DGB$L_64SENSELEN)

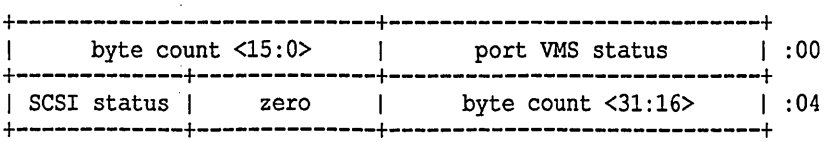
If S2DGB$V_AUTOSENSE is 1, then this field should contain the number of bytes in the sense buffer associated with this operation. (Legal values: 0 to 255. Note: a value of 0 instructs the class driver to discard any sense data received. Recommended value: 18. Some ports may restrict the number of sense bytes to 18.) If S2DGB$V_AUTOSENSE is 0, this field will be ignored.

4.4.1 64-bit S2DGB Example

The following example shows how to set up a 64-bit S2DGB:

```
#include <s2dgbdef.h>                                   /* Define S2DGB   */
#include <far_pointers.h>                               /* Define VOID_PQ */

   S2DGB diag_desc;

   /* Set up some default S2DGB descriptor values */

   diag_desc.s2dgb$l_opcode = OP_XCDB64                 /* Use 64-bits  */
   diag_desc.s2dgb$l_flags =  (S2DGB$M_READ |           /* Flags*/
                       S2DGB$M_TAGGED_REQ |
                       S2DGB$M_AUTOSENSE);
   diag_desc.s2dgb$v_tag = S2DGB$K_SIMPLE;             /* SIMPLE que tag */
   diag_desc.s2dgb$pq_64cdbaddr = (VOID_PQ)(&cdb[0]);/* Command addr  */
   diag_desc.s2dgb$l_64cdblen =  6;                    /* Command length */
   diag_desc.s2dgb$pq_64dataddr = (VOID_PQ)(&buf[0]);/* Data addr      */
   diag_desc.s2dgb$l_64datlen = 20;                    /* Data length    */
   diag_desc.s2dgb$l_64padcnt = 0;                     /* Pad length     */
   diag_desc.s2dgb$l_64phstmo = 20;                    /* Phase timeout */
   diag_desc.s2dgb$l_64dsctmo = 10;                    /* Disc timeout  */
   diag_desc.s2dgb$pq_64senseaddr = (VOID_PQ)(&asn[0]);/* Autosense addr */
   diag_desc.s2dgb$l_64senselen = 255;                 /* Sense length  */
   diag_desc.s2dgb$l_reserved_1 = 0;                   /* Reserved     */
        .
        .
        .

   status = sys$qiow(0, target_chan, IO$_DIAGNOSE, &iosb, 0, 0,
            &diag_desc,  S2DGB$K_XCDB64_LENGTH,  0,  0,  0,  0);
```

If all parameters are valid, the class driver will invoke the necessary port functions to send the CDB, transfer the data, and return, save or discard sense data as defined by the input S2DGB. Upon completion, the return IOSB will have the following format:



The DKDRIVER, GKDRIVER, and MKDRIVER class drivers, which implement other QIO functions, might intermix other tagged requests with IO$_DIAGNOSE requests. The order in which requests are sent generally match the order in which requests are presented to the driver. An exception to this ordering occurs when the driver receives REQUEST SENSE for which autosense data previously has been recovered and stored. In this case, the IO$_DIAGNOSE will complete immediately and no command will be sent to the target.

The DKDRIVER, GKDRIVER, and MKDRIVER class drivers permit only one IO$_DIAGNOSE operation to be active (in the start I/O routine) at given time, except as described in the next paragraph. However, applications must single thread IO$_DIAGNOSE requests in order to properly detect the presence of sense data and send the required REQUEST SENSE command. This is consistent with the VAX IO$_DIAGNOSE behavior. For example, if three reads are issued with no waiting and the first read gets a CHECK CONDITION, the sense data will be discarded by the target when the second read arrives.

The DKDRIVER, GKDRIVER, and MKDRIVER drivers permit more than one IO$_DIAGNOSE operation to be active (in the start I/O routine) only when all active operations have the S2DGB$V_AUTOSENSE flag equal to 1. Upon encountering the first IO$_DIAGNOSE with S2DGB$V_AUTOSENSE equal to 0, the class driver will apply the restrictions described in the previous paragraph.

5

OpenVMS Alpha 64-Bit API Guidelines

This chapter describes the guidelines used to develop 64-bit friendly interfaces to support OpenVMS Alpha 64-bit virtual addressing. Application programmers who are developing their own 64-bit application programming interfaces might find this information useful.

Note that these are recommendations, not hard and fast rules. Some of the guidelines are examples of good programming practices, and many of them address style issues.

5.1 Quadword/Longword Argument Pointer Guidelines

Because OpenVMS Alpha 64-bit adressing support allows application programs to access data in 64-bit address spaces, pointers that are not 32-bit sign-extended values (64-bit pointers) will become more common within applications. Existing 32-bit APIs will continue to be supported, and the existence of 64-bit pointers creates some potential pitfalls that programmer must be aware of.

For instance, addresses that are not 32-bit sign-extended values may be inadvertently passed to a routine that can handle only a 32-bit address. Another dimension of this would be a new API that embeds 64-bit pointers in data structures. Such pointers might be restricted to point to 32-bit address spaces initially, residing within the new data structure as a sign-extended 32-bit value.

All routines should guard against programming errors where 64-bit addresses are being passed instead of 32-bit addresses. This type of checking is called sign-extension checking, which means that the address is checked to ensure that the upper 32 bits are all zeros or all ones, matching the value of bit #31. This checking can be performed at the routine interface that is imposing this restriction.

When defining a new routine interface, you must consider the ease of programming a call to the routine from a 32-bit source module. And you must consider calls written in all OpenVMS programming languages, not just those languages initially intending to support 64-bit addressing. To avoid promoting awkward programming practices for the 32-bit caller of a new routine, you must accommodate 32-bit callers as well as 64-bit callers.

Arguments passed by reference that are restricted to reside in a 32-bit address space (P0/P1/S0/S1) should have their reference addresses sign-extension checked.

The OpenVMS Calling Standard requires that 32-bit values passed to a routine be sign-extended to 64-bits before the routine is called. Therefore, the called routine always receives 64-bit values. A 32-bit routine cannot tell if its caller correctly called the routine with a 32-bit address, unless the reference to the argument is checked for sign-extension.

This sign-extension checking would also apply to the reference to a descriptor when data is being passed to a routine by descriptor.

The called routine should return the error status SS$_ARG_GTR_32_BITS if the sign-extension check fails.

Alternately, if you want the called routine to accept the data being passed in a 64-bit location without error and if the sign-extension check fails, the data can be copied by the called routine to a 32-bit address space. The 32-bit address space to which the routine copies the data can be local routine storage (that is, the current stack). If the data is copied to a 32-bit location other than local storage, memory leaks and reentrancy issues must be considered.

Note

Copying to the stack may not be acceptable in future releases of OpenVMS Alpha, because stacks are restricted to 32-bit address spaces only for the first release of OpenVMS 64-bit support. This restriction might be lifted in a future release. OpenVMS system services are examples of routines that rely on the stack being in a 32-bit address space.

When new routines are developed, pointers to code and all data pointers passed to the new routines, should be accommodated in 64-bit address spaces where possible. This is desirable even if the data is a routine or is typically considered "static data", which the programmer, compiler, or linker would not naturally put in a 64-bit address space in this release. When code and static data is supported in 64-bit address spaces, this routine should not need additional changes.

32-bit descriptor arguments should be validated to be 32-bit descriptors.

Routines that accept descriptors should test the field that allow you to distinguish the 32-bit and 64-bit descriptor forms. If a 64-bit descriptor is received, the routine should return an error.

Avoid passing pointers by reference.

If passing a pointer by reference is necessary, as with certain memory management routines, the pointer should be defined to be 64-bit wide.

Mixing 32-bit and 64-bit pointers can cause programming errors when the caller incorrectly passes a 32-bit wide pointer by reference when a 64-bit wide pointer is expected.

If the called routine reads a 64-bit wide pointer that was allocated only a longword by the programmer, the wrong address could be used by the routine.

If the called routine returns a 64-bit pointer, and therefore writes a 64-bit wide address into a longword allocated by the programmer, data corruption can occur.

Existing routines that are passed pointers by reference require new interfaces for 64-bit support. Old routine interfaces would still be passed the pointer in a 32-bit wide memory location and the new routine interface would require that the pointer be passed in a 64-bit wide memory location. Keeping the same interface and passing it 64-bit wide pointers would break existing programs.

Example

The return virtual address in the new SYS$CRETVA_64 service. Virtual addresses created in P0 and P1 space are guaranteed to have only 32 bits of significance, however all 64 bits are returned. SYS$CRETVA_64 can also create address space in 64-bit space and thus return a 64-bit address. The value that is returned must always be 64 bits because a 64-bit address can be returned.

Memory allocation routines should return the pointer to the data allocated by value (that is, in R0), if possible. The C allocation routines, malloc, calloc, and realloc are examples of this.

New interfaces for routines that are not memory management routines should avoid defining output arguments to receive addresses. Problems will arise whenever a 64-bit subsystem allocates memory and then returns a pointer back to a 32-bit caller in an output argument. The caller may not be able to support or express a 64-bit pointer. Instead of returning a pointer to some data, the caller should provide a pointer to a buffer and the called routine should copy the data into the user's buffer.

A 64-bit pointer passed by reference should be defined in such a way that a call to the routine can be written in a 64-bit language or a 32-bit language. It should be clearly indicated that a 64-bit pointer is required to be passed by all callers.

Routines must not return 64-bit addresses unless they are specifically requested.

It is extremely important that routines which allocate memory and return an address to their callers always allocate 32-bit addressable memory, unless it is known absolutely that the caller is capable of handling 64-bit addresses. This is true for both function return values and output parameters. This rule prevents 64-bit addresses from "creeping in" to applications which do not expect them. As a result, programmers developing callable libraries should be particularly careful to follow this rule.

Suppose an existing routine returns the address of memory it has allocated, as the routine value. If the routine accepts an input parameter which in some way allows it to determine that the caller is 64-bit capable, it is safe to return a 64-bit address. Otherwise, it MUST continue to return 32-bit sign-extended addresses. In the latter case, a new version of the routine could be provided, which 64-bit callers could invoke instead of the existing version if they prefer that 64-bit memory be allocated.

Example: The routines in LIBRTL which manipulate string descriptors can be sure that a caller is 64-bit capable if the descriptor passed in is in the new 64-bit format. As a result, it is safe for them to allocate 64-bit memory for string data, in that case. Otherwise, they will continue to use only 32-bit addressable memory.

Avoid embedded pointers in data structures in public interfaces.

If embedded pointers are necessary for a new structure in a new interface, provide storage within the structure for a 64-bit pointer (quadword aligned). The called routine, which may have to read the pointer from the structure, simply reads all 64 bits.

If the pointer is constrained to be a 32-bit sign-extended address (for example, because the pointer will be passed to a 32-bit routine) a sign-extension check should be performed on the 64-bit pointer at the entrance to the routine. If the sign-extension check fails, the error status SS$_ARG_GTR_32_BITS may be returned to the caller, or the data found to reside in a 64-bit address space may be copied to a 32-bit address space.

The new structure should be defined in such a way that a 64-bit caller or a 32-bit caller do not contain awkward code. The structure should provide a quadword field for the 64-bit caller overlaid with two longword fields for the 32-bit caller. The first of these longwords is the 32-bit pointer field and the next is a MBSE (must be sign-extension) field. For most 32-bit callers, the MBSE field will be zero because the pointer will be a 32-bit process space address. The key here is to define the pointer as a 64-bit value and make it clear to the 32-bit caller that the full quadword must be filled in.

In the following example, both 64-bit and 32-bit callers would pass a pointer to the "block" structure and use the same function prototype when calling the function "routine". (Assume "data" is an unknown structure defined in another module.)

```
#pragma required_pointer_size save
#pragma required_pointer_size 32

typedef struct block {
    int blk$l_size;
    int blk$l_flags;
    union  {
#pragma required_pointer_size 64
        struct data *blk$pq_pointer;
#pragma required_pointer_size 32
        struct  {
            struct data *blk$ps_pointer;
            int blk$l_mbz;
            } blk$r_long_struct;
        } blk$r_pointer_union;
    } BLOCK;

#define blk$pq_pointer    blk$r_pointer_union.blk$pq_pointer
#define blk$r_long_struct blk$r_pointer_union.blk$r_long_struct
#define blk$ps_pointer    blk$r_long_struct.blk$ps_pointer
#define blk$l_mbz         blk$r_long_struct.blk$l_mbz

/* Routine accepts 64-bit pointer to the "block" structure */
#pragma required_pointer_size 64
int routine(struct block*);

#pragma required_pointer_size restore
```

For an existing 32-bit routine specifying an input argument, which is a structure that embeds a pointer, you can use a different approach to preserve the existing 32-bit interface. You can develop a 64-bit form of the data structure that is distinguished from the 32-bit form of the structure at run-time. Existing code that accepts only the 32-bit form of the structure should automatically fail when presented with the 64-bit form.

The structure definition for the new 64-bit structure should contain the 32-bit form of the structure (for example, with an SDL union). Including the 32-bit form of the structure allows the called routine to declare the input argument as a pointer to the 64-bit form of the structure and cleanly handle both cases.

Two different function prototypes must be provided for languages that provide type checking. The default function prototype should specify the argument as a pointer to the 32-bit form of the structure. The 64-bit form of the function prototype can be selected by defining a symbol, specified by documentation.

Descriptors:

The 64-bit versus 32-bit descriptor is an example of how this can be done.

Example: In the following example, the state of the symbol FOODEF64 selects the 64-bit form of the structure along with the proper function prototype. If the symbol FOODEF64 is undefined, the old 32-bit structure is defined and the old 32-bit function prototype is used.

The source module that implements the function foo_print would define the symbol FOODEF64 and be able to handle calls from 32-bit and 64-bit callers. The 64-bit caller would set the field foo64$l_mbmo to -1. Foo_print would test the field foo64$l_mbmo for -1 to determine if the caller used the 64-bit form of the structure or the 32-bit form of the structure.

```
#pragma required_pointer_size save
#pragma required_pointer_size 32

typedef struct foo {
    short int     foo$w_flags;
    short int     foo$w_type;
    struct data * foo$ps_pointer;
    } FOO;

#ifndef FOODEF64

/* Routine accepts 32-bit pointer to "foo" structure */
int foo_print(struct foo * foo_ptr);

#endif

#ifdef FOODEF64

typedef struct foo64 {
    union  {
        struct  {
            short int     foo64$w_flags;
            short int     foo64$w_type;
            int           foo64$l_mbmo;
#pragma required_pointer_size 64
            struct data * foo64$pq_pointer;
#pragma required_pointer_size 32
            } foo64$r_foo64_struct;
        FOO foo64$r_foo32;
        } foo64$r_foo_union;
    } FOO64;

#define foo64$w_flags    foo64$r_foo_union.foo64$r_foo64_struct.foo64$w_flags
#define foo64$w_type     foo64$r_foo_union.foo64$r_foo64_struct.foo64$w_type
#define foo64$l_mbmo     foo64$r_foo_union.foo64$r_foo64_struct.foo64$l_mbmo
#define foo64$pq_pointer foo64$r_foo_union.foo64$r_foo64_struct.foo64$pq_pointer
#define foo64$r_foo32    foo64$r_foo_union.foo64$r_foo32

/* Routine accepts 64-bit pointer to "foo64" structure */
#pragma required_pointer_size 64
int foo_print(struct foo64 * foo64_ptr);
```

```
#endif

#pragma required_pointer_size restore
```

In the previous example, if the structures "foo" and "foo64" will be used interchangeably within the same source module, you can eliminate the symbol FOODEF64. The routine foo_print would then be defined as follows:

```
int foo_print (void * foo_ptr);
```

Eliminating the FOODEF64 symbol allows 32-bit and 64-bit callers to use the same function prototype, however less strict type checking is then available during the C source compilation.

Context or User Data Arguments Should Be 64 Bits

Often, arguments or data structure fields are intended to convey context back to the original caller when an AST is delivered or when an operation completes. Currently, a longword is the common size for these arguments. Examples include the AST parameter arguments to many services or the user context field in RMS data structures. These fields should never be shorter than a quadword. Applications must be able to use a 64-bit address as a context argument.

5.2 Alpha/VAX Guidelines

Only address, size, and length arguments should be passed as quadwords by value.

Arguments passed by value are restricted to longwords on VAX. To be compatible with VAX APIs, quadword arguments should be passed by reference instead of by value. However, addresses, sizes and lengths are examples of arguments which, because of the architecture, could logically be longwords on OpenVMS VAX and quadwords on OpenVMS Alpha.

Even if the API will not be available on OpenVMS VAX, this guideline should still be followed for consistency across all APIs.

Avoid pagelets, pages, VBNs, and LBNs.

Arguments such as lengths and offsets should be represented in units that are page size independent, such as bytes.

A pagelet is an awkward unit. It was invented for compatibility with VAX and is used on OpenVMS Alpha within OpenVMS VAX compatible interfaces. A pagelet is equivalent in size to a VAX page and should not be considered a page size independent unit, because it is often confused with a CPU-specific page on Alpha.

Example: Length_64 argument in EXPREG_64 is passed as a quadword byte count by value.

Naturally align all data passed by reference.

The called routine should specify to the compiler that arguments are aligned, and the compiler can perform more efficient load and store sequences. If the data is not naturally aligned, users will experience performance penalties.

If the called routine executes incorrectly because the data passed by reference is not naturally aligned, the routine should do explicit checking and return an error if not aligned. For example, if a load/locked, store/conditional is being done internally in the routine on the data; and the data is not aligned, the load/locked, store/conditional will not work properly.

Specify the arguments in a consistent order.

Specify the argument order as follows:

1. Input arguments
2. Output arguments
3. Optional input arguments
4. Optional output arguments

Not only is the order of the arguments easier to remember, this order more efficiently uses the Alpha calling standard's register arguments in the case where more than 6 arguments can be specified. Every argument after the sixth argument is presented to the called routine on the stack. If optional arguments are not specified, memory references can be saved.

Do not use bytes and words.

Although this is not necessarily a guideline, it is worth noting.

Arguments that have less than or equal to 16 bits of significant data should be longwords or quadwords, not words or bytes. Reading and writing bytes and words from memory on Alpha systems is slower and can create word-tearing and synchronization problems.

Example: Return_prot in the SYS$SET_PRT_64 service. The returned protection is less than 32-bits, yet the argument is defined to be a longword.

5.3 Style Guidelines

Avoid documenting arguments passed by value as optional, unless they are at the end of the argument list.

This is a problem because if the argument is omitted, the value zero will be passed. Simply specifying that zero is the default value is clearer than specifying it as optional. Also, there is no such thing as an optional argument passed by value in the middle of an argument list.

An example of this problem is in the EFN argument to $QIO. In the documentation, it is listed as optional, yet if you do not specify it in the MACRO-32 macro, a zero is passed to the service. EFN #0 is used when the programmer might have thought that no EFN is used. Another instance where the programmer explicitly specified EFN #0 may trip over the inadvertent use of EFN #0.

Keep similar arguments in the same relative order between routines within the same API.

Example:

SYS$EXPREG_64 (region_id,length,acmode,flags,return_va,return_length)
SYS$CRETVA_64 (region_id,start_va,length,acmode,flags,return_va,return_length)

Don't add null arguments to artificially keep arguments in matching argument positions.

Avoid routine names that are too short and compact.

Routine names that are too long are also not recommended. Watch out for actual limits on name lengths. If you must abbreviate routine names, be consistent so the caller can remember abbreviations for the same word in different routines.

Example 1: $CRMPSC is too short. SYS$CREATE_AND_MAP_GLOBAL_PAGE_FILE_SECTION is too long. SYS$CRMPSC_GPFILE_64 is okay.

Example 2: The spelling of $SETIMR, $CANTIM, $GETTIM, $SETIME, etc., is difficult to remember because time and timer are abbreviated differently.

Use the suffix "_64" when appropriate

For system services, this suffix will be used for routines which accept 64-bit addresses by reference. For promoted routines, this distinguishes the 64-bit capable version from its 32-bit counterpart, and for new routines, it is a visible reminder that a 64-bit wide address cell will be read/written. This is also used when a structure is passed which contains an embedded 64-bit address, IF the structure is not self-identifying as a 64-bit structure. Hence, a routine name need not include "_64" simply because it receives a 64-bit decriptor. Remember that passing an arbitrary value by reference does not mean the suffix is required; passing a 64-bit address by reference does.

This practice is recommended for other routines as well.

Examples:

SYS$EXPREG_64(region_id_64, length_64, acmode, return_va_64, return_length_64)
SYS$CMKRNL_64(routine_64, quad_arglst_64)

Don't pass two different types of data in the same argument, and avoid flags that determine how to interpret arguments or which arguments must be specified.

This is confusing to the caller, as well as the documentation writer and the code maintainer.

Example: Look at the documentation for $CRMPSC!

5.4 Promoting an API from a 32-Bit API to a 64-Bit API

Promoting an API to support 64-bit addressing should not be viewed as an opportunity to improve the 32-bit design or add new functionality. Calling a routine within the new 64-bit API should be an easy programming task.

64-bit routines should accept 32-bit forms of structures as well as 64-bit forms.

To make it easy to modify calls to an API, the 32-bit form of a structure should be accepted by the interface as well as the 64-bit form.

Example: If the 32-bit API passed information by descriptor, the new interface should pass the same information by descriptor.

64-bit routines should provide the same functionality as the 32-bit routines.

An application currently calling the 32-bit API should be able to completely upgrade to calling the 64-bit API without having to preserve some of the old calls to the old 32-bit API just because the new 64-bit API is not a functional superset of the old API.

Example: SYS$EXPREG_64 works for P0, P1 and P2 process space. Callers can replace all calls to SYS$EXPREG since SYS$EXPREG_64 is a functional superset of $EXPREG.

5.5 No new 64-bit MACRO-32 macros are available for system services.

The MACRO-32 caller will have to use the new AMACRO built-in EVAX_CALLG_64.

5.6 Example of a 32-bit routine and a 64-bit routine

The following example illustrates a 32-bit routine interface that has been promoted to support 64-bit addressing. It handles several of the issues addressed in the guidelines.

The C function declaration for an old system service SYS$CRETVA looks like the following:

```
#pragma required_pointer_size save
#pragma required_pointer_size 32
int sys$cretva (
        struct _va_range * inadr,
        struct _va_range * retadr,
        unsigned int       acmode);
#pragma required_pointer_size restore
```

The C function declaration for a new system service SYS$CRETVA_64 looks like the following:

```
#pragma required_pointer_size save
#pragma required_pointer_size 64
int sys$cretva_64 (
        struct _generic_64 * region_id_64,
        void *               start_va_64,
        unsigned __int64     length_64,
        unsigned int         acmode,
        void **              return_va_64,
        unsigned __int64 *   return_length_64);
#pragma required_pointer_size restore
```

The new routine interface for SYS$CRETVA_64 corrects the embedded pointers within the "_va_range" structure, passes the 64-bit region_id_64 argument by reference and passes the 64-bit length_64 argument by value.

6

OpenVMS Alpha Tools and Utilities That Support 64-Bit Addressing

This chapter briefly describes the following OpenVMS Alpha tools that have been enhanced to support 64-bit virtual addressing.

- OpenVMS Debugger
- System-code debugger
- XDELTA
- Watchpoint utility
- SDA
- LIB$ and CVT$ Facilities of the OpenVMS Run-Time Library

6.1 OpenVMS Debugger

On OpenVMS Alpha systems, the Debugger can access the extended memory made available by 64-bit addressing support. You can examine and manipulate data in the complete 64-bit address space.

You can examine a variable as a quadword by using the new option Quad, which is on the Typecast menu of both the Monitor pull-down menu and the Examine dialog box.

The default type for the debugger is longword, which is appropriate for debugging 32-bit applications. It might be advisable to change the default type to quadword for debugging applications that utilize the 64-bit address space. To do this, the SET TYPE QUADWORD command.

Note that hexadecimal addresses are now 16-digit numbers on Alpha. For example,

```
DBG> EVALUATE/ADDRESS/HEX %hex 000004A0
00000000000004A0DBG>
```

For more information about using the OpenVMS Debugger, see the *OpenVMS Debugger Manual.*

6.2 OpenVMS Alpha System-Code Debugger

The OpenVMS Alpha system-code debugger accepts 64-bit addresses and uses full 64-bit addresses to retrieve information.

6.3 XDELTA

For more information about 64-bit addressing support for Delta/XDelta, see the *OpenVMS Delta/XDelta Debugger Manual.*

6.4 LIB$ and CVT$ Facilities of the OpenVMS Run-Time Library

For more information about 64-bit addressing support for the LIB$ and CVT$ facilities of the OpenVMS RTL library, refer to the *OpenVMS RTL Library (LIB$) Manual.*

6.5 Watchpoint Utility

The WATCHPOINT utility is a debugging tool that maintains a history of modification that are made to a particular location in shared system space by setting watchpoints on 64-bit addresses. It watches any system address, whether in S0, S1, or S2 space.

A $QIO interface to the WATCHPOINT supports 64-bit addresses. The WATCHPOINT command interpreter (WP) issues $QIO request to the WATCHPOINT driver (WPDRIVER) from commands that follow the standard rules of DCL grammar.

Commands may be entered at the watchpoint> prompt to set, delete, and obtain information from watchpoints. Before invoking the WATCHPOINT command interpreter (WP), or loading the WATCHPOINT driver one must first set the SYSGEN MAXBUF dynamic parameter to 64000. This can be accomplished as follows:

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN> SET MAXBUF 64000
SYSGEN> WRITE ACTIVE
SYSGEN> WRITE CURRENT
SYSGEN> EXIT
```

Before invoking the WATCHPOINT command interpreter (WP), the WATCHPOINT driver (WPDRIVER), must first be installed with SYSMAN. This can be accomplished as follows:

```
$ RUN SYS$SYSTEM:SYSMAN
SYSMAN> IO CONNECT WPA0/DRIVER=SYS$WPDRIVER/NOADAPTER
SYSMAN> EXIT
```

The WATCHPOINT command interpreter (WP) can then be invoked with the command:

```
$ RUN SYS$SYSTEM:WP
```

Commands may then be entered at the watchpoint> prompt to set, delete,and obtain information from Watchpoints.

Some of the WP help screens as well as the output to the WP Utility are best viewed using a terminal with a character width of 132. This can be accomplished as follows:

```
$ SET TERM/WID=132
```

6.6 SDA

For more information about using SDA 64-bit addressing support, see the *OpenVMS Alpha System Dump Analyzer Utility Manual.*

7

DEC C RTL Support for 64-Bit Addressing

This chapter describes the 64-bit addressing support provided by the DEC C run-time library on OpenVMS Alpha Version 7.0 systems and higher.

The DEC C run-time library includes the following features in support of 64-bit pointers:

- Guaranteed binary and source compatibility of existing programs
- No impact on applications that are not modified to exploit 64-bit support
- Enhanced memory allocation routines that allocate 64-bit memory
- Widened function parameters to accommodate 64-bit pointers
- Dual implementations of functions that need to know the pointer size used by the caller
- New information available to the DEC C Version 5.2 compiler or higher to seamlessly call the correct implementation
- Ability to explicitly call either the 32-bit or 64-bit form of functions for applications that mix pointer sizes
- A single shareable image for use by 32-bit and 64-bit applications

7.1 Using the DEC C Run-Time Library

The DEC C Run-Time library on OpenVMS Alpha Version 7.0 systems and higher can generate and accept 64-bit pointers. Functions that require a second interface to be used with 64-bit pointers reside in the same object libraries and shareable images as their 32-bit counterparts. No new object libraries or shareable images are introduced. Using 64-bit pointers does not require changes to your link command or link options files.

The DEC C 64-bit environment allows an application to use both 32-bit and 64-bit addresses. For more information about how to manipulate pointer sizes, see the /POINTER_SIZE qualifier and `#pragma pointer_size` and `#pragma required_pointer_size` preprocessor directives in the DEC C User's Guide.

The /POINTER_SIZE qualifier requires you to specify a value of 32 or 64. This value is used as the default pointer size within the compilation unit. As an application programmer, you can compile one set of modules using 32-bit pointers and another set using 64-bit pointers. Care must be taken when these two separate groups of modules call each other.

Use of the /POINTER_SIZE qualifier also influences the processing of DEC C RTL header files. For those functions that have a 32-bit and 64-bit implementation, specifying /POINTER_SIZE enables function prototypes to access both functions, regardless of the actual value supplied to the qualifier. In addition, the value

specified to the qualifier determines the default implementation to call during that compilation unit.

The #pragma pointer_size and #pragma required_pointer_size preprocessor directives can be used to change the pointer size in effect within a compilation unit. You can default pointers to 32-bit pointers and then declare specific pointers within the module as 64-bit pointers. You would also need to specifically call the _malloc64 form of malloc to obtain memory from the 64-bit memory area.

7.2 Obtaining 64-bit Pointers to Memory

The DEC C RTL has many functions that return pointers to newly allocated memory. In each of these functions, the application owns the memory pointed to and is responsible for freeing that memory.

Functions that allocate memory are:

```
malloc
calloc
realloc
strdup
```

Each of these functions have a 32-bit and 64-bit implementation. When the /POINTER_SIZE qualifier is used, the following functions can also be called:

```
_malloc32, _malloc64
_calloc32, _calloc64
_realloc32, _realloc64
_strdup32, _strdup64
```

When /POINTER_SIZE=32 is specified, all malloc calls default to _malloc32.

When /POINTER_SIZE=64 is specified, all malloc calls default to _malloc64.

Regardless of whether the application calls a 32-bit or 64-bit memory allocation routine, there is still a single free function. This function accepts either pointer size.

Note that the memory allocation functions are the only ones that return pointers to 64-bit memory. All DEC C RTL structure pointers returned to the calling application (such as a FILE, WINDOW, or DIR) are always 32-bit pointers. This allows both 32-bit and 64-bit callers to pass these structure pointers within the application.

7.3 DEC C Header Files

The header files distributed with DEC C Version 5.2 and higher support 64-bit pointers. Each function prototype whose signature contains a pointer is constructed to indicate the size of the pointer accepted.

A 32-bit pointer can be passed as an argument to functions that accept either a 32-bit or 64-bit pointer for that argument.

A 64-bit pointer, however, cannot be passed as an argument to a function that accepts a 32-bit pointer. Attempts to do this are diagnosed by the compiler with a MAYLOSEDATA message. The diagnostic message IMPLICITFUNC means the compiler can do no additional pointer-size validation for calls to that function. If this function is a DEC C RTL function, refer to the reference section of this manual for the name of the header file that defines that function.

You might find the following pointer-size compiler diagnostics useful:

- %CC-IMPLICITFUNC

  A function prototype was not found before using the specified function. The compiler and run-time system rely on prototype definitions to detect incorrect pointer-size usage. Failure to include the proper header files can lead to incorrect results and/or pointer truncation.

- %CC-MAYLOSEDATA

  A truncation is necessary to do this operation. The operation could be passing a 64-bit pointer to a function that does not support a 64-bit pointer in the given context. Or it could be a function returning a 64-bit pointer to a calling application that is trying to store that return value in a 32-bit pointer.

- %CC-MAYHIDELOSS

  This message (when enabled) helps expose real MAYLOSEDATA messages that are being suppressed because of a cast operation.

7.4 Functions Affected

The DEC C RTL shipped with OpenVMS Alpha Version 7.0 accommodates applications that use only 32-bit pointers, only 64-bit pointers, or combinations of both. To use 64-bit memory, you must, at a minimum, recompile and relink an application. The amount of source code change required depends on the application itself, calls to other runtime libraries, and the combinations of pointer sizes used.

With respect to 64-bit pointer support, the functions in the DEC C RTL fall into four categories:

- Functions not impacted by choice of pointer size
- Functions enhanced to accept either pointer size
- Functions having a 32-bit and 64-bit implementation
- Functions that accept only 32-bit pointers

From an application developer's perspective, the first two types of functions are the easiest to use in either a single or mixed-pointer mode.

The third type requires no modifications when used in a single-pointer compilation, but might require source code changes when used in a mixed-pointer mode.

The fourth type requires careful attention whenever 64-bit pointers are used.

7.4.1 No Pointer-Size Impact

The choice of pointer-size has no impact on a function if its prototype contains no pointer-related parameters or return values. The mathematical functions are good examples of this.

Even some functions in this category that do have pointers in their prototype are not impacted by pointer size. For example, `strerror` has the prototype:

```
char * strerror (int error_number);
```

This function returns a pointer to a character string, but this string is allocated by the DEC C RTL. As a result, to support both 32-bit and 64-bit applications, these types of pointers are guaranteed to fit in a 32-bit pointer.

7.4.2 Functions Accepting Both Pointer Sizes

The Alpha architecture supports 64-bit pointers. The OpenVMS Alpha calling standard specifies that all arguments are actually passed as 64-bit values. Before OpenVMS Alpha Version 7.0, all 32-bit addresses passed to procedures were sign-extended into this 64-bit parameter. The called function declared the parameters as 32-bit addresses, which caused the compiler to generate 32-bit instructions (such as LDL) to manipulate these parameters.

Many functions in the DEC C RTL are enhanced to receive the full 64-bit address. For example, consider `strlen`:

```
size_t strlen (const char *string);
```

The only pointer in this function is the character-string pointer. If the caller passes a 32-bit pointer, the function works with the sign-extended 64-bit address. If the caller passes a 64-bit address, the function works with that address directly.

The DEC C RTL continues to have only a single entry point for functions in this category. There are no source-code changes required to add any of the four pointer-size options for functions of this type. The OpenVMS documentation refers to these functions as 64-bit friendly.

7.4.3 Functions with Two Implementations

There are many reasons why a function might need two implementations— one for 32-bit pointers, the other for 64-bit pointers. Some of these reasons include:

- The pointer size of the return value is the same size as the pointer size of one of the arguments. If the argument is 32-bits, the return value is 32-bits. If the argument is 64-bits, the return value is 64-bits.
- One of the arguments is a pointer to an object whose size is pointer-size sensitive. To know how many bytes are being pointed to, the function must know if the code was compiled in 32-bit or 64-bit pointer-size mode.
- The function returns the address of dynamically allocated memory. The memory is allocated in 32-bit space when compiled for 32-bit pointers, and is allocated in 64-bit space when compiled for 64-bit pointers.

From the application developer's point of view, there are three function prototypes for each of these functions. The `<string.h>` header file contains many functions whose return value is dependent upon the pointer size used as the first argument to the function call. For example, consider the `memset` function. The header file defines three entry points for this function:

```
void * memset   (void *memory_pointer, int character, size_t size);
void *_memset32 (void *memory_pointer, int character, size_t size);
void *_memset64 (void *memory_pointer, int character, size_t size);
```

The first prototype is the function that your application would currently call if using this function. The compiler changes a call to `memset` into a call to either `_memset32` when compiled /POINTER_SIZE=32, or `_memset64` when compiled /POINTER_SIZE=64.

You can override this default behavior by directly calling either the 32-bit or the 64-bit form of the function. This accommodates applications using mixed pointer sizes, regardless of the default pointer size specified with the /POINTER_SIZE qualifier.

Note that if the application is compiled without specifying the /POINTER_SIZE qualifier, *neither* the 32-bit specific nor the 64-bit specific function prototypes are defined. In this case, the compiler automatically calls the 32-bit interface for all interfaces having dual implementations.

Table 7–1 shows the DEC C RTL functions that have dual implementations in support of 64-bit pointer size. When compiling with the /POINTER_SIZE qualifier, calls to the unmodified function names are changed to calls to the function interface that matches the pointer size specified on the qualifier.

Table 7–1 Functions with Dual Implementations

basename	malloc	strpbrk	wcsncat
bsearch	mbsrtowcs	strptime	wcsncpy
calloc	memccpy	strrchr	wcspbrk
catgets	memchr	strsep	wcsrchr
ctermid	memcpy	strstr	wcsrtombs
cuserid	memmove	strtod	wcsstr
dirname	memset	strtok	wcstok
fgetname	mktemp	strtol	wcstol
fgets	mmap	strtoll	wcstoul
fgetws	qsort	strtoq	wcswcs
fullname	realloc	strtoul	wmemchr
gcvt	rindex	strtoull	wmemcpy
getcap	strcat	strtouq	wmemmove
getcwd	strchr	tgetstr	wmemset
getname	strcpy	tmpnam	
gets	strdup	wcscat	
index	strncat	wcschr	
longname	strncpy	wcscpy	

7.4.4 Restricted to 32-Bit Pointers

Some functions in the DEC C RTL do not support 64-bit pointers. There are few of these. If you try to pass a 64-bit pointer to one of these functions, the compiler generates a %CC-W-MAYLOSEDATA warning. Applications compiled with /POINTER_SIZE=64 might need to be modified to avoid passing 64-bit pointers to these functions.

Table 7–2 shows the functions restricted to using 32-bit pointers. The DEC C RTL offers no 64-bit support for these functions. You must ensure that only 32-bit pointers are used with these functions.

Table 7–2 Functions restricted to 32-bit pointers

atexit	getopt	modf	setstate
execve	iconv	recvmsg	setvbuf
execvp	initstate	sendmsg	
frexp	ioctl	setbuf	

Table 7–3 shows functions that make callbacks to user-supplied functions as part of processing that function call. The callback procedures are not passed 64-bit pointers.

Table 7–3 Callbacks that Pass Only 32-Bit Pointers

from_vms	to_vms
ftw	tputs

7.5 Reading Header Files

This section introduces the pointer-size manipulations used in the DEC C RTL header files. Use the following examples to become more comfortable reading these header files and to help modify your own header files.

Examples

1.
```
:
#if __INITIAL_POINTER_SIZE ❶
#   if (__VMS_VER < 70000000) || !defined __ALPHA ❷
#      error " Pointer size usage not permitted before OpenVMS Alpha V7.0"
#   endif
#   pragma __pointer_size __save ❸
#   pragma __pointer_size 32 ❹
#endif
:
:
#if __INITIAL_POINTER_SIZE ❺
#   pragma __pointer_size 64
#endif
:
:
#if __INITIAL_POINTER_SIZE ❻
#   pragma __pointer_size __restore
#endif
:
```

All DEC C compilers that support the /POINTER_SIZE qualifier predefine the macro __INITIAL_POINTER_SIZE. The DEC C RTL header files take advantage of the ANSI rule that if a macro is not defined, it has an implicit value of 0.

The macro is defined as 32 or 64 when the /POINTER_SIZE qualifier is used. It is defined as 0 if the qualifier is not used. The statement shown as ❶ can be read as "if the user has specified either /POINTER_SIZE=32 or /POINTER_SIZE=64 on the command line".

DEC C Version 5.2 and higher is supported on many OpenVMS platforms. The lines shown as ❷ generate an error message if the target of the compilation is one that does not support 64-bit pointers.

A header file cannot assume anything about the actual pointer-size context in effect at the time the header file is included. Furthermore, the DEC C compiler offers only the __INITIAL_POINTER_SIZE macro and a mechanism to change the pointer size, but no way to determine the current pointer size.

All header files that have a dependency on pointer sizes are responsible for saving ❸, initializing ❹, altering ❺, and restoring ❻ the pointer-size context.

2.
```
:
#ifndef ___CHAR_PTR32 ❶
#   define ___CHAR_PTR32 1
    typedef char * ___char_ptr32;
    typedef const char * ___const_char_ptr32;
#endif
:
:
#if ___INITIAL_POINTER_SIZE
#   pragma ___pointer_size 64
#endif
:
:
#ifndef ___CHAR_PTR64 ❷
#   define ___CHAR_PTR64 1
    typedef char * ___char_ptr64;
    typedef const char * ___const_char_ptr64;
#endif
:
```

Some function prototypes need to refer to a 32-bit pointer when in a 64-bit pointer-size context. Other function prototypes need to refer to a 64-bit pointer when in a 32-bit pointer-size context.

DEC C binds the pointer size used in a typedef at the time the typedef is made. The typedef declaration of __char_ptr32 ❶ is made in a 32-bit context. The typedef declaration of __char_ptr64 ❷ is made in a 64-bit context.

3.
```
:
#if ___INITIAL_POINTER_SIZE
#   if (___VMS_VER < 70000000) || !defined ___ALPHA
#      error " Pointer size usage not permitted before OpenVMS Alpha V7.0"
#   endif
#   pragma ___pointer_size ___save
#   pragma ___pointer_size 32
#endif
:
❶
:
#if ___INITIAL_POINTER_SIZE ❷
#   pragma ___pointer_size 64
#endif
:
❸
:
int abs (int ___j); ❹
:
___char_ptr32 strerror (int ___errnum); ❺
:
```

Before declaring function prototypes that support 64-bit pointers, the pointer context is changed ❷ from 32-bit pointers to 64-bit pointers.

Functions restricted to 32-bit pointers are placed in the 32-bit pointer context section ❶ of the header file. All other functions are placed in the 64-bit context section ❸ of the header file.

Functions that have no pointer-size impact (❹ and ❺) are located in the 64-bit section. Functions that have no pointer-size impact, except for a 32-bit address return value ❺, are also in the 64-bit section, and use the 32-bit specific typedefs previously discussed.

4.
```
:
#if __INITIAL_POINTER_SIZE
#   pragma __pointer_size 64
#endif
:
:
#if __INITIAL_POINTER_SIZE == 32 (1)
#   pragma __pointer_size 32
#endif
:
char *strcat (char *__s1, __const_char_ptr64 __s2); (2)
:
#if __INITIAL_POINTER_SIZE
#   pragma __pointer_size 32
    :
    char *_strcat32  (char *__s1, __const_char_ptr64 __s2); (3)
    :
#   pragma __pointer_size 64
    :
    char *_strcat64  (char *__s1, const char *__s2); (4)
    :
#endif
:
```

This example shows declarations of functions that have both a 32-bit and 64-bit implementation. These declarations are located in the 64-bit section of the header file.

The normal interface to the function (2) is declared using the pointer size specified on the /POINTER_SIZE qualifier. Because the header file is in 64-bit pointer context and because of the statements at (1), the declaration at (2) is made using the same pointer size context as the /POINTER_SIZE qualifier.

The 32-bit specific interface (3) and the 64-bit specific interface (4) are declared in 32-bit and 64-bit pointer-size context, respectively.

8

MACRO–32 Programming Support for 64-Bit Addressing

This chapter describes the new 64-bit addressing support provided by the MACRO–32 compiler and associated components. The changes are primarily for argument passing and receiving and for address computations.

Note

When you use the latest version of the compiler, regardless of whether you use the 64-bit addressing features, you must also use the latest version of ALPHA$LIBRARY:STARLET.MLB. Make sure that the latest version is installed on your system and that the ALPHA$LIBRARY logical points to the correct directory.

8.1 Guidelines for 64-Bit Addressing

The following guidelines pertain to using 64-bit addressing in VAX MACRO code that is compiled for OpenVMS Alpha:

- Limit its use to code that you have ported to OpenVMS Alpha.

  For any new development on OpenVMS Alpha, Digital recommends the use of higher level languages.

- Make 64-bit addressing explicit in your code.

  The 64-bit addressing macros, directives, built-ins, and qualifier produce code that is more reliable and easier to maintain. (With an in-depth knowledge of the Alpha calling standard, you can make 64-bit argument list references without the new 64-bit address support, but such references are limited.)

8.2 New and Changed Components for 64-Bit Addressing

The new and changed components that provide MACRO–32 programming support for 64-bit addressing are shown in Table 8–1.

Table 8–1 New and Changed Components for 64-Bit Addressing

Component	Description
$SETUP_CALL64	New macro that initializes the call sequence
$PUSH_ARG64	New macro that does the equivalent of argument pushes

(continued on next page)

Table 8–1 (Cont.) New and Changed Components for 64-Bit Addressing

Component	Description
$CALL64	New macro that invokes the target routine
$IS_32BITS	New macro for checking the sign extension of the low 32 bits of a 64-bit value
QUAD=NO/YES	New parameter for page macros to support 64-bit virtual addresses
/ENABLE=QUADWORD	The QUADWORD parameter was extended to include 64-bit address computations
.CALL_ENTRY QUAD_ ARGS=TRUE ! FALSE	QUAD_ARGS=TRUE ! FALSE is a new parameter that indicates the presence (or absence) of quadword references to the argument list
.ENABLE QUADWORD /.DISABLE QUADWORD	The QUADWORD parameter was extended to include 64-bit address computations
EVAX_SEXTL	New built-in for sign extending the low 32 bits of a 64-bit value into a destination
EVAX_CALLG_64	New built-in to support 64-bit calls with variable-size argument lists
$RAB64 and $RAB64_STORE	New RMS macros for using buffers in 64-bit address space

8.3 Passing 64-Bit Values

The method that you use for passing 64-bit values depends on whether the size of the argument list is fixed or variable. These methods are described in the following sections.

8.3.1 Calls with a Fixed-Size Argument List

For calls with a fixed-size argument list, use the new macros, as shown in the steps in Table 8–2.

Table 8–2 Passing 64-Bit Values with a Fixed-Size Argument List

Step	Use...
Initialize the call sequence	$SETUP_CALL64
"Push" the call arguments	$PUSH_ARG64
Invoke the target routine	$CALL64

An example of using these macros follows. Note that the arguments are pushed in reverse order, which is the same way a 32-bit PUSHL instruction is used.

```
MOVL            8(AP), R5          ; fetch a longword to be passed
$SETUP_CALL64   3                  ; Specify three arguments in call
$PUSH_ARG64     8(R0)              ; Push argument #3
$PUSH_ARG64     R5                 ; Push argument #2
$PUSH_ARG64     #8                 ; Push argument #1
$CALL64         some_routine       ; Call the routine
```

The $SETUP_CALL64 macro initializes the state for a 64-bit call. It is required before $PUSH_ARG64 or $CALL64 can be used. If the number of arguments is greater than six, this macro creates a local JSB routine, which is invoked to perform the call. Otherwise, the argument loads and call are inline and very efficient. Note that the argument count specified in the $SETUP_CALL64 does *not* include a number sign (#). (The JSB routine facilitates proper placement of the stacked arguments, because the standard call sequence requires octaword alignment of the stack and of the stack arguments at the top of the octaword aligned stack.)

The inline option can be used to force a call with greater than six arguments to be done without a local JSB routine. However, there are restrictions on its use (see Appendix B).

The $PUSH_ARG64 macro moves the argument directly to the correct argument register or stack location. It is not actually a stack push, but it is the analog of the PUSHL instructions used in a 32-bit call.

The $CALL64 macro sets up the argument count register and invokes the target routine. If a JSB routine was created, it ends the routine. It reports an error if the number of arguments pushed does not match the count specified in $SETUP_CALL64. Both $CALL64 and $PUSH_ARG64 check that $SETUP_CALL64 has been invoked prior to their use.

8.3.1.1 Usage Notes for $SETUP_CALL64, $PUSH_ARG64, and $CALL64

Keep these points in mind when using $SETUP_CALL64, $PUSH_ARG64, and $CALL64:

- The arguments are read as aligned quadwords. To pass a longword from memory, move it to a register first, and then use that register in $PUSH_ARG64, as shown in the example in Section 8.3.1. Similarly, if you know the quadword you want to pass is unaligned, move the value to a register first. Also, keep in mind that indexed operands, such as (R4)[R0], will be evaluated using quadword indexing when used in $PUSH_ARG64.
- If the number of arguments is greater than six, so that a local JSB routine is created, no SP or AP references are allowed between the $SETUP_CALL64 and $CALL64. The $PUSH_ARG64 and $CALL64 macros do report uses of these registers in operands, but they are not allowed in other instructions in this range either, and cannot be flagged. To pass an AP- or SP-based argument in this case, move it to a register before the $SETUP_CALL64 invocation.
- If the number of arguments is greater than six, do not rely on values in registers above R15 surviving the $SETUP_CALL64 invocation. Use a nonscratch register as a temporary register instead. For example, suppose you want to pass a value from a stack location, and the call has more than six arguments. In this case, you need to move the value to a register. Rather than using a scratch register such as R28, use a VAX register, such as R0. If all the VAX registers are in use, use R13, R14, or R15.
- It is safe to use the scratch registers above R16 within the range between the $SETUP_CALL64 and the $CALL64. However, you must be careful not to use an argument register that has already been loaded. The argument registers are loaded in downward order, from R21 through R16. So, suppose a call passes six arguments. It is not safe to use R21 after the first $PUSH_ARG64, because that has loaded R21. The $PUSH_ARG64 macro checks for operands which refer to argument registers that have already been loaded. If any are

found, the compiler reports a warning. The safest approach is to use registers R22 through R28 when a temporary register is required.

Note

The $SETUP_CALL64, $PUSH_ARG64, and $CALL64 macros are intended to be used in an inline sequence. That is, you cannot branch into the middle of a $SETUP_CALL64/$PUSH_ARG64/$CALL64 sequence, nor can you branch around $PUSH_ARG64 macros, or branch out of the sequence to avoid the $CALL64.

For more information about $SETUP_CALL64, $PUSH_ARG64, and $CALL64, see Appendix B.

8.3.2 Calls with a Variable-Size Argument List

For calls with a variable-size argument list, use the new EVAX_CALLG_64 built-in, as shown in the following steps:

1. Create an in-memory argument list
2. Call a routine, passing the in-memory argument list, for example:

```
EVAX_CALLG_64 (Rn), routine
```

The argument list in the EVAX_CALLG_64 built-in is read as a series of quadwords, beginning with a quadword argument count.

8.4 Declaring 64-Bit Arguments

You can use QUAD_ARGS=TRUE, a new .CALL_ENTRY parameter, to declare the use of quadword arguments in a routine's argument list. With the presence of the QUAD_ARGS parameter, the compiler behaves differently when a quadword reference to the argument list occurs. First, it does not force argument-list homing, which such a reference normally requires. (An argument list containing a quadword value cannot be homed, because homing, by definition, packs the arguments into longword slots.) Second, "unaligned memory reference" will not be reported on these quadword references to the argument list.

Note that the actual code generated for the argument-list reference itself is not changed by the presence of the QUAD_ARGS clause, except when the reference is in a VAX quadword instruction, such as MOVQ. For the most part, QUAD_ARGS only prevents argument-list homing due to a quadword reference, and suppresses needless alignment messages. This suppression applies to both EVAX_ built-ins and VAX quadword instructions such as MOVQ.

For VAX quadword instructions, the QUAD_ARGS clause causes the compiler to read the quadword argument as it does for EVAX_ builtins—as an actual quadword. Consider the following example:

```
MOVQ    4(AP), 8(R2)
```

If the QUAD_ARGS clause is specified, MOVQ stores the entire 64 bits of argument 1 into the quadword at 8(R2). If the QUAD_ARGS clause is not specified, MOVQ stores the low longwords of arguments 1 and 2 into the quadword at 8(R2).

Note

Deferred mode argument list references, such as "@4(AP)", should be avoided in routines that use QUAD_ARGS=TRUE, because the result may not be what you expect. The indirection of this type of reference may require the effective address to be loaded from the argument list, if the argument is in memory. This is always performed as a longword load, *not* a quadword load. (If the argument is in a register, it need not be loaded.)

8.4.1 Usage Notes for QUAD_ARGS

Keep these points in mind when using QUAD_ARGS:

- AP-based quadword argument-list references look strange because they appear to overlap. You can improve this situation by defining symbolic names for the argument-list offsets, for example, FIRST_ARG, SECOND_ARG, and so forth. Users are encouraged to define meaningful symbolic names that describe the uses of the arguments to make the source code more readable. Alternatively, you can still use direct argument register references to refer to the first six arguments. Either way, it is useful to declare QUAD_ARGS to ensure that the argument list is not homed.
- Routines that share code must have the same setting for QUAD_ARGS. If they do not, the compiler will report a warning message.
- JSB routines cannot refer to their caller's argument list if the caller has QUAD_ARGS. References to AP within JSB routines require that the last CALL_ENTRY have its argument list homed. HOME_ARGS and QUAD_ARGS are mutually exclusive.
- QUAD_ARGS causes the $ARGn symbols, which the compiler places in the debug symbol table, to be defined as quadwords rather than longwords. These symbols allow easy access to received argument values and can be used in place of register numbers or stack offsets when debugging with the symbolic debugger.

8.5 Specifying 64-Bit Address Arithmetic

There are no explicit pointer-type declarations in MACRO–32. You can create a 64-bit pointer value in a register in a variety of ways. The most common are the EVAX_LDQ built-in for loading an address stored in memory and the MOVA*x* for getting the address of the specified operand.

After a 64-bit pointer value is in a register, an ordinary instruction will access the 64-bit address. The amount of data read from that address depends on the instruction used. Consider the following example:

```
MOVL     4(R1), R0
```

The MOVL instruction reads the longword at offset 4 from R1, regardless of whether R1 contains a 32- or 64-bit pointer.

However, certain addressing modes require the generation of arithmetic instructions to compute the effective address. For VAX compatibility, the compiler computes these as longword operations. For example, `4 + <1@33>` yields the value 4, because the shifted value exceeds 32 bits. If quadword mode was enabled, the upper bit would not be lost.

The /ENABLE=QUADWORD qualifier (and the corresponding .ENABLE QUADWORD and .DISABLE QUADWORD directives) were extended to affect address computations. Prior to these extensions, the QUADWORD options only affected the mode in which constant expression evaluations were performed.

By specifying /ENABLE=QUADWORD, addresses will be computed using quadword instructions, such as S*x*ADDQ and ADDQ. If you do not want quadword operations applied to an entire code module but only to certain sections, you can omit the /ENABLE=QUADWORD qualifier. Instead, use the .ENABLE QUADWORD and .DISABLE QUADWORD directives to enclose the sections where you want quadword operations performed.

There is no performance penalty when using /ENABLE=QUADWORD.

8.5.1 Dependence on Wrapping Behavior of Longword Operations

The compiler cannot use quadword arithmetic for all addressing computations, because existing code may rely on the wrapping behavior of the 32-bit operations. That is, code may perform addressing operations that actually overflow 32 bits, knowing that the upper bits are discarded. Doing the calculation in quadword mode will cause an incompatibility.

Before using /ENABLE to set quadword evaluation for an entire module, check the existing code for dependence on longword wrapping. There is no simple way to do this, but as a programming technique, it should be rare and may be called out in the code.

The following example shows the wrapping problem:

```
MOVAL   (R1)[R0], R2
```

Suppose R1 contains the value 7FFFFFFF, and R0 contains 1. The MOVAL instruction generates an S4ADDL instruction. The shift and add result exceeds 32 bits, but the stored result is the low 32 bits, sign extended.

If quad arithmetic were used (S4ADDQ), the true quad value would result, as shown in the following example:

```
S4ADDL  R0, R1, R2   =>  FFFFFFFF 80000003
S4ADDQ  R0, R1, R2   =>  00000000 80000003
```

The wrapping problem is not limited to indexed mode addressing. Consider the following example:

```
MOVAB   offset(R1), R0
```

If the symbol offset is not a compile-time constant, this instruction causes a value to be read from the linkage section and added (using an ADDL instruction) to the value in R1. Changing this to ADDQ may change the result if the value exceeds 32 bits.

8.6 Sign Extending and Checking

A new built-in, EVAX_SEXTL (sign extend longword), is available for sign extending the low 32 bits of a 64-bit value into a destination. This built-in makes explicit the sign extension of the low longword of the source into the destination.

EVAX_SEXTL takes the low 32 bits of the 64-bit value, fills the upper 32 bits with the sign extension (whatever is in bit 31 of the value), and writes the 64-bit result to the destination.

Neither operand must be a register. The following examples are all legal uses:

```
evax_sextl r1,r2
evax_sextl r1,(r2)
evax_sextl (r2), (r3)[r4]
```

A new macro, $IS_32BITS is available for checking the sign extension of the low 32 bits of a 64-bit value. It is described in Appendix B.

8.7 Alpha Instruction Built-ins

The compiler supports many Alpha instructions as built-ins. Many of these built-ins (available since the compiler first shipped with OpenVMS Alpha) can be used to operate on 64-bit quantities. The function of each built-in and its valid operands are documented in *Migrating to an OpenVMS AXP System: Porting VAX MACRO Code*. A full description of each Alpha instruction is documented in the *MACRO–64 Assembler for OpenVMS AXP Systems Reference Manual.*

8.8 Calculating Page-Size Dependent Values

A new parameter, QUAD=NO/YES, for supporting 64-bit virtual addresses is available for each of the page macros, shown in the following list:

- $BYTES_TO_PAGES
- $NEXT_PAGE
- $PAGES_TO_BYTES
- $PREVIOUS_PAGE
- $ROUND_RETADR
- $START_OF_PAGE

These macros provide a standard, architecture-independent means for calculating page-size dependent values. For more information about these macros, see *Migrating to an OpenVMS AXP System: Porting VAX MACRO Code.*

8.9 Creating and Using Buffers in 64-Bit Address Space

The $RAB and $RAB_STORE control block macros have been extended for creating and using data buffers in 64-bit address space. The 64-bit versions are named $RAB64 and $RAB64_STORE respectively. The rest of the RMS interface is restricted to 32 bits at this time. For more information about $RAB64 and $RAB64_STORE, see Chapter 3.

8.10 Coding for Moves Longer Than 64K Bytes

The MACRO–32 instructions MOVC3 and MOVC5 properly handle 64-bit addresses but the moves are limited to a 64K bytes length. This limitation is because MOVC3 and MOVC5 accept word-sized lengths.

For moves longer than 64K bytes, use OTS$MOVE3 and OTS$MOVE5. OTS$MOVE3 and OTS$MOVE5 accept longword-sized lengths. (LIB$MOVC3 and LIB$MOVC5 have the same 64K byte length restriction as MOVC3 and MOVC5.) An example of replacing MOVC3 with OTS$MOVE3 follows.

Code using MOVC3:

```
MOVC3       BUF$W_LENGTH(R5), (R6), OUTPUT(R7)    ; Old code, word length
```

The equivalent 64-bit code with longword length:

```
$SETUP_CALL64  3              ; Specify three arguments in call
EVAX_ADDQ      R7, #OUTPUT, R7
$PUSH_ARG64    R7             ; Push destination, arg #3
$PUSH_ARG64    R6             ; Push source, arg #2
MOVL           BUF$L_LENGTH(R5), R16
$PUSH_ARG64    R16            ; Push length, arg #1
$CALL64        OTS$MOVE3

MOVL           BUF$L_LENGTH(R5), R16
EVAX_ADDQ      R6, R16, R1    ; MOVC3 returns address past source
EVAX_ADDQ      R7, R16, R3    ; MOVC3 returns address past destination
```

Because MOVC3 clears R0, R2, R4, and R5, make sure that these side effects are no longer needed.

OTS$MOVE3 and OTS$MOVE5 are documented with other LIBOTS routines in the *OpenVMS RTL General Purpose (OTS$) Manual*.

A

64-Bit Example Program

This example program demonstrates the 64-bit region creation and deletion system services. It uses SYS$CREATE_REGION_64 to create a region and then uses SYS$EXPREG_64 to allocate virtual addresses within that region. The virtual address space and the region are deleted by calling SYS$DELETE_REGION_64.

```
/*
 ******************************************************************************
 *
 * Copyright © Digital Equipment Corporation, 1995 All Rights Reserved.
 * Unpublished rights reserved under the copyright laws of the United States.
 *
 * The software contained on this media is proprietary to and embodies the
 * confidential technology of Digital Equipment Corporation.  Possession, use,
 * duplication or dissemination of the software and media is authorized only
 * pursuant to a valid written license from Digital Equipment Corporation.
 *
 * RESTRICTED RIGHTS LEGEND   Use, duplication, or disclosure by the U.S.
 * Government is subject to restrictions as set forth in Subparagraph
 * (c)(1)(ii) of DFARS 252.227-7013, or in FAR 52.227-19, as applicable.
 *
 ******************************************************************************
/*
    This program creates a region in P2 space using the region creation
    service and then creates VAs within that region.  The intent is to
    demonstrate the use of the region services and how to allocate virtual
    addresses within a region.  The program also makes use of 64-bit
    descriptors and uses them to format return values into messages with the
    aid of SYS$GETMSG.

    To build and run this program type:

    $ CC/POINTER_SIZE=SHORT/STANDARD=RELAXED/DEFINE=("__NEW_STARLET=1") -
        REGIONS.C
    $ LINK REGIONS.OBJ
    $ RUN REGIONS.EXE
*/

#include    <descrip.h>             /* Descriptor Definitions              */
#include    <far_pointers.h>        /* Long Pointer Definitions            */
#include    <gen64def.h>            /* Generic 64-bit Data Type Definition */
#include    <iledef.h>              /* Item List Entry Definitions         */
#include    <ints.h>                /* Various Integer Typedefs            */
#include    <iosbdef.h>             /* I/O Status Block Definition         */
#include    <psldef.h>              /* PSL$ Constants                      */
#include    <ssdef.h>               /* SS$_ Message Codes                  */
#include    <starlet.h>             /* System Service Prototypes           */
#include    <stdio.h>               /* printf                              */
#include    <stdlib.h>              /* malloc, free                        */
#include    <string.h>              /* memset                              */
#include    <syidef.h>              /* $GETSYI Item Code Definitions       */
#include    <vadef.h>               /* VA Creation Flags and Constants     */
```

64-Bit Example Program

```
/*  Module-wide constants and macros.                                    */

#define     BUFFER_SIZE         132
#define     HW_NAME_LENGTH       32
#define     PAGELET_SIZE        512
#define     REGION_SIZE         128

#define     good_status(code)   ((code) & 1)

/*  Module-wide Variables                                                */

int
    page_size;

$DESCRIPTOR64 (msgdsc, "");

/*  Function Prototypes                                                  */

int get_page_size (void);
static void print_message (int code, char *string);

main (int argc, char **argv)
{
    int
        i,
        status;

    uint64
        length_64,
        master_length_64,
        return_length_64;

    GENERIC_64
        region_id_64;

    VOID_PQ
        master_va_64,
        return_va_64;

/*  Get system page size, using SYS$GETSYI.                              */

    status = get_page_size ();
    if (!good_status (status))
        return (status);

/*  Get a buffer for the message descriptor.                             */

    msgdsc.dsc64$pq_pointer = malloc (BUFFER_SIZE);
    printf ("Message Buffer Address = %016LX\n\n", msgdsc.dsc64$pq_pointer);

/*  Create a region in P2 space.                                         */

    length_64 = REGION_SIZE*page_size;
    status = sys$create_region_64 (
        length_64,                  /* Size of Region to Create         */
        VA$C_REGION_UCREATE_UOWN,   /* Protection on Region             */
        0,                          /* Allocate in Region to Higher VAs */
        &region_id_64,              /* Region ID                        */
        &master_va_64,              /* Starting VA in Region Created    */
        &master_length_64);         /* Size of Region Created           */
    if (!good_status (status))
    {
        print_message (status, "SYS$CREATE_REGION_64");
        return (status);
    }

    printf ("\nSYS$CREATE_REGION_64 Created this Region:  %016LX - %016LX\n",
        master_va_64,
        (uint64) master_va_64 + master_length_64 - 1);
```

```
/*  Create virtual address space within the region.                    */

    for (i = 0; i < 3; ++i)
    {
        status = sys$expreg_64 (
            &region_id_64,      /* Region to Create VAs In              */
            page_size,          /* Number of Bytes to Create            */
            PSL$C_USER,         /* Access Mode                          */
            0,                  /* Creation Flags                       */
            &return_va_64,      /* Starting VA in Range Created         */
            &return_length_64); /* Number of Bytes Created              */
        if (!good_status (status))
        {
            print_message (status, "SYS$EXPREG_64");
            return status;
        }
        printf ("Filling %016LX - %16LX with %0ds.\n",
            return_va_64,
            (uint64) return_va_64 + return_length_64 - 1,
            i);
        memset (return_va_64, i, page_size);
    }

/*  Return the virtual addresses created within the region, as well as the
    region itself.                                                     */

    printf ("\nReturning Master Region:  %016LX - %016LX\n",
        master_va_64,
        (uint64) master_va_64 + master_length_64 - 1);

    status = sys$delete_region_64 (
        &region_id_64,      /* Region to Delete                       */
        PSL$C_USER,         /* Access Mode                            */
        &return_va_64,      /* VA Deleted                             */
        &return_length_64); /* Length Deleted                         */

    if (good_status (status))
        printf ("SYS$DELETE_REGION_64 Deleted VAs Between:  %016LX - %016LX\n",
            return_va_64,
            (uint64) return_va_64 + return_length_64 - 1);
    else
    {
        print_message (status, "SYS$DELTE_REGION_64");
        return (status);
    }

/*  Return message buffer.                                             */

    free (msgdsc.dsc64$pq_pointer);
}

/*  This routine obtains the system page size using SYS$GETSYI.  The return
    value is recorded in the module-wide location, page_size.          */

int get_page_size ()
{
int
    status;

IOSB
    iosb;

ILE3
    item_list [2];

/* Fill in SYI item list to retrieve the system page size.             */
```

```
    item_list[0].ile3$w_length       = sizeof (int);
    item_list[0].ile3$w_code         = SYI$_PAGE_SIZE;
    item_list[0].ile3$ps_bufaddr     = &page_size;
    item_list[0].ile3$ps_retlen_addr = 0;
    item_list[1].ile3$w_length       = 0;
    item_list[1].ile3$w_code         = 0;

/* Get the system page size.                                              */

    status = sys$getsyiw (
                0,                              /* EFN                    */
                0,                              /* CSI address            */
                0,                              /* Node name              */
                &item_list,                     /* Item list              */
                &iosb,                          /* I/O status block       */
                0,                              /* AST address            */
                0);                             /* AST parameter          */

    if (!good_status (status))
    {
        print_message (status, "SYS$GETJPIW");
        return (status);
    }
    if (!good_status (iosb.iosb$w_status))
    {
        print_message (iosb.iosb$w_status, "SYS$GETJPIW IOSB");
        return (iosb.iosb$w_status);
    }

    return SS$_NORMAL;
}

/*  This routine takes the message code passed to the routine and then uses
    SYS$GETMSG to obtain the associated message text.  That message is then
    printed to stdio along with a user-supplied text string.              */

#pragma inline (print_message)
static void print_message (int code, char *string)
{
    msgdsc.dsc64$q_length = BUFFER_SIZE;
    sys$getmsg (
        code,                                       /* Message Code       */
        (unsigned short *) &msgdsc.dsc64$q_length,  /* Returned Length    */
        &msgdsc,                                    /* Message Descriptor */
        15,                                         /* Message Flags      */
        0);                                         /* Optional Parameter */
    *(msgdsc.dsc64$pq_pointer+msgdsc.dsc64$q_length) = '\0';
     printf ("Call to %s returned:  %s\n",
        string,
        msgdsc.dsc64$pq_pointer);
}
```

B

MACRO-32 Macros for 64-Bit Addressing

This appendix describes the MACRO-32 macros for manipulating 64-bit addresses and for checking the sign extension of the low 32 bits of 64-bit values.

These macros reside in the directory ALPHA$LIBRARY:STARLET.MLB (generally synonymous with SYS$LIBRARY:STARLET.MLB) and can be used by both application code and system code. The page macros have also been enhanced for 64-bit addresses. The support is provided by a new parameter, QUAD=NO/YES.

Note that you can use certain arguments to the macros described in this appendix to indicate register sets. To express a register set, list the registers, separated by commas, within angle brackets. For example:

```
<R1,R2,R3>
```

If the set contains only one register, the angle brackets are not required.

B.1 Macros for Manipulating 64-Bit Addresses

This section describes the following macros, designed to manipulate 64-bit addresses:

- $SETUP_CALL64
- $PUSH_ARG64
- $CALL64

$SETUP_CALL64

Initializes the call sequence.

Format

$SETUP_CALL64 arg_count, inline=true or false

Parameters

arg_count
The number of arguments in the call.

inline
Forces inline expansion, rather than creation of a JSB routine.

Description

This macro initializes the state for a 64-bit call. It *must* be used before using $PUSH_ARG64 and $CALL64.

If there are six or fewer arguments, the code is always inline.

By default, if there are more than six arguments, this macro creates a JSB routine which is invoked to perform the actual call. However, if the inline option is specified as inline=true, the code will be generated inline. This option should *only* be enabled if the code in which it appears has a fixed stack depth. A fixed stack depth can be assumed if no RUNTIMSTK or VARSIZSTK messages have been reported. Otherwise, if the stack alignment is not at least quadword, there might be many alignment faults in the called routine and in anything the called routine calls. The default behavior (inline=false) does not have this problem.

If there are more than six arguments, there can be no references to AP or SP between a $SETUP_CALL64 and the matching $CALL64, because the $CALL64 code may be in a separate JSB routine. In addition, temporary registers (R16 and above) may not survive the $SETUP_CALL64. However, they can be used *within* the range, except where of R16 through R21 interfere with the argument registers already set up. In such cases, higher temporary registers should be used instead.

Note

The $SETUP_CALL64, $PUSH_ARG64, and $CALL64 macros are intended to be used in an inline sequence. That is, you cannot branch into the middle of a $SETUP_CALL64/$PUSH_ARG64/$CALL64 sequence, nor can you branch around $PUSH_ARG64 macros, or branch out of the sequence to avoid the $CALL64.

$PUSH_ARG64

Does the equivalent of argument pushes for a call.

Format

$PUSH_ARG64 argument

Parameters

argument
The argument to be pushed.

Description

This macro pushes a 64-bit argument for a 64-bit call. The macro $SETUP_CALL64 *must* be used before you can use $PUSH_ARG64.

Arguments will be read as aligned quadwords. That is, $PUSH_ARG64 4(R0) will read the quadword at 4(R0), and push the quadword. Any indexed operations will be done in quadword mode.

To push a longword value from memory as a quadword, first move it into a register with a longword instruction, and then use $PUSH_ARG64 on the register. Similarly, to push a quadword value which you know is *not* aligned, move it to a temporary register first, and then use $PUSH_ARG64.

If the call contains more than six arguments, this macro checks for SP or AP references in the argument. If the call contains more than six arguments, SP references are not allowed, and AP references are only allowed if the inline option is used.

The macro also checks for references to argument registers that have already been set up for the current $CALL64. If it finds such references, a warning is reported to advise the user to be careful not to overwrite an argument register before it is used as the source in a $PUSH_ARG64.

The same checking is done for AP references when there are six or fewer arguments; they are allowed, but the compiler cannot prevent you from overwriting one before you use it. Therefore, if such references are found, an informational message is reported.

Note that if the operand uses a symbol whose name includes one of the strings R16 through R21, *not* as a register reference, this macro might report a spurious error. For example, if the invocation `$PUSH_ARG64 SAVED_R21` is made after R21 has been set up, this macro will unnecessarily report an informational message about overwriting argument registers.

Also note that $PUSH_ARG64 *cannot* be in conditional code. $PUSH_ARG64 updates symbols that keep track of argument count, and so forth. Attempting to write code that branches around a $PUSH_ARG64 in the middle of a $SETUP_CALL64/$CALL64 sequence will not work properly.

$CALL64

Invokes the target routine.

Format

$CALL64 call_target

Parameters

call_target
The routine to be invoked.

Description

This macro calls the specified routine, assuming $SETUP_CALL64 has been used to specify the argument count, and $PUSH_ARG64 has been used to push the quadword arguments. This macro checks that the number of pushes matches what was specified in the setup call.

The call target operand must not be AP- or SP-based.

B.2 Macro for Checking the Sign Extension

The macro in this section is used for checking the sign extension of the low 32 bits of a 64-bit value.

$IS_32BITS

Checks the sign extension of the low 32 bits of a 64-bit value and directs the program flow, based on the outcome of the check.

Format

$IS_32BITS quad_arg, leq_32bits, gtr_32bits, temp_reg=22

Parameters

quad_arg
A 64-bit quantity, either in a register or in an aligned quadword memory location.

leq_32bits
Label to branch to if quad_arg is a 32-bit sign-extended value.

gtr_32bits
Label to branch to if quad_arg is greater than 32-bits.

temp_reg=22
Register to use as a temporary register for holding the low longword of the source value—R22 is the default.

Description

$IS_32BITS checks the sign extension of the low 32 bits of a 64-bit value and directs the program flow, based on the outcome of the check.

Example

```
$is_32bits  R9, 10$
$is_32bits  4(R8), 20$, 30$, R28
```

In the first example, the compiler checks the sign extension of the low 32 bits of the 64-bit value at R9, using the default temporary register, R22. Depending on the type of branch and the outcome of the test, the program either branches or continues inline.

In the second example, the compiler checks the sign extension of the low 32 bits of the 64-bit value at 4(R8), using R28 as a temporary register, and, based on the check, branches to either 20$ or 30$.

Index

64-bit addresses and the debugger, 6–1
64-bit pointer support, 7–1
64-Bit Virtual Addressing
  definition, 1–1

A

Addresses
  passing 64-bit values, 8–2, B–1
  specifying 64-bit computing, 8–5
Addressing guidelines
  64-bit, 8–1
Argument list
  fixed-size, 8–2
  suppressing homing, 8–4
  variable-size, 8–4
Arguments
  declaring quadword, 8–4
Assembly language instructions
  Alpha built-ins, 8–7

B

Built-ins
  Alpha assembly language instructions, 8–7

C

$CALL64 macro, 8–1, B–3
  passing 64-bit values, 8–2
.CALL_ENTRY directive
  QUAD_ARGS parameter
    declaring 64-bit values, 8–1, 8–4

D

Debugger
  quadwords, 6–1
.DISABLE directive
  QUADWORD option, 8–1

E

.ENABLE directive
  QUADWORD option, 8–1
/ENABLE qualifier
  QUADWORD option, 8–1
EVAX_CALLG_64 built-in
  64-bit address support, 8–2, 8–4
EVAX_SEXTL built-in
  sign extension for 64-bit address support, 8–2, 8–6

I

Instructions
  compiler built-ins for Alpha assembly language, 8–7
$IS_32BITS macro
  checking sign extension, 8–1, 8–7, B–4

L

LIB$MOVC3 routine, 8–7
LIB$MOVC5 routine, 8–7
LIBOTS routines, 8–7

M

MOVC3 instruction, 8–7
MOVC5 instruction, 8–7

O

OTS$MOVE3 routine, 8–7
OTS$MOVE5 routine, 8–7

P

Page size
  calculations based on, 8–1, 8–7
  macro parameter for 64-bit addressing, 8–1, 8–7
Pointer-type declarations, 8–5

Pointers
  64-bit support, 7–1
$PUSH_ARG64 macro, 8–1, B–2
  passing 64-bit values, 8–2

Q

Quadword addresses
  computing, 8–5
Quadword arguments
  declaring, 8–4
  passing, 8–2

R

$RAB macro, 8–7
$RAB64 macro, 8–2, 8–7
$RAB64_STORE macro, 8–2, 8–7
$RAB_STORE macro, 8–7
RMS macros
  support for data buffers in 64-bit address space, 8–2, 8–7

S

$SETUP_CALL64 macro, 8–1, B–1
  passing 64-bit values, 8–2
Sign extension
  checking with $IS_32BITS macro, 8–7, B–4
  using EVAX_SEXTL built-in, 8–6