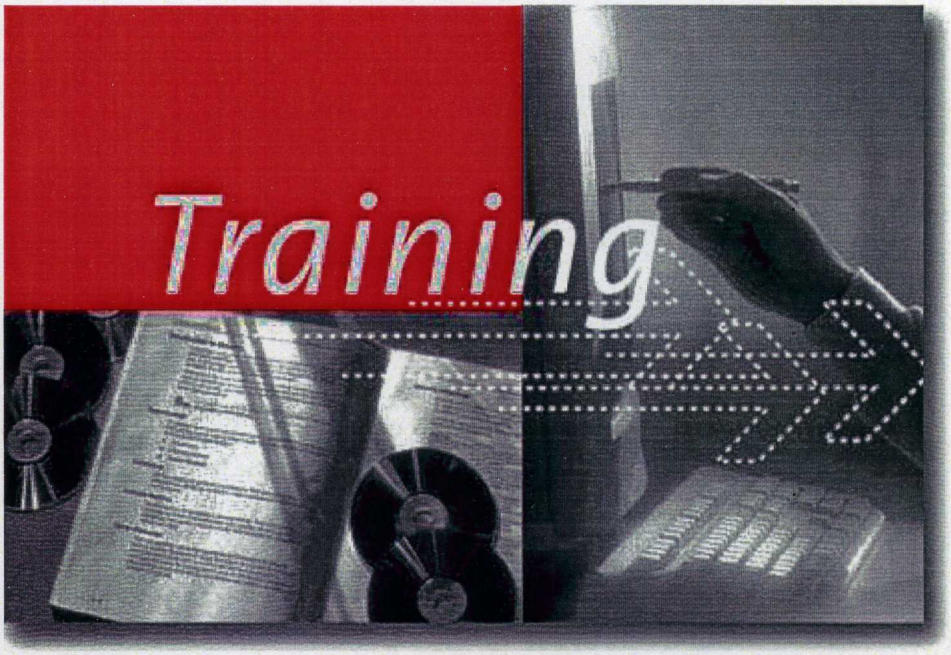
OpenVMS V7.3 Alpha Programming Features I



Student Guide

Rev. 1.21

COMPAQ

OpenVMS V7.3 Alpha Programming Features I

Student Guide

Rev. 1.21

COMPAQ

OpenVMS V7.3 Alpha Programming Features I
Student Guide
May 2001

Contents

Overview

Course Content and Objectives 1
- Course Length 2
- Intended Audience 2
- Prerequisites 2
- Course Goals 2
- Course Nongoals 2
- Course Outline 3
  - General Programming 3
  - Event Synchronization 4
  - The Process And Scheduling Subsystem 4
  - Interprocess Synchronization And Communication 4
  - Intracluster Communication 5

Module 1 — Programming: Introduction to Alpha Architecture

Introduction 1
- Objectives 1
- Topics 1

CISC vs. RISC Architectures 2
- Computer Architectures 2
  - CISC Architectures 2
  - RISC Architectures 2

Alpha Risc Architecture Overview 4
- Alpha Data Types 4
- Alpha Registers 4
- Instruction Set Types 5
- Privileged Architecture Library 5
  - OpenVMS PALcode 6

OpenVMS Access Modes 7
- Kernel Mode 7
- Executive Mode 7
- Supervisor Mode 7
- User Mode 7

OpenVMS Processes 8
The Image 8
The Process 8
Viewing Your Process 9
Viewing All Processes 11
OpenVMS Alpha Memory Management 12
Memory Terms 12
Address Space 12
OpenVMS Alpha Memory Layout 13
OpenVMS Alpha Address Translation 14
OpenVMS Alpha Paging Concepts 14
OpenVMS Alpha Paging Concepts 15
Demand Paging 15
Page Fault 15
Free Page List 15
Working Set List 15
Modified Page List 15
Page File 15
OpenVMS Alpha Page Lists 16
Viewing Memory Usage And Limits 17
Alpha Calling Standard 18
Descriptors 19
Descriptors Example 20
More on Descriptors 22
C Implementation of Descriptors 24
Example of using Descriptor Structure and Macro 25
Example of Interchangeable Use of $DESCRIPTOR and struct dsc$descriptor 26
Example of Incomplete Descriptor Initialization 27
Inline Initialization of Descriptors 28
Sample of Warning Associated with Initialization of Pointer From Automatic Storage 29
Dynamic String Descriptors 30
Example of Returning Space to the Heap 32
Compiler Technology 34
Common Programming Languages For OpenVMS 35
AMACRO 35
VAX MACRO 35
C 35
BLISS 35

Module 2 — Compiling and Linking

Introduction 1
Objectives 1
Topics 1
OpenVMS Symbolic Naming Conventions 2
Locating Symbol Names For C 5
Locating Symbol Names for Bliss 9
Locating Symbol Names for Macro32 10
Producing Programs 11
Producing Programs Example 12
Program Sections 13
Program Section Example 14
Producing Programs: Compiling/Assembling 16
Data Alignment Issues 17
Non-Aligned Data Example 18
Non-Aligned and Compiler Aligned Data Sample Runs 20
Aligned Data Example 21
Using Traceback with Listings 23
Using Maps and Machine Code Listings 27
Debugging 38
Sample Debug Session Using Command Interface 39
Object Modules 46
Linking 47
$Analyze/Image Example 48
Linker Qualifiers 57
Linker Options 58
Alpha Calling Standard Mechanics 59
Procedure Descriptors and Values 60
Procedure descriptor formats: 60
Accessing Procedure Descriptor Example 61

Module 3 — Libraries and DCL

Introduction 1
Objectives 1
Topics 1
Libraries 2
Library Commands 2
Libraries and The Linker 3
Library Example 4
Linker Qualifiers and Libraries 12
Text Libraries and C Headers 12

The Run-Time Library 13
Run-Time Library Example 14
Utility Routines 16
System Services 17
Procedure Status 18
Condition Value Processing 19
Condition Value Processing Example 20
Condition Value Handling 21
LIB$SIGNAL Example 22
DCL Commands 25
The Command Definition Utility 26
CDU Routines 28
Using CDU 29
Simple CDU Example 30
CDU Example 33
HELP Libraries 45
HELP Library Example 46
Getting Help from a Program Example 59

Module 4 — Event Synchronization

Introduction 1
Objectives 1
Topics 1
Synchronization 2
Event Flags 3
Flow Using Event Flags for Synchronization 4
The "I/O" Status Block 5
SYS$SYNCH System Service 6
Time and Time-Related Services 7
Timer Example 8
Additional Time and Time-Related Services 9
Asynchronous System Traps 10
AST Flow 11
Asynchronous System Traps Example 12

Module 5 — Process and Scheduling Services

Introduction 1
Objectives 1
Topics 1
The Process 2
Conceptual View Of A Process With Threads 3
Software Context 4
Hardware and Virtual Address Space Context 7
The Create Process System Service 8
LIB$SPAWN Example 10
Interactive DCL Example 12
Getting Job/Process Information 14
Create Process Example 15
Wildcard GETJPI Example 19
Process SCAN Example 25
Setting Process Characteristics 30
Process Management/Control 31
Hiber/Wake Example 32
Terminating Images/Processes 37
Exit Handlers 38
Force Exit Example 43
Delete Process Example 45

Module 6 — Interprocess Synchronization and Communication

Introduction 1
Objectives 1
Topics 1
Locking Concepts 2
The OpenVMS Lock Manager 3
Enqueueing and Dequeueing Locks 4
Lock Features 6
Lock Example 7
Common Event Flags 16
Common Event Flag Example 17
Logical Names 22
Logical Name Translation Example 23
Clusterwide Logicals 27
Cluster-Wide Logicals Implementation 28
Cluster-Wide Logical Name API 29
Cluster-Wide Logical Name API Example 30

Mailboxes 36
Typical Mailbox Design 36
Mailbox Example 38
Termination Mailboxes 42

Module 7 — Intracluster Communication

Introduction 1
Objectives 1
Topics 1
Intra-Cluster Communications (ICC) 2
ICC Features 2
ICC Concepts 2
ICC Model 4
ICC Routines 5
ICC Programming Considerations 7
Very Simple ICC Example 10
Node ALLEN session 19
Node VEDDER session 19
More Complex ICC Example 21
ICC Security Considerations 43
ICC Security Example 43
ICC Security Example 44

Appendix A — Programming Galaxy CPUs

CPU Migration Using DCL 2
System Event Notification 4
System Event Notification Example 5
CPU Management Programming Interface 17
CPU Transitioning Example 18

Appendix B — DCL Parsing

DCL Parsing Interfaces 2
Sample DCL Parser 3

Appendix C — $SNDJBC Example

Sample Submit From a Program 2
$ 4

Appendix D — Performance Management API

The Performance Management API 2
SYS$GETRMI Example 3

Course Content and Objectives

This course is intended to quickly bring the System Programmer up to speed in an OpenVMS environment. The scope of material is fairly broad from basic compiling to advanced system services. Although some of the material will be provided as an exposure to capabilities available to the programmer, it is expected that the students will be able to program at an in depth level upon completion of this course. The first week of the course is dedicated to laying out the fundamentals of programming under OpenVMS, process management, and synchronization. The course is divided into categories that naturally align with the major subsystems of OpenVMS:

- General Programming
- Process and Scheduling Subsystem
- Synchronization

This course is designed to take a programmer with basic DCL understanding and provide the foundations to take advantage of all available features of the OpenVMS operating system. The course will be driven by examples of the use of major system services and run-time library procedures. Most examples will be provided in C. Only components provided with a standard distribution will be addressed in this course, i.e. other than C, no layered products will be incorporated into the course.

The course will include lab exercises designed to reinforce the skills taught in the course.

Although the material is broad and some of it will be absorbed only as an overview, it is expected when you complete the course that you will *minimally* be able to:

- Compile and link a program
- Read listing and map files
- Call procedures, specifically system services
- Create a process
- Communicate with a process
- Synchronize using event flags and locks

Course Length

4.5 days

Intended Audience

OpenVMS Application and System Programmers.

Prerequisites

Students should have a fundamental understanding of OpenVMS System concepts. Students should be familiar with the C programming language, including the use of structures and pointers.

Course Goals

Although the material is broad and some of it will be absorbed only as an overview, it is expected when the student leaves the course, she/he will *minimally* be able to:

- Compile and link a program
- Read listing and map files
- Call procedures, specifically system services
- Create a process
- Communicate with a process
- Synchronize using event flags and locks
- Describe Intracluster Communication programming techniques.

Course Nongoals

- Kernel mode programming will not be covered.

Course Outline

General Programming

- General Alpha Architecture
  - Registers and Data Types
  - Instruction Overview
  - PALcode
  - Virtual Address Space and Address Translation
  - OpenVMS Calling Standard
  - Status Values and System Messages
  - Procedure Descriptors
- Programming Languages
  - AMACRO
  - MACRO32
  - DEC C
  - BLISS
- Producing Programs
  - Symbolic Naming Conventions
  - Program Sections
  - Compiling and Object Modules
  - Linking and Executable Images
  - The Symbolic Debugger
  - Libraries
  - Text Libraries
  - C header "files"
  - Object Libraries
  - Help Libraries
  - DCLTABLES and Command Definition Utility Basics

Event Synchronization

- Asynchronous Events
- Local Event Flags
- Asynchronous System Traps
- Timers and Time
- Time Based Events

The Process And Scheduling Subsystem

- The Process
  - Software Context
  - Hardware Context
  - Memory Management Context
  - Methods for Process Creation/Management
  - The Create Process (SYS$CREPRC) System Service
  - Spawning (LIB$SPAWN)
  - Deleting Processes/Running Down Images
  - Exit Handlers
  - Suspending and Resuming Processes
- Hibernating and Waking Processes
- Obtaining Information about Processes

Interprocess Synchronization And Communication

- Lock Management
  - Locks and Resources
  - Lock Modes
  - Lock Conversion
  - Value Blocks
- Process Synchronization/Communication
  - Common Event Flags
  - Logical Names
  - Mailboxes (SYS$CREMBX/SYS$QIO)
  - Termination Mailboxes

Intracluster Communication

- ICC Concepts
- ICC Programming
- ICC Security

General Programming: Introduction to Alpha Architecture

Module 1

Introduction

This module presents the fundamental architectural concepts of OpenVMS under an Alpha platform. It will provide students with the foundations to understand and appreciate the fundamental Alpha instruction set and registers. Students will be able to describe OpenVMS virtual address spaces and fundamental memory management concepts. Students will be presented with a basic view of a process. Students will also be able to describe the procedure calling standard as implemented on OpenVMS Alpha.

Objectives

After completing this module, you should be able to:

- List differences between Complex Instruction Set (CISC) and Reduced Instruction Set (RISC) computers.
- Identify the 4 OpenVMS access modes and describe activities that happen in each access mode.
- Identify the three contexts that define a process.
- Describe the lists associated with memory management and their function.
- Identify the three general methods of passing arguments to a function.
- Describe the purpose and use of descriptors.

Topics

- CISC vs RISC Architectures
- Alpha RISC Architecture Overview
- OpenVMS Access Modes
- Alpha OpenVMS Memory Management
- Alpha Calling Standard
- Compiler Technology

CISC vs. RISC Architectures

Computer Architectures

Computer architectures describe aspects of a computer system which provide consistency across families of computer products. The fundamental aspect of a computer system is its machine instruction set definition and characteristics. Other aspects of a computer system also define its architecture, such as the way it manages memory and the implementation of its I/O subsystem. The VAX used an architecture referred to as *a Complex Instruction Set Computer* (CISC). Newer architectures, such as Alpha, support a *Reduced Instruction Set Computer* (RISC).

CISC Architectures

- CISC Architecture Goals
  - Ease of compiler development
  - Large number of instructions for compiler writers
  - Smaller Programs
- CISC Characteristics
  - Many processor cycles per instruction, i.e. slower instructions due to complexity
  - Some non-interruptible instructions
  - Pipelining instructions is more difficult

RISC Architectures

- RISC Goals
  - Performance through pipelining, parallelism, shorter cycle times
  - Flexibility and platform independence
  - Faster processor development time
- RISC Characteristics
  - Speed through pipelining, caching, and parallelism
  - Large data paths
  - Alignment issues
  - Lack of microcode
  - Typically fewer instructions
  - Dependence on good compilers
  - Large Number of registers

CISC vs. RISC type instructions load, operate, store sequence:

VAX	Alpha

INCL X	LDQ R28, 40(R27)
	LDL R25, (R28)
	ADDL R25, 1, R25
	STL R25, (R28)

Alpha Risc Architecture Overview

Alpha Data Types

Integral Data types

- Byte (8-bit little-endian, support exists for big-endian)
- Word (16-bit)
- Longword (32-bit)
- Quadword (64-bit)

Floating-Point Data types

- F_floating
- G_floating (D_floating conversions)
- S_floating (IEEE single precision)
- T_floating (IEEE double precision)

Memory Accesses

- The VAX supports odd byte aligned memory accesses.
- Alpha supports quadword and longword aligned memory accesses. Byte and word operations are supported through extract, insert, and mask operations. EV6+ processors support odd-byte/word accesses.

Alpha Registers

Special Registers

- The Program Counter (PC) and Processor Status (PS).
- Note: the PC is implied on branch and subroutine jump instructions. The PC is not accessible as an integer register. PS does not contain condition codes.

Integer Registers

- 32 integer registers R0 through R31, 64-bits wide. R31 returns 0 as a source operand and is discarded as a destination.

Floating-Point Registers

- Floating-point registers F0 through F31, 64-bits wide. F31 returns a true zero value when used as a source register.

Lock Registers

- 2 per-processor registers used with the Load Memory into Integer Register Locked (**LD*x*_L**) and Store Integer Register into Memory Conditional (**ST*x*_C**) instructions for guaranteed read/write ordering.

Alpha instructions are fixed in size (32-bit).

Instruction Set Types

- Memory Integer Load/Store
- Control Instructions
  - Conditional Branch, Unconditional Branch, and Jumps
- Integer Operate Instructions
  - Integer Arithmetic
  - Add, Multiply, Subtract, and Compare
  - Logical and Shift
  - Byte Manipulation
- Memory Format Floating-Point Instructions
- Floating-Point Operate Instructions
  - Add, Multiply, Subtract, and Compare
- Miscellaneous
  - Call PAL, Prefetch Data, Memory Barrier, Trap
    - VAX Compatibility (NOT compatibility mode)

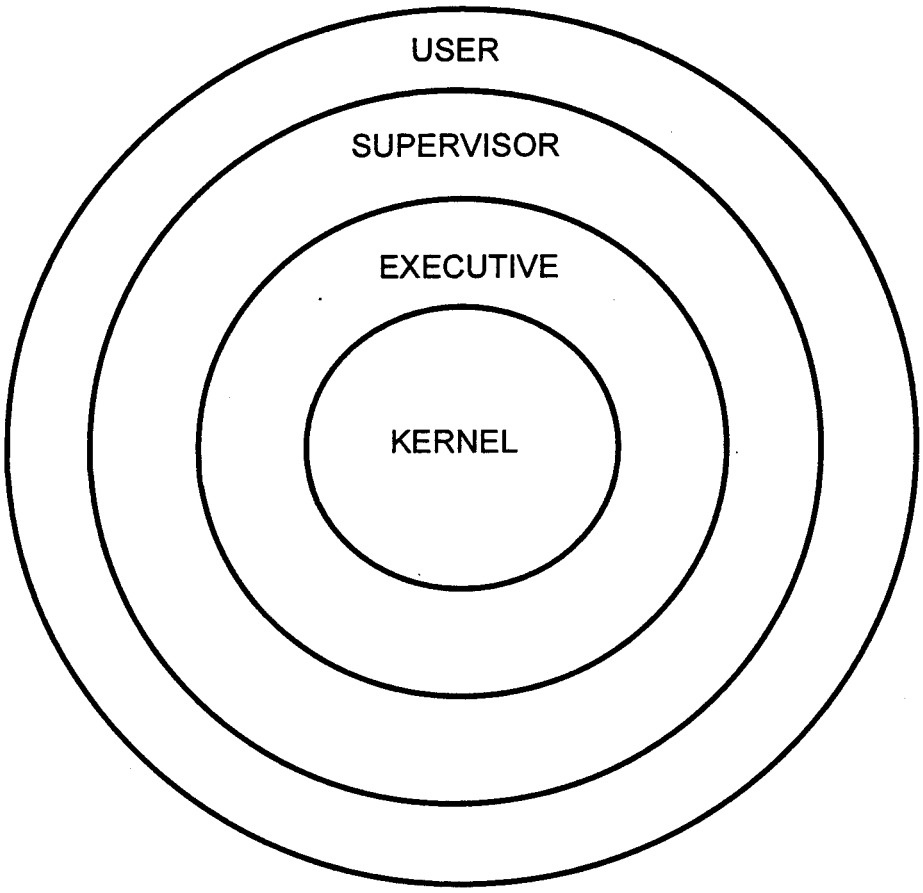
Privileged Architecture Library

Operating systems require support for primitive type operations such as context switching, I/O, interrupts, exceptions, and memory management. This support is not built directly into the Alpha instruction set, but is instead supported through *Privileged Architecture Library* routines (PALcode). PALcode acts as a set of subroutines activated by the hardware or software CALL_PAL instructions. PALcode is, typically, written by hardware designers, in machine code, to support operating system features not addressed by the instruction set. PALcode will vary from one Alpha platform (such as OpenVMS, OSF/1, or Windows NT) to another.

OpenVMS PALcode

- Registers
  - PS (Processor Status), SP/R30 (Stack Pointer), IPRs (Internal Processor Registers).
- Unprivileged OpenVMS PALcode Instructions
  - Directly mapped to VMS instructions: BPT, BUGCHK, CHME, CHMK, CHMS, CHMU, INSQxxx, PROBEx, REI, REMQxxx, RD_PS (*MOVPSL*).
  - Miscellaneous: AMOVRM, AMOVRR (Atomic moves), GENTRAP, IMB, RSCC, SWASTEN, WRITE_UNQ, WR_PS_SW.
- Privileged OpenVMS PALcode Instructions
  - Directly mapped to VMS instructions: HALT, MFPR, MTPR.
  - Miscellaneous: CFLUSH, DRAINA, LDQP, STQP, SWPCTX.
- Activated through CALL_PAL pal_instruction.
- Support for absolute, self-relative, longword, and quadword queues.
- Memory management and address translation are supported through PALcode.

OpenVMS Access Modes



Kernel Mode

System Services, Pager, Scheduler, Device Drivers, etc.

Executive Mode

Record Management Services (RMS)

Supervisor Mode

Command Language Interpreter (DCL)

User Mode

User programs, Utilities, Commands (Not handled internally by DCL)

OpenVMS Processes

The Image

- An executable, normally a program, may be a set of library routines.

The Process

- Supports execution of images.
- Provides a scheduling context (actually scheduled as kernel threads).
- Created when you login, spawn, **$ RUN** with most qualifiers, initiate batch execution, and issue the **SYS$CREPRC** system service.
- Has rights and restrictions.
  - User Identification code (UIC) (Group/Member).
  - Identifiers.
  - Privileges.
- Has identification:
  - Process Id
  - Process Name (Unique to group)
- Limits system resource usage through quotas and limits.
- May or may not have a command language interpreter (DCL).
- Has private address space.

Space	Region	Life	Size Limit
P0	program	image	1 gigabyte
P1	control	process	1 gigabyte
P2	64-bit process (may have user-defined regions)	image	terabytes

Viewing Your Process

```
$ sh process/all
16-JUL-1996 15:42:29.33   User: ALLEN          Process ID:   00000056
                          Node: ALLEN          Process name: "_RTA1:"

Terminal:            RTA1:  (ALLEN::ALLEN)
User Identifier:     [ALLEN]
Base priority:       4
Default file spec:   _DKA300:[ALLEN]
Number of Kthreads: 1

Devices allocated:   ALLEN$RTA1:

Process Quotas:
 Account name:
 CPU limit:                        Infinite  Direct I/O limit:        150
 Buffered I/O byte count quota:       99808  Buffered I/O limit:      150
 Timer queue entry quota:                10  Open file quota:         100
 Paging file quota:                   47872  Subprocess quota:         10
 Default page fault cluster:             64  AST quota:               248
 Enqueue quota:                        1999  Shared file limit:         0
 Max detached processes:                 0  Max active jobs:           0

Accounting information:
 Buffered I/O count:          46  Peak working set size:       1968
 Direct I/O count:            14  Peak virtual size:         166464
 Page faults:                144  Mounted volumes:                0
 Images activated:             1
 Elapsed CPU time:             0 00:00:00.25
 Connect time:                 0 00:00:17.90
```

Viewing Your Process (continued)

```
Authorized privileges:
 NETMBX    SETPRV    TMPMBX

Process privileges:
 NETMBX                may create network device
 SETPRV                may set any privilege bit
 TMPMBX                may create temporary mailbox

Process rights:
 INTERACTIVE
 REMOTE

System rights:
 SYS$NODE_ALLEN

Auto-unshelve: on

Image Dump: off
Process Dynamic Memory Area
    Current Size (bytes)          57344    Current Size (pagelets)        112
    Free Space (bytes)            47756    Space in Use (bytes)          9588
    Size of Largest Block         47548    Size of Smallest Block          24
    Number of Free Blocks             5    Free Blocks LEQU 64 Bytes        3

There is 1 process in this job:

  _RTA1: (*)
$
```

Viewing All Processes

```
$
$ show system
OpenVMS V7.0  on node ALLEN  16-JUL-1996 15:42:36.90  Uptime  0 05:07:14
  Pid     Process Name    State  Pri      I/O       CPU         Page flts  Pages
00000041 SWAPPER         HIB     16        0   0 00:00:00.36         0      0
00000045 IPCACP          HIB     10       11   0 00:00:00.04      1716     18
00000046 ERRFMT          HIBO     8       --  swapped  out  --             26
00000047 OPCOM           HIBO     7       --  swapped  out  --             32
00000048 AUDIT_SERVER    HIB     10       59   0 00:00:00.25       522     18
00000049 JOB_CONTROL     HIB      8       30   0 00:00:00.12      1280     27
0000004A SECURITY_SERVER HIB     10       44   0 00:00:36.51     15224    169
0000004C DECW$SERVER_0   HIB      6      576   0 00:00:33.77     13494    190
0000004D DECW$SESSION    LEF      6      360   0 00:00:02.46      5631     26
0000004F DECW$MWM        LEFO     4       --  swapped  out  --             27
00000050 ALLEN           LEFO     4       --  swapped  out  --             32
00000051 DECW$TE_0051    COM      4      181   0 00:00:12.87     11432    169
00000052 _FTA3:          HIB      7     1424   0 00:00:04.55      1798    114
00000053 NETACP          HIB     10       42   0 00:00:00.18        61     72
00000054 EVL             HIB      5       50   0 00:00:00.21       103     24  N
00000055 REMACP          HIB      9       15   0 00:00:00.04        38     20
00000056 _RTA1:          CUR      4      149   0 00:00:00.36       196     94
$
```

OpenVMS Alpha Memory Management

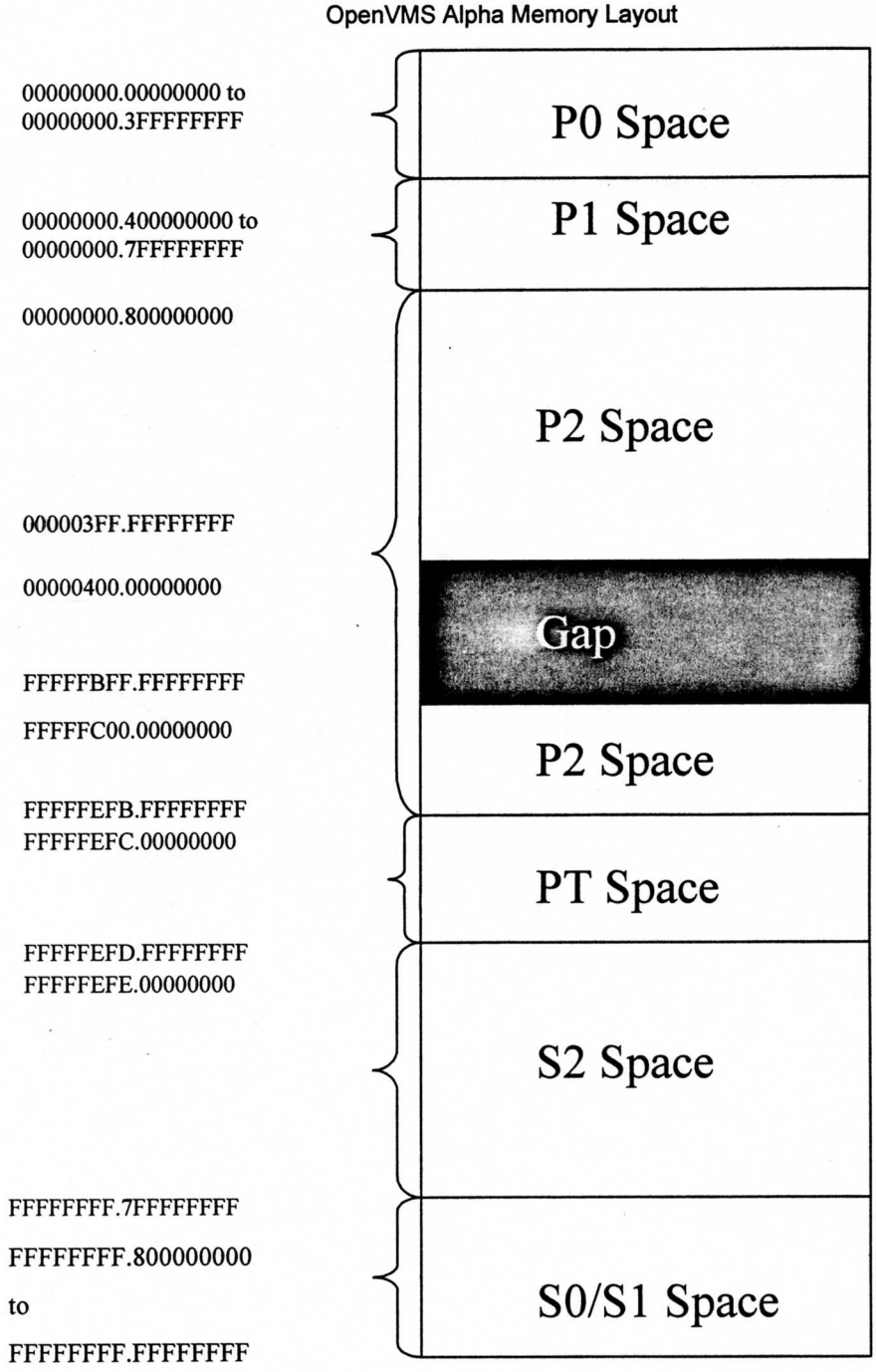
Memory Terms

- Terabyte (TB) 2^{40} (~1,000,000,000,000) bytes.
- Petabyte (TB) 2^{50} (~1,000,000,000,000,000) bytes.
- Perspective
  - Gigabyte 2^{30} bytes (~1,000,000,000)
  - Megabyte 2^{20} bytes (~1,000,000),
  - Kilobyte 2^{10} (~1,000) bytes.

Address Space

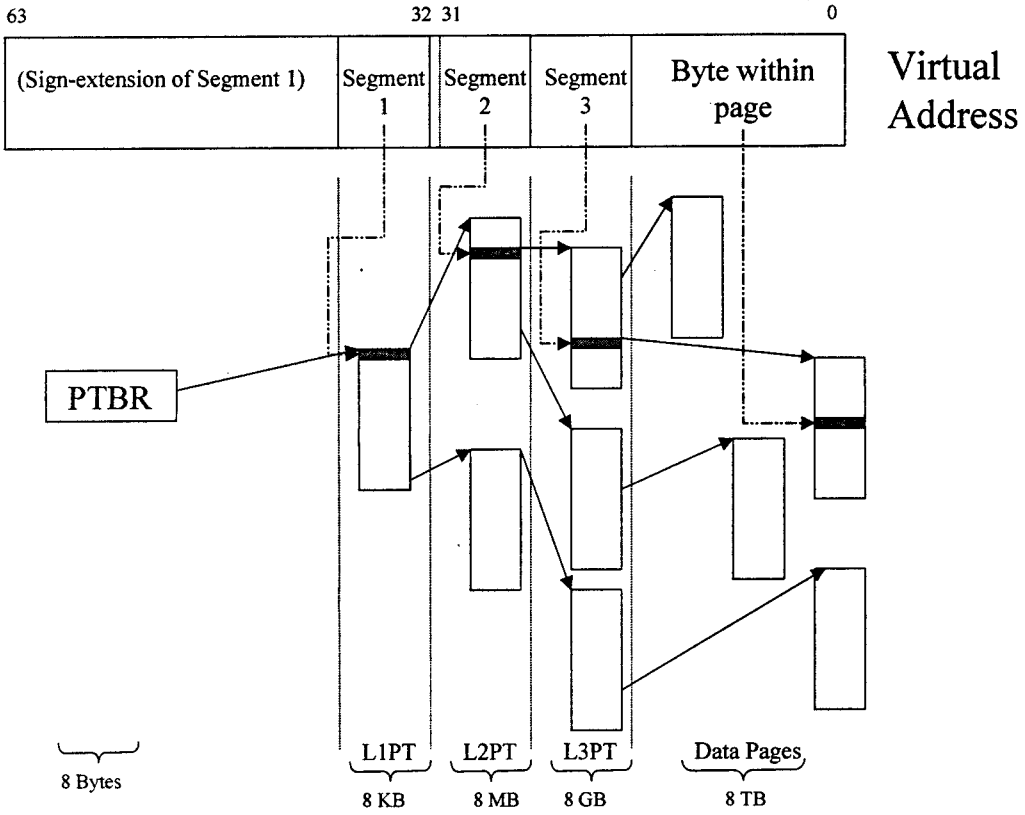
- Variable page size: 8, 16, 32, or 64 KB vs fixed 512 byte on VAX.
- 512 byte Pagelets provide a translation mechanism to disk blocks and back to VAX Architectures.
- Architectural limitation of 64-bit virtual and physical addresses.
- Alpha does not distinguish P0, P1, P2, S0, S1 and S2 space; OpenVMS does.

OpenVMS Alpha Memory Layout



Not to scale

OpenVMS Alpha Address Translation



OpenVMS Alpha Paging Concepts

Demand Paging

- Pager acts as program loader.

Page Fault

- Address Translation Not Valid Exception.
- Low bit clear in page table entry.
- Types of Initial Faults:
  - Image Section Faults
  - Demand Zero
  - Global Valid

Free Page List

- Provides a list of available pages.
- Tracks pages with valid information, yet not in working set list.

Working Set List

- Tracks pages owned by a process.
- Used for page replacement.

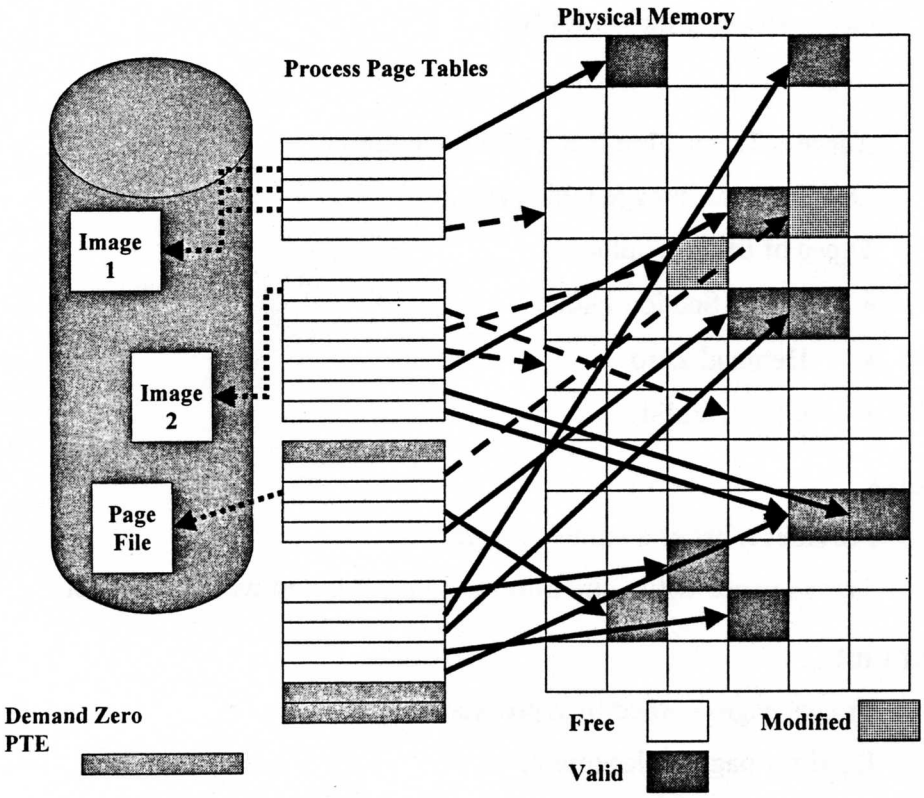
Modified Page List

- Tracks *dirty* pages still of interest to a process, but not yet written to disk.

Page File

- Holding *tank* for *dirty* pages written to disk .

OpenVMS Alpha Page Lists



Viewing Memory Usage And Limits

```
$
$
$ show memory/physical
                System Memory Resources on 16-JUL-1996 15:42:49.86
Physical Memory Usage (pages):      Total        Free      In Use    Modified
  Main Memory (32.00Mb)              4096         154        3257         685
Of the physical pages in use, 2359 pages are permanently allocated to OpenVMS.
$
$ show work
  Working Set (pagelets)  /Limit=2000  /Quota=4000  /Extent=16384
  Adjustment enabled      Authorized Quota=4000  Authorized Extent=16384

  Working Set (8Kb pages) /Limit=125  /Quota=250  /Extent=1024
                          Authorized Quota=250  Authorized Extent=1024
$
$
$ show memory/file
                System Memory Resources on 16-JUL-1996 15:43:10.61
Paging File Usage (blocks):                      Free  Reservable       Total
  DISK$AXPVMSSYS:[SYS0.SYSEXE]SWAPFILE.SYS
                                                  512         512        6656
  DISK$AXPVMSSYS:[SYS0.SYSEXE]PAGEFILE.SYS
                                                56688       13504       73600
$
```

Alpha Calling Standard

The OpenVMS operating system provides a standard that defines how procedure (i.e. subroutines and functions) calls will be implemented internally, how arguments are passed and received, and how values will be returned. The intent of this standard is to provide a uniform interface allowing for easy inter-language calls. This standard is followed on run-time library and system service calls.

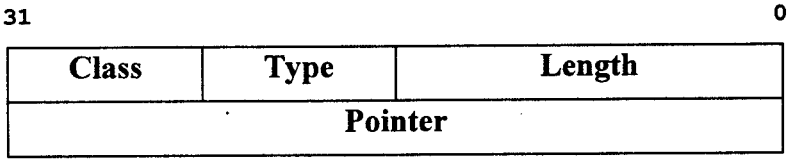
- Arguments are described under the run-time library and system service calls by:
  - type (byte, word (2 bytes), longword (4 bytes), quadword (8 bytes), etc.)
  - passing mechanism.
- General Argument passing mechanisms:
  - By value
  - By reference
  - By descriptor (descriptor is passed by reference)
- Descriptors are used to describe strings, arrays, and other atomic data types. Descriptors are either quadword (8 byte) entities or may be 24 bytes for 64-bit descriptors.

Descriptors

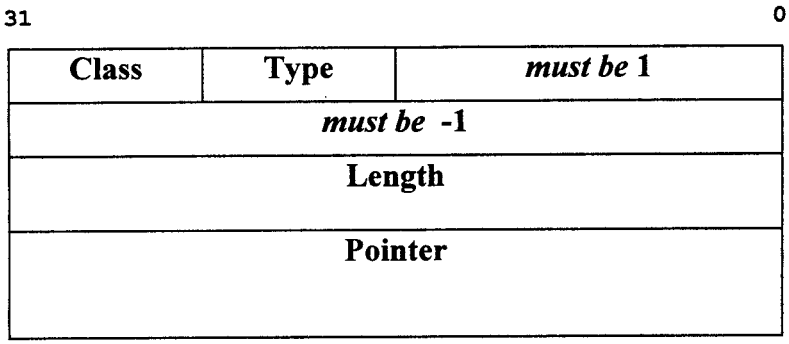
Descriptor layouts are defined by the macro **$DSCDEF** and the C header **dscdef.h**.

- Structure layouts for descriptors are provided by the **descrip.h** C header.

32-bit Descriptor format



64-bit Descriptor format



- The ***type*** field describes the atomic data type. Types are defined in the header **descrip.h** and are of the form **DSC$K_DTYPE_*type***. For instance, **DSC$K_DTYPE_T** describes text.
- The ***class*** field describes special aspects of the data type, such as with string descriptors, the string may be returned into a fixed area (scalar) or may be allocated dynamically. Classes are defined in the header **descrip.h** and are of the form **DSC$K_CLASS_*type***. For instance, **DSC$K_CLASS_S** describes a scalar array.
- ***Length*** is the size of the data, normally in bytes.
- Pointer is the address of the data.

Descriptors Example

```
$
$ type dsc_ex.c
/* Example of using descriptors in C. */
#include <stdio.h>
#include <stdlib.h>
#include <ssdef.h>
#include <lib$routines.h>
#include <starlet.h>
#include <descrip.h>
#define  check_call(status)      if(!(status&1)) sys$exit(status)
#define TIME_SIZE 23

int  main(void)
{
     char     system_time[TIME_SIZE+1];
     const    float  five_secs = 5.0;

/* Declare a descriptor which will be initialized in the program. */
     struct   dsc$descriptor_s systim_dsc = {0,DSC$K_DTYPE_T,DSC$K_CLASS_S};
/* Declare a statically defined descriptor. */
     $DESCRIPTOR(str_dsc,"See you in five seconds!");
     int      status;

/* Send a message using a descriptor. */
     lib$put_output(&str_dsc);
/* Initialize the size and location of the descriptor. */
     systim_dsc.dsc$w_length = TIME_SIZE;
     systim_dsc.dsc$a_pointer = system_time;
```

Descriptors Example (continued)

```
/* Get the system time. */
    status = lib$date_time(&systim_dsc);
    check_call(status);
/* Display the system time. */
    lib$put_output(&systim_dsc);
/* Stall for five seconds. */
    status = lib$wait(&five_secs);
    check_call(status);
/* Reset the length. */
    systim_dsc.dsc$w_length = TIME_SIZE;
    status = lib$date_time(&systim_dsc);
    check_call(status);
/* Display the time. */
    lib$put_output(&systim_dsc);
/* Redisplay the time using a C type string. */
    system_time[TIME_SIZE] = '\0';
    printf("Time from printf: %s\n",system_time);
    return(SS$_NORMAL);
}

$ cc dsc_ex
$ link dsc_ex
$ r dsc_ex
See you in five seconds!
16-JUL-1996 18:03:01.95
16-JUL-1996 18:03:06.95
Time from printf: 16-JUL-1996 18:03:06.95
$
```

More on Descriptors

Why does OpenVMS use descriptors?

- Descriptors provide a well defined mechanism for passing arrays to subroutines.
- Consider the C style mechanism for passing strings.
  - The string array is passed as the address of an array of characters that is null ('\0') terminated.
  - This mechanism works fine for outputting a string, but on input the called function does not know how much space has been allocated to hold characters for the string.

Note

Note: the problem with the following example:

```
$ type trash.c
#include <stdio.h>
#include <stdlib.h>
int     main(void)
{
        int     age;
        char    name[8];
        printf("Enter age: ");
        scanf("%d",&age);
        while(getchar() != '\n'){;}
        printf("Enter name: ");
        gets(name);
        printf("%s is %d years old\n",name,age);
        return(EXIT_SUCCESS);
}
$ cc trash
$ link trash
$ r trash
Enter age: 29
Enter name: Billy Bitsenbites
Billy Bitsenbites is 7563628 years old
$
```

More on Descriptors (continued)

In the example, the integer "age" is allocated from the stack followed by the array name, so when the information is read any overflow from name over-writes the age.

Illustration:

State before any reads

Name:				
Age:				

State after reading age

Name:				
Age:			29	

State after reading name

Name:	l	l	i	B
	i	B	‘ ‘	y
Age:	n	e	s	t

C Implementation of Descriptors

C provides two general mechanisms for defining descriptors:

- struct dsc$descriptor_s *name*;
  - This mechanism is normally used to define descriptors that are non-constant.
- $DESCRIPTOR(*name*,*string*);
  - This mechanism is normally used to define descriptors that correspond to fixed literal strings.
- The methods may be used interchangeably.
- The "struct dsc$descriptor_s" structure definition:

```
$ set def             sys$common:[decc$lib.reference.decc$rtldef]
$ search/win=(0,10) descrip.h "struct   dsc$descriptor_s"
struct  dsc$descriptor_s
{
       unsigned short  dsc$w_length;   /* length of data item in bytes,
                          or if dsc$b_dtype is DSC$K_DTYPE_V, bits,
                          or if dsc$b_dtype is DSC$K_DTYPE_P, digits (4 bits each) */
       unsigned char   dsc$b_dtype;    /* data type code */
       unsigned char   dsc$b_class;    /* descriptor class code = DSC$K_CLASS_S */
       char            *dsc$a_pointer; /* address of first byte of data storage*/
};
...
$
```

- The $DESCRIPTOR macro definition:

```
$ search/win=5 descrip.h "$DESCRIPTOR("
 *      A simple macro to construct a 32-bit string descriptor:
 */
#define $DESCRIPTOR(name,string)        struct dsc$descriptor_s name = \
{ sizeof(string)-1, DSC$K_DTYPE_T, DSC$K_CLASS_S, string }


/*
$
```

Example of using Descriptor Structure and Macro

```
$ type stat_desc.c
#include <stdio.h>
#include <lib$routines.h>
#include <descrip.h>
#include <stdlib.h>
#define MAX_LINE 132

int     main(void)
{
```

Static storage is used to prevent compiler warnings because automatic storage is allocated from the stack, and its address is not known until run-time.

```
        static char     name_str[MAX_LINE+sizeof('\0')];
```

Structure is initialized to define the maximum size of the input string, type of text, class of scalar/statically declared, and the address of the string.

```
        struct dsc$descriptor_s  name = {MAX_LINE,DSC$K_DTYPE_T,
                        DSC$K_CLASS_S,name_str};
```

The descriptor "name" will descripe the string "Name ".

```
        $DESCRIPTOR(name_pmt,"Name: ");
```

Descriptors are passed by address.

```
        lib$get_input(&name,&name_pmt,&name.dsc$w_length);
        lib$put_output(&name);
        return(EXIT_SUCCESS);
}
$ cc stat_desc
$ link stat_desc
$ r stat_desc
Name: Billy Bitsenbites
Billy Bitsenbites
$
```

Example of Interchangeable Use of $DESCRIPTOR and struct dsc$descriptor

```
$ type stat_desc2.c
#include <lib$routines.h>
#include <descrip.h>
#include <stdlib.h>
#define MAX_LINE 132

int     main(void)
{
        static char     name_str[MAX_LINE+sizeof('\0')];

                        Replace the "struct dsc$descriptor_s" with $DESCRIPTOR macro.
        $DESCRIPTOR(name,name_str);
        $DESCRIPTOR(name_pmt,"Name: ");

        lib$get_input(&name,&name_pmt,&name.dsc$w_length);
        lib$put_output(&name);

        return(EXIT_SUCCESS);
}
$ cc stat_desc2
$ link stat_desc2
$ r stat_desc2
Name: Billy Bitsenbites
Billy Bitsenbites
$
```

Example of Incomplete Descriptor Initialization

```
$ typ bad_desc.c
#include <lib$routines.h>
#include <descrip.h>
#include <stdlib.h>
#define MAX_LINE 132

int     main(void)
{
                                The name.dsc$a_pointer field was not initialized.
        struct dsc$descriptor_s  name = {MAX_LINE,DSC$K_DTYPE_T,DSC$K_CLASS_S};
        $DESCRIPTOR(name_pmt,"Name: ");

        lib$get_input(&name,&name_pmt,&name.dsc$w_length);
        lib$put_output(&name);

        return(EXIT_SUCCESS);
}
$ cc bad_desc
$ link bad_desc
$ r bad_desc
Name: Seg Fault
%SYSTEM-F-ACCVIO, access violation, reason mask=00, virtual address=000000000000
0008, PC=FFFFFFFF8045AD94, PS=0000001B
%TRACE-F-TRACEBACK, symbolic stack dump follows
  image    module    routine             line      rel PC           abs PC
                                            0 FFFFFFFF8045AD94 FFFFFFFF8045AD94
                                            0 FFFFFFFF8047784C FFFFFFFF8047784C
 BAD_DESC  BAD_DESC  main                6655 00000000000000F0 00000000000200F0
 BAD_DESC  BAD_DESC  __main                 0 0000000000000064 0000000000020064
                                            0 FFFFFFFF8777D3D4 FFFFFFFF8777D3D4
$
```

Inline Initialization of Descriptors

```
$ type inline_desc.c
#include <lib$routines.h>
#include <descrip.h>
#include <stdlib.h>
#define MAX_LINE 132

int     main(void)
{
                                    Storage for name_str is automatic.
        char    name_str[MAX_LINE+sizeof('\0')];
        struct dsc$descriptor_s  name = {MAX_LINE,DSC$K_DTYPE_T,
                        DSC$K_CLASS_S};
        $DESCRIPTOR(name_pmt,"Name: ");

/* Pointer is initialized in the mainline code, because the string is in
   automatic storage.
*/
                        The pointer is initialized inline with no complaints from the compiler.
        name.dsc$a_pointer=name_str;
        lib$get_input(&name,&name_pmt,&name.dsc$w_length);
        lib$put_output(&name);

        return(EXIT_SUCCESS);
}
$ cc inline_desc
$ link inline_desc
$ r inline_desc
Name: Fred Flintstone
Fred Flintstone
$
```

Sample of Warning Associated with Initialization of Pointer From Automatic Storage

```
$ type bad_stat.c
#include <stdio.h>
#include <lib$routines.h>
#include <descrip.h>
#include <stdlib.h>
#define MAX_LINE 132

int     main(void)
{
        char    name_str[MAX_LINE+sizeof('\0')];
        struct dsc$descriptor_s  name = {MAX_LINE,DSC$K_DTYPE_T,
                        DSC$K_CLASS_S,name_str};
        $DESCRIPTOR(name_pmt,"Name: ");

        lib$get_input(&name,&name_pmt,&name.dsc$w_length);
        lib$put_output(&name);
        return(EXIT_SUCCESS);
}
$ cc bad_stat

                        DSC$K_CLASS_S,name_str};
.....................................^
%CC-W-ADDRCONSTEXT, In the initializer for name.dsc$a_pointer, "name_str" does n
ot have a constant address, but occurs in a context that requires an address con
stant.  This is an extension of the language.
at line number 11 in file SYS$SYSDEVICE:[TEACHER.SOL]BAD_STAT.C;2
$
```

Dynamic String Descriptors

All of the examples up to this point have illustrated the use of statically allocated (scalar) string descriptors. One of the capabilities of the Run-Time Library routines is that they will dynamically allocate space for a string, on input, using the class DSC$K_CLASS_D. Note: this feature is not available using system services.

Dynamic string allocation Example

```
$ type dyn_desc.c
#include <stdio.h>
#include <lib$routines.h>
#include <descrip.h>
#include <stdlib.h>
#define MAX_LINE 132

int     main(void)
{
```

Note: the pointer field is not initialized, but the class of DSC$K_CLASS_D is used, designating the request that RTL procedure allocate space for the string.

```
    struct dsc$descriptor_d  name = {MAX_LINE,DSC$K_DTYPE_T,DSC$K_CLASS_D};
    $DESCRIPTOR(name_pmt,"Name: ");

    printf("%p\n",name.dsc$a_pointer);
    lib$get_input(&name,&name_pmt,&name.dsc$w_length);
    lib$put_output(&name);
    printf("%p\n",name.dsc$a_pointer);

    return(EXIT_SUCCESS);
}
$ cc dyn_desc
```

Dynamic String Descriptors (continued)

```
$ link dyn_desc
$ r dyn_desc
```

The uninitialized pointer value.

```
0
Name: Dino
Dino
```

The pointer value after allocation of the string by the RTL routine.

```
34804
$
```

One potential danger of using dynamically allocated strings is that the space should eventually be returned to the heap, or you may face heap exhaustion.

- The routine `lib$sfree1_dd` can be used to return dynamically allocated strings to the heap.

Example of Returning Space to the Heap

```
$ type dyn_desc2.c
#include <lib$routines.h>
#include <descrip.h>
#include <stdlib.h>
#include <stdio.h>
#define MAX_LINE 132
#define INUSE 3

int     main(void)
{
        struct dsc$descriptor_d  name = {MAX_LINE,DSC$K_DTYPE_T,DSC$K_CLASS_D};
        $DESCRIPTOR(name_pmt,"Name: ");
        int     vm_inuse;
        int     vm_ret_size;
        int     item = INUSE;

                                Determine the amount of dynamic memory in use.
        lib$stat_vm(&item,&vm_inuse);
        printf("%d bytes of dynamic vm in use.\n",vm_inuse);


                                        Grab a dynamic string.
        lib$get_input(&name,&name_pmt,&name.dsc$w_length);
        lib$put_output(&name);


                                Track the space utilization after grabbing dynamic memory.
        lib$stat_vm(&item,&vm_inuse);
        printf("%d bytes of dynamic vm in use.\n",vm_inuse);
```

Example of Returning Space to the Heap (continued)

```
        vm_ret_size = name.dsc$w_length;

                        Free the string and determine the amount of dynamic memory in use.
        lib$sfree1_dd(&name);
        lib$stat_vm(&item,&vm_inuse);
        printf("%d bytes of dynamic vm in use.\n",vm_inuse);
        return(EXIT_SUCCESS);
}
$
$ cc dyn_desc2
$ link dyn_desc2
$
$ run dyn_desc2
0 bytes of dynamic vm in use.
Name: Dino Mite
Dino Mite
24 bytes of dynamic vm in use.
0 bytes of dynamic vm in use.
$
```

Compiler Technology

A compiler translates source code written in a high-level language and translates it to machine language.

- Compilers are, typically, built with at least two phases:
  - Front end Source code processor
  - Back end Machine code generator (optimizer)
- Digital provides a common back end called GEM to handle architectural issues involved in code generation and optimization.
- GEM optimizations include:
  - Register allocation
  - Inline expansion of function calls
  - Redundancy elimination
  - Instruction scheduling to keep the pipeline full
  - Branch Prediction
  - Loop optimization and replication
- OpenVMS Alpha does not ship with an assembler. Note: due to optimizations provided by compilers it is usually better to write in a high-level language (Drivers may be written in C).
- MACRO32 is provided as a compiler under OpenVMS Alpha.

Common Programming Languages For OpenVMS

AMACRO

- Everything gets translated into this.
- AMACRO is a separately licensed product and may be used to assemble Alpha macro code.
- You will learn some AMACRO by osmosis looking through dumps and listings.

VAX MACRO

- This ships as the lone programming language on a base OpenVMS kit.
- Elegant CISC assembler.
- Much of the core kernel code is still written in VAX Macro.

C

- Movement toward C.
- Version 7.0 hooks allow for kernel level system programming in C.

BLISS

- A Digital cultivated system programming language from CMU.
- Highly macro driven.
- Looks a lot like PASCAL.
- Many utilities were (and still are) being written in BLISS.