COURSE OVERVIEW

- **Day 2 - Subsystems; System Management**

  - Subsystems

    * Overview
    * Definitions
    * Structure
    * View operations
    * Code views
    * Compatibility
    * Multi-machine management
    * Target builder
    * Reservations and check-in/out
    * Work orders

  - Users, Groups, Access Control, and Accounting

    * Users and groups
    * Access control rules
    * Job identity
    * Privileged mode
    * Universe Acls
    * Problems and diagnosis

COURSE OVERVIEW

- **Day 3 - Networking; Disks and Deamons**
  - Networking
  - Disk daemon management
  - Garbage problems
  - Daemons
  - Error log
  - System availability

COURSE OVERVIEW

- Day 4 - System Diagnosis and Recovery
  - System structure
  - DFS
  - Crash dumps
  - Kernel and EEDB
  - Diagnostic Tools
  - Disk errors
  - Questions and answers

OBJECTS - DEFINITION

- Defined by R1000 architecture
- Low-level, execution-oriented
  - The term "object" is also used to refer to environment objects such as files and Ada units
- Consist of a "type part" and a "value part"
- "Types" are special kinds of objects with a null value part
- The "value part" is some number of bits of data
- The "type part" indicates the type and structure of the value. Part of the type part of an object is a type descriptor. The format of the type descriptor is defined by the R1000 architecture.

R1000 GURU COURSE

- **Objectives**

  - Understand the R1000 system implementation

  - Be able to diagnose system problems
    * find solutions
    * know what information to send
    * know when to get help

  - Be more effective at giving customers advice and training their system experts

  - Be a more worldly person

COURSE OVERVIEW

- Day 1 - Basic Structures and Ada

    - Objects
    - R1000 storage organization
    - Instruction set
    - Storage hierarchy
    - Execution environment
    - Exceptions, unchecked deallocation, unchecked conversion
    - Program structure
    - Tasking and task operation
    - Virtual processors and jobs
    - Object management system

OBJECTS - DEFINITION (CONT'D)

- **Part of the type information is the object kind** 7 bits 128 kinds ~50 used

- **The kind divides objects into classes that the R1000 knows about. For example:**

  - Discrete (includes integer, fixed, enumerations, etc.)
  - Float
  - Access
  - Entry
  - Package/Task
  - Array
  - Record
  - Etc.

- **The type identifies the specific Ada type and includes information about the data format**

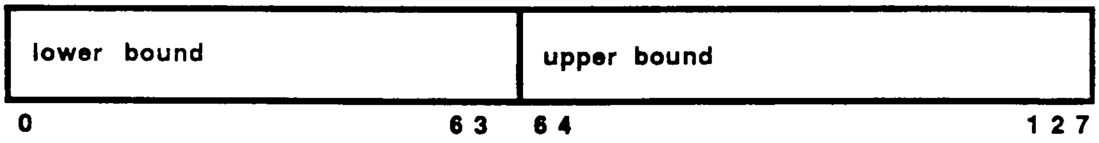
- **Types follow Ada semantics with respect to type compatibility**

- **Type descriptors describe a type by storing all information relevant to interpreting the bits that represent the value**

OBJECT - TYPE DESCRIPTORS

- Examples of type descriptors

For discrete types:



lower bound

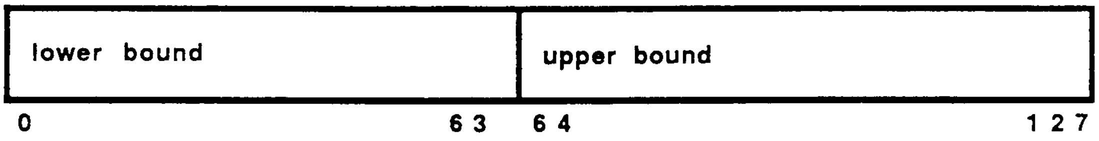
upper bound

0

63 64

127

For floating types:



lower bound

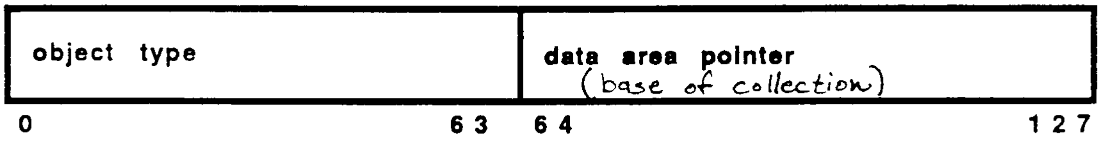
upper bound

0

63 64

127

For access types:



object type

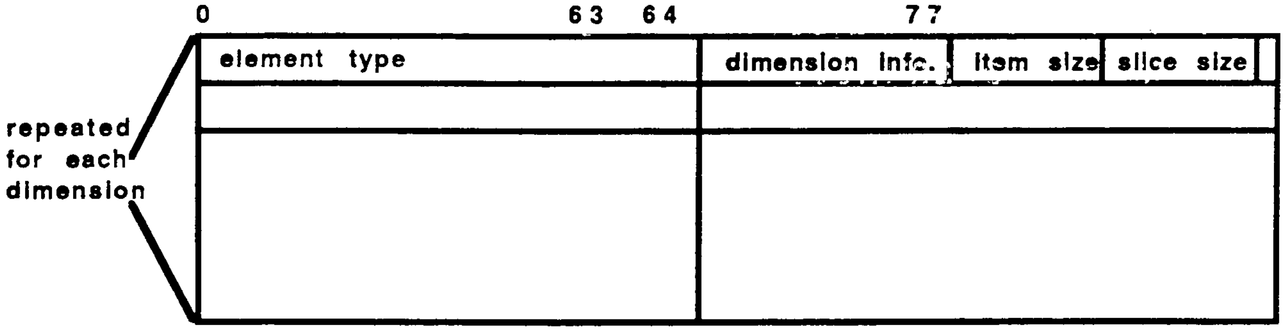
data area pointer
(base of collection)

0

63 64

127

For array types:



0

63 64

77

element type

dimension info.

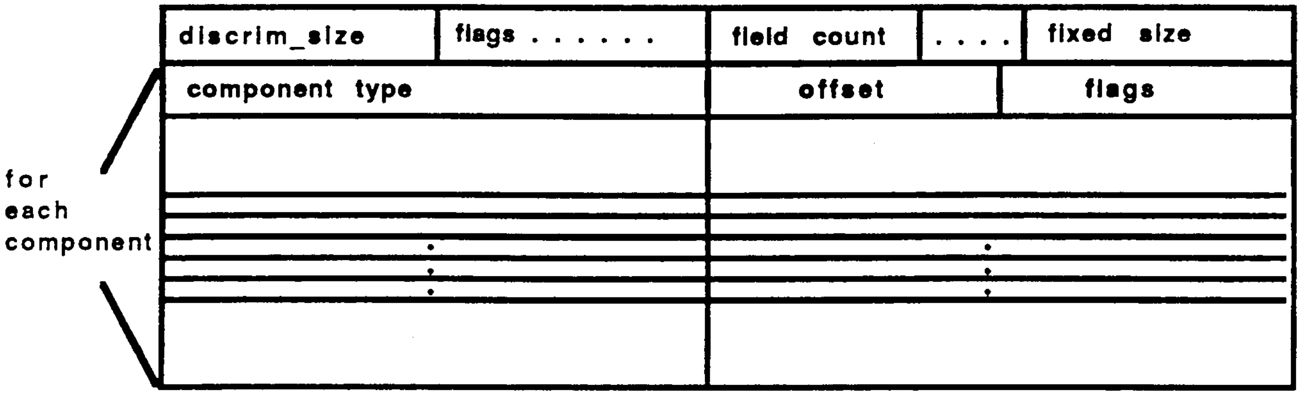
item size

slice size

repeated
for each
dimension

OBJECTS - TYPE DESCRIPTORS (CONT'D)

- For a record type:



discrim_size | flags . . . . . . | field count | . . . . | fixed size

component type | offset | flags

for each component

- Each reference to a type is a pointer to one of these descriptors.

- Most Ada type declarations result in the creation of one (or more) of these type descriptors.

- Many operations on objects require that the descriptor be examined.

OBJECTS - TYPE DESCRIPTORS (CONT'D)

- Sometimes some type information is associated with an individual object rather than in the type descriptor.

- Variant record objects are more complicated and have more information associated with them. They are slower to create and access.

- Variant objects can be constrained or unconstrained. An unconstrained object can assuem any of its variants. A constrained object cannot - it can have only one specific variant.

OBJECTS - ADA EXAMPLES

```
type Array_Type is array (integer range <>) of character;

type T (x: integer:=10; b:boolean:=false) is record

        A: Array_Type (1..x);

        case B is
              when true => F: integer
              when false => F2: Array_type (1..500)
        end case

end record;


V: T (1000,false);              --    big constrained object
W: T (1,true);                  --    small constrained object
X: T;                           --    unconstrained, giant object
```

- Default initial values versus constraints

```
subtype T1 is T (100,true);             --    constraints
V: T (1000,false);                      --    constraints
```

OBJECTS - REPRESENTATION OF VALUES

- For discrete and other "simple" objects, the representation is simple binary. Only the number of bits needed to represent the value are used. Values can be unsigned or two's complement (the type indicates which).

- Examples:

X: integer;	32 bits
b: boolean:	1 bit unsigned
type E is (A,B,C);	
V:E	2 bits unsigned
C: character;	8 bits - special case
I: long integer	64 bits

- Records and arrays are fully packed. (More details on this later)

- "Scalar" objects have simple, fixed representations

- Records with no discriminants are also simply represented by the values of each of the fields

OBJECTS - REPRESENTATION OF VALUES (CONT'D)

- Array and variant record objects may require additional descriptive information
  - There is an indicator of whether the value is constrained (the "is_constrained bit")
  - There is a "variant clause index" indicating which specific variant is active
  - There may be array bounds information for array objects ("bounds with object")

- See handout #1 - Note the following:
  - Objects of constrained array (sub)types do not have extra bits
  - Unconstrained values have descriptors that indicate their current configuration
  - Constants also have such descriptors

```
                                                        5+1sign = 6 bits
    subtype Short_Int is Integer range 0 .. 20;
    type T1 (D1 : Boolean := False; D2 : Short_Int := 20) is
        record
            F1 : Natural := 100;
            case D1 is
                when True =>
                    F2 : Boolean := True;
                when False =>
                    F4 : String (1 .. D2);
            end case;
        end record;

    V1 : T1;

type T1 size     =  334 bits
Object V1 size   =  334 bits.
Object fits in  42 bytes.
0128000001900000 0100000003800000 0004000000500000 0000000000000000
0000000000000000 0002

    subtype T2 is T1 (D1 => True, D2 => 3);
    V2 : T2;

type T2 size   =  47 bits
Object V2 size =  47 bits.
Object fits in  6 bytes.
80C7FFFFFFFE

    subtype T3 is String (1 .. 5);
    V3 : T3 := "abcde";

type String size  =  4294967359
type T3 size      =  40

Object "abc" size =  88 bits.       64 bits of bound, 24 bits of data (for constants)
Object fits in  11 bytes.
0000000100000003 616263

Object V3 size    =  40 bits.
Object fits in  5 bytes.
6162636465
```

OBJECTS - OTHER KINDS

- Not all R1000 architectural objects are Ada data objects. Others include:

  - Subprograms
  - Packages and tasks
  - Exceptions
  - "Select" objects
  - Task entries

  See handout #2 for a full list

```
The Kind field is 7 bits.

  0 Discrete_Var,
  1 Control_State,
  2 Subprogram_Ref_For_Call,
  3 Word3_Flag,
  4 Discrete_Ref,
  5 Module_Key,
  6 Subprogram_For_Call,
  7 Mark_Word_Flag,
  8 Float_Var,
  A Variable_Ref,
  C Float_Ref,
  D Deletion_Key,
  E Entry_Var,
 10 Access_Var,
 11 Static_Connection,
 12 Subprogram_Ref_For_Call_Elaborated,
 14 Access_Ref,
 15 Interface_Key,
 16 Subprogram_For_Call_Elaborated,
 18 Task_Var,
 19 Dependence_Link,
 1C Task_Ref,
 1E Select_Var,
 1F Auxiliary_Mark,
 21 Micro_State1,
 22 Subprogram_Ref_For_Call_Visible,
 24 Subvector_Var,
 26 Subprogram_For_Call_Visible,
 29 Micro_State2,
 2C Subarray_Var,
 2E Family_Var,
 32 Subprogram_Ref_For_Call_Visible_Elaborated,
 36 Subprogram_For_Call_Visible_Elaborated,
 38 Heap_Access_Var,
 3C Heap_Access_Ref,
 3E Default_Var,
 3F Activation_Link,
 41 Control_Allocation,
 42 Accept_Subprogram_Ref,
 44 Record_Var,
 46 Accept_Subprogram,
 49 Scheduling_Allocation,
 4C Variant_Record_Var,
 4E Delay_Alternative,
 52 Interface_Subprogram_Ref,
 56 Interface_Subprogram,
 58 Package_Var,
 5F Accept_Link,
 6C Vector_Var,
 6E Family_Alternative,
 74 Matrix_Var,
 76 Null_Subprogram,
 7C Array_Var,
 7E Exception_Var,
 7F Activation_State
```

OBJECTS - BASIC MACHINE PREDEFINED TYPES

NAME	USE
Discrete	scalar types (integer, fixed point, enumeration)
Float	floating point
Access	pointers (access types)
Package	Ada packages
Task	Ada tasks
Array	arrays
Record	records
Variant_Record	variant records
Select	internal state for select statement
Entry	task entry
Family	family of task entries
Exception	Ada exceptions
Subprogram	represent machine callable entities

- These are called "operand classes". If an instruction tries to operate on the wrong kind of object, the exception "operand_class_error" is raised.

OBJECTS - TASK/PACKAGE OBJECTS

- Tasks and packages are treated almost identically in the R1000 architecture

- While a package is elaborating it runs on its own "thread" of control

- Each package and task has its own id which is unique on that machine

- The task id (or more generally, module id) is the "value" part of a task or package object

- Each module takes several pages, minimum
  task or package

- Packages declare visible types, subprograms and data objects. Each of these exists at a specific offset within the package. marked visible or not

- Tasks are similar, but can only declare visible entries to be called

- Operations on tasks and packages include calling the entries of a task or the visible subprograms of a package

- More on this later

R1000 STORAGE ORGANIZATION - INTRODUCTION

- A module (package or task) is the basic container of information

- Each module consists of several memory <u>segments</u>
  A segment is a contiguous region of virtual memory

- The basic segments for a module are:

  - Type stack - contains type descriptions for types declared in the module

  - Control stack - contains task control block (TCB), for packages also
    visible objects, and procedure call stack

  - Data stack - contains value part of arrays and records. Operates in parallel with control stack.

  - Import space - used to address non-local objects. More later.

  - Queue space - used to store entry queue information (tasks only)

R1000 STORAGE ORGANIZATION - SEGMENTS

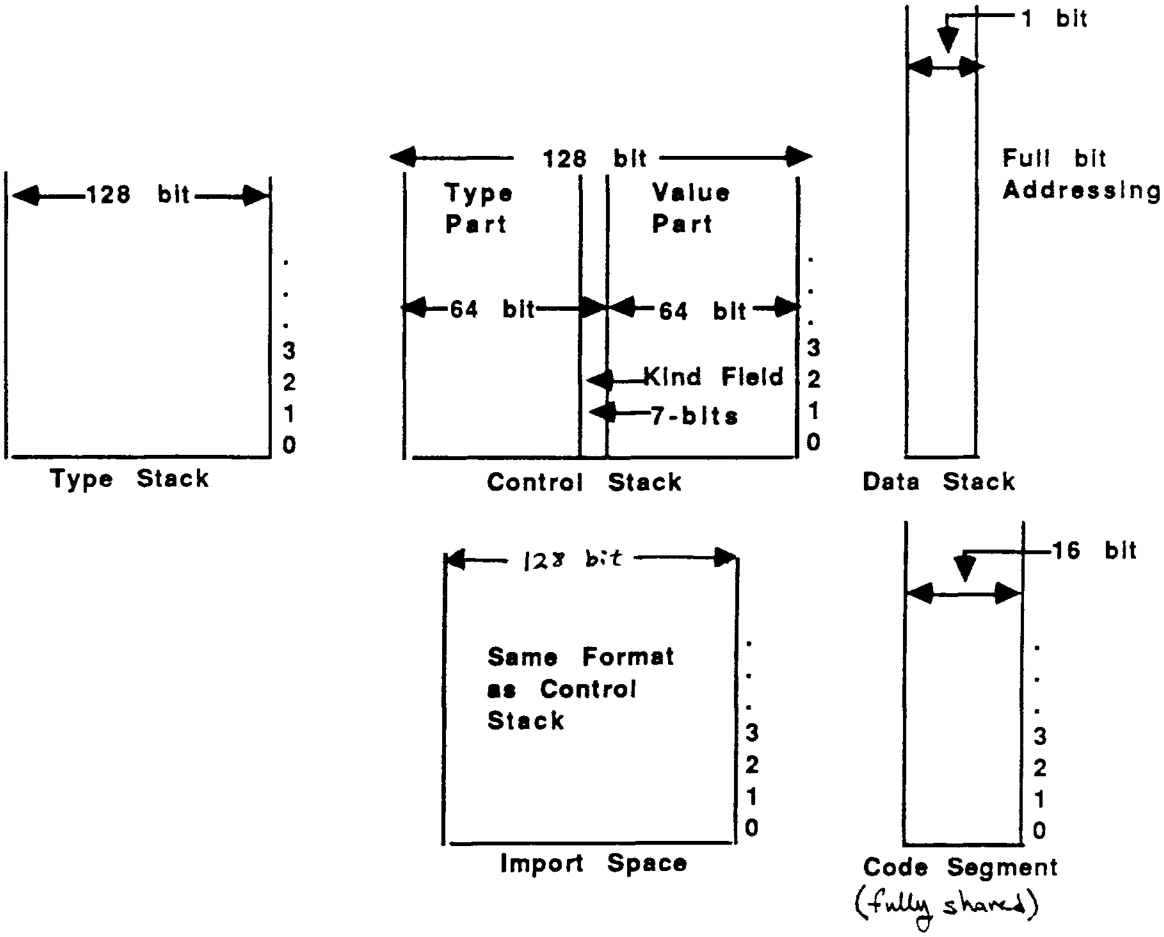
- In addition to Control, Type, Data, Import and Queue segments, there are also:

  - Code segments - which contain instructions to be executed

  - System segments - which are used for internal machine state

- Segments (memory in general) are bit addressable. Memory address bus addresses are always bit addresses.

- From a software standpoint, some segments have common word addressing structure.



OBJECTS - TYPE OBJECTS

- Same as a normal object, but value part is null
  - Zero for discrete and float objects
  - Null pointer for record and array objects
  - Null task id for module objects
- Generic packages and task types are examples of module types
- Each package has such a type (even for non-generic packages)

OBJECTS - REVIEW

Type part and value part

Type descriptions

Variant records - constrained vs unconstrained

Basic machine types

Packages and task objects

Objects of type "type"

Key areas of impact:

- Performance: costs in size and access time
- Unchecked_Conversion: bounds and constrained bits are visible
- * Debugging: misuse of unconstrained objects is a common customer mistake and has big impact on system performance and disk use

R1000 STORAGE ORGANIZATION - SEGMENTS (CONT'D)

- Each segment has a limit to its size:

Segment	Unit	Unit Limit	Byte Limit
Control stack	128 bit word	2^{20} (1M) words	16 MB
Type stack	128 bit word	2^{20} (1M) words	16 MB
Data stack	Bit	2^{32} (4G) bits	512 MB
Import	128 bit word	2^{20} (1M) words	16 MB
Code	16 bit instr	32K instructions	64 KB

- Control stack depth limits recrusive calls. An infinite recrusion will consume 16 MB, then raise storage error

* Debug tables can extend up to 128 KB
in Code segment

R1000 STORAGE ORGANIZATION - PHYSICAL MEMORY

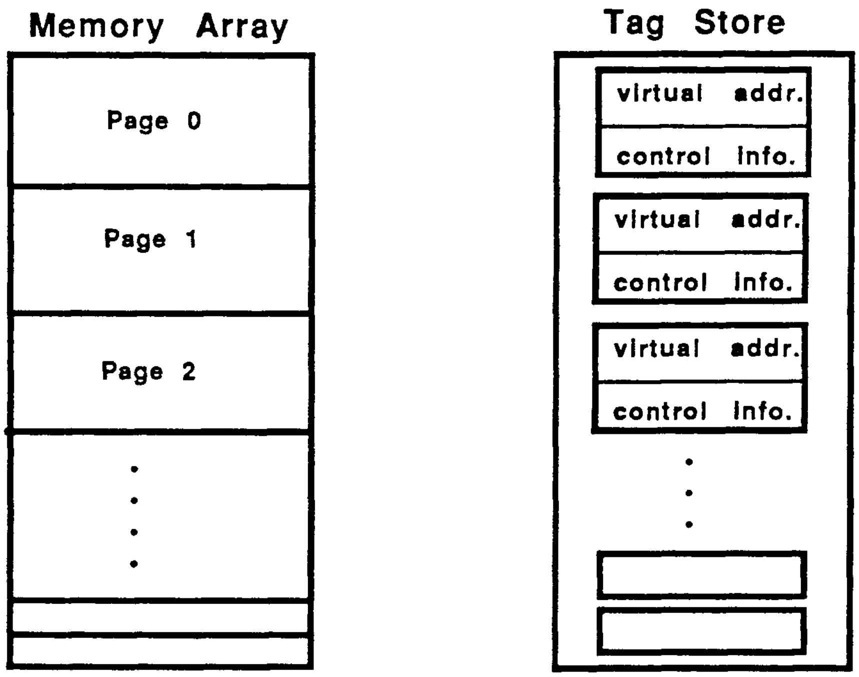
- R1000 physical memory consists of four 8 MB memory arrays.

- Memory is accessed in words of 128 bits (16 bytes) ECC is done on this unit.

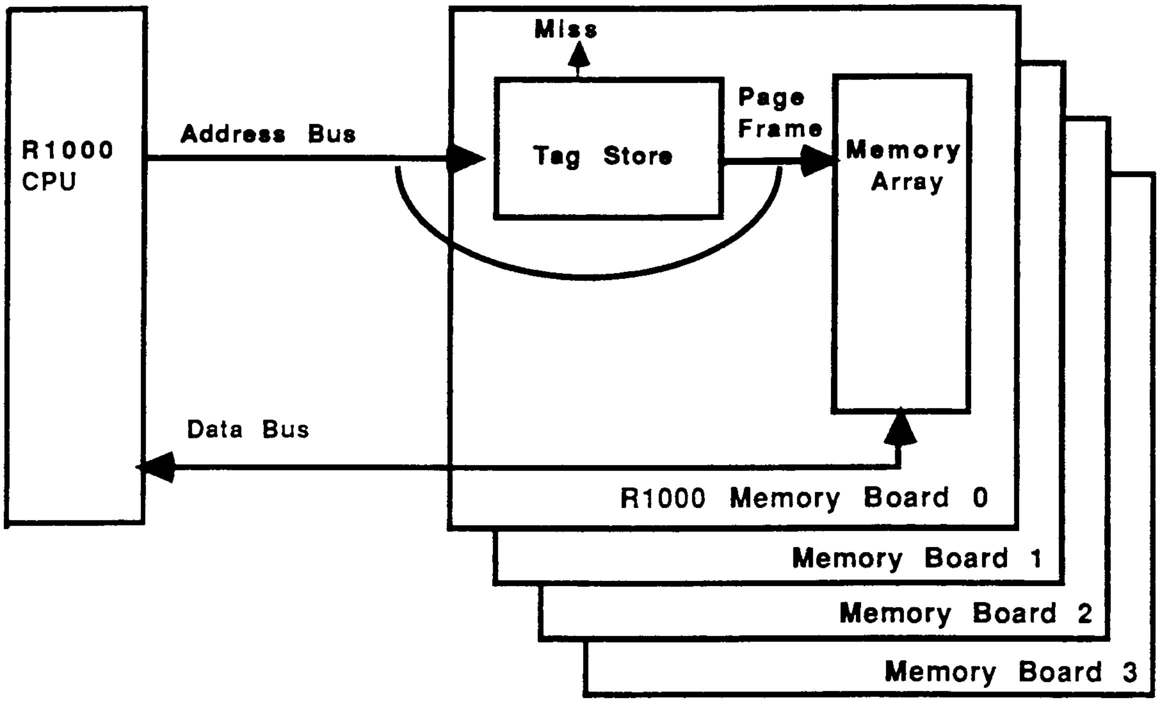
- Memory modules respond directly to virtual addresses; there is no virtual -> physical address map.

- A Tag Store on each board identifies which virtual locations are present. Associative

- Memory is further divided into 1024 byte pages.



R1000 STORAGE ORGANIZATION - PHYSICAL MEMORY (CONT'D)



- On a memory access, if all four boards generate a "miss" then the virtual address is not present in the physical memory and a page fault is signalled.

- The physical memory operates as a "cache" for virtual pages from disk.

- All tags on all boards are searched in parallel for fast access.

R1000 STORAGE ORGANIZATION - REVIEW

- 7 segment ~~bytes~~ types, each for different purpose
- Physical memory organization: tag store, ECC, 1K pages 128 bit bus
- Segment size limits

INSTRUCTION SET - INTRODUCTION

- Code stored in code segments
- 16 bit basic instructions; some 32 bit
- Stack oriented instructions
  - load (push)
  - load (push)
  - Add – no addresses in operator instruction
  - Store (pop)
- Basic instruction breakdown
  - Declarative instructions (4) - create types, variable, subprograms
  - Stack instructions (9) - push, pop
  - Subprogram call (3) - call, exit, end redezvous
  - Classed action (2) - operate on typed operands
  - Unclassed action (1) - operate on untyped operands
  - Branch (6) - jump, case branch, counting loop
  - Literal (2) - load constants
  - Special (5) - markers, not really instructions

INSTRUCTION SET - CODE SEGMENT STRUCTURE

- Code segment consists of a header, a series of subprograms, and a debug table

- The header is the first 8 words (16 bit words) and has the following structure:

	0	4	8 15
0	version of machine code		
1	Diana version	RCG major version	RCG minor version
2	0		
3	offset of debug table		
4	default exception handler - raise instruction		
5	module termination instruction - signal completion		
6	offset to segment table (only in elab segments)		
7	0	wired	# pages in seg - 1

- Each subprogram has a 3 word header and begins on a 128 bit word boundary. There may be a few unused words at the end of a subprogram so that the next starts at the proper boundary.

  The header is:

  XXX0 offset of begin (first statement)
  XXX1 offset of exception handler
  XXX2 number of locals
  XXX3 first instruction (entry point) of subprogram

INSTRUCTION SET - GETTING CODE LISTINGS

- Controlled by switches

R1000_CG. Seg_Listing	- controls code segment listing
R1000_CG. Elab_Order_Listing	- shows order of module elaboration
R1000_CG. ASM_Listing	- produces earlier code generating listing - less useful
•TTY_Echo	

- Don't forget to do Switches.Associate when you create a new switch file

- When the unit is coded, files will be created as children of the Ada unit:

```
Foo                              : C Proc_Spec
Foo                              : C Main_Body
 .<Code_Listing>                 : File
 .<Elaboration_Order_Listing>    : File
 .<Elaboration_Code_Listing>     : File
```

INSTRUCTION SET - SAMPLE

Short_Literal n	–	push n on the control stack
Load_Encached n	–	push contents of "special" register on the control stack
Declare_Type . . .	–	create a new type (pushes a control word)
Complete_Type . . .	–	fill in more info. about a type
Declare_Variable. . .	–	create a new object (pushes a control word)
Declare subprogram	–	create a new subprogram (32 bit instruction)
Load lex, offset	–	push a copy of the specified word on the stack
Store lex, offset	–	pop a word and store into the specified location
Execute class,field_write_op	--	store into a record, array, or package field
Exit_Subprogram n	–	return and pop n words off the control stack
Action accept_activation	–	finished package is visible part of elaboration
Action signal_activation	–	finished package elaboration, starting body statements
Action break_optional	–	debugger statement marker inserted only if no other way for debugger to find statement boundary

INSTRUCTION SET - OVERVIEW

- Declarative instructions create types, objects and subprograms. There are many flavors of these instructions

  - Declare_subprogram
  - Declare_variable <class> <options>
  - Declare_type <class> <options>
  - Complete_type <class> <options>

- Notes

  <class> refers to the object class. For example: record, array, task, etc.

INSTRUCTION SET - OVERVIEW (CONT'D)

- **Stack Instructions**

Load lex level, offset	
Store lex level, offset	
Store_unchecked lex level, offset	– no constraint check
Reference lex level, offset	– push a reference rather than a value
Store_top <class> <offset>	– tos relative address
Store_top_unchecked <offset>	
Load_top <offset>	– tos relative address
Pop_control count	
Load_encached <register#>	– magic values

* tos = Top of Stack

Continued following next page

```
Machine type summary

  Operand_Class
  -------------
  Discrete_Class             +, -, *, /, =, /=, etc
  Float_Class                +, -, *, /, =, /=, etc
  Access_Class
  Heap_Access_Class
  Package_Class              Declare, Activate, etc.
  Task_Class                 Declare, Abort, etc.
  Module_Class
  Array_Class                Field_Read/Write, Get_Bounds, etc.
  Vector_Class               Field_Read/Write, Get_Bounds, etc.
  Matrix_Class               Field_Read/Write, Get_Bounds, etc.
  Subarray_Class             Field_Read/Write, Get_Bounds, etc.
  Subvector_Class            Field_Read/Write, Get_Bounds, etc.
  Record_Class               Field_Read/Write
  Variant_Record_Class       Field_Read/Write
  Select_Class               call
  Entry_Class                Rendezvous
  Family_Class               Rendezvous
  Exception_Class            Raise, Test_Value
```

INSTRUCTION SET - OVERVIEW (CONT'D)

- **Subprogram call**

Call	lex level, offset
Exit_Subprogram	pop count, exit options
End_Rendezvous	parm count

also

Execute, Package, Field_Execute - - call to another package

- **Unclassed actions**

Action <operation>
signal_activated
swap_control
initiate_delay
etc.

General miscellaneous operations (about 56 of them).

- **Classed Actions - cover most operations on typed operands**

Execute	<class>	<operation>
	float	is_unsigned
	select	time_guard_write
	variant_record	field_read_dynamic
	module	check_elaborated
	array	concatenate
	discrete	not_equal
	etc.	

- Note: Vector, Subvector, Subarray, and Matrix are special kinds of Array classes.

Handout

```
package Machine_Code is

    pragma Subsystem (Ada_Base);

    pragma Module_Name (4, 67);


    Version : constant := 15;


    type Operand_Class is (Discrete_Class, Float_Class, Access_Class,
                           Heap_Access_Class, Package_Class,
                           Task_Class, Module_Class, Array_Class,
                           Vector_Class, Matrix_Class, Subarray_Class,
                           Subvector_Class, Record_Class,
                           Variant_Record_Class, Select_Class, Entry_Class,
                           Family_Class, Exception_Class, Any_Class);


    type Operation is (Equal_Op, Not_Equal_Op, Greater_Op, Less_Op,
                       Greater_Equal_Op, Less_Equal_Op, Case_Compare_Op,
                       Below_Bound_Op, Above_Bound_Op, In_Range_Op,
                       Not_In_Range_Op, Case_In_Range_Op, Bounds_Check_Op,
                       Reverse_Bounds_Check_Op, Bounds_Op, Reverse_Bounds_Op,
                       First_Op, Last_Op, Set_Value_Op, Set_Value_Unchecked_Op,
                       Set_Value_Visible_Op, Set_Value_Visible_Unchecked_Op,
                       Test_And_Set_Previous_Op, Test_And_Set_Next_Op,
                       Successor_Op, Predecessor_Op, And_Op, Or_Op, Xor_Op,
                       Complement_Op, Unary_Minus_Op, Absolute_Value_Op,
                       Plus_Op, Minus_Op, Times_Op, Divide_Op, Modulo_Op,
                       Remainder_Op, Minimum_Op, Maximum_Op, Exponentiate_Op,
                       Binary_Scale_Op, Multiply_And_Scale_Op,
                       Divide_And_Scale_Op, Partial_Plus_Op, Partial_Minus_Op,
                       Arithmetic_Shift_Op, Logical_Shift_Op,
                       Rotate_Op, Insert_Bits_Op, Extract_Bits_Op,
                       Is_Unsigned_Op, Count_Nonzero_Bits_Op,
                       Count_Leading_Zeros_Op, Count_Trailing_Zeros_Op,
                       Round_To_Discrete_Op, Truncate_To_Discrete_Op,
                       Convert_From_Discrete_Op, Equal_Zero_Op,
                       Not_Equal_Zero_Op, Greater_Zero_Op, Less_Zero_Op,
                       Greater_Equal_Zero_Op, Less_Equal_Zero_Op, Is_Null_Op,
                       Not_Null_Op, Set_Null_Op, Element_Type_Op, All_Read_Op,
                       All_Write_Op, All_Reference_Op, Convert_Reference_Op,
                       Allow_Deallocate_Op, Deallocate_Op, Hash_Op,
                       Construct_Segment_Op, Get_Segment_Op, Get_Offset_Op,
                       Diana_Tree_Kind_Op, Diana_Map_Kind_To_Vci_Op,
                       Diana_Arity_For_Kind_Op, Diana_Allocate_Tree_Node_Op,
                       Diana_Put_Node_On_Seq_Type_Op,
                       Diana_Seq_Type_Get_Head_Op,
                       Diana_Find_Permanent_Attribute_Op, Diana_Spare0_Op,
                       Diana_Spare1_Op, Diana_Spare2_Op,
                       Adaptive_Balanced_Tree_Lookup_Op,
                       Elaborate_Op, Check_Elaborated_Op,
                       Augment_Imports_Op, Activate_Op, Abort_Op,
                       Abort_Multiple_Op, Rendezvous_Op, Count_Op,
                       Is_Callable_Op, Is_Terminated_Op,
                       Timed_Duration_Write_Op, Terminate_Guard_Write_Op,
                       Timed_Guard_Write_Op, Length_Op,
                       Structure_Write_Op, Structure_Clear_Op,
                       Slice_Read_Op, Slice_Write_Op,
```

```
                       Slice_Reference_Op, Catenate_Op,
                       Append_Op, Prepend_Op, Subarray_Op,
                       Component_Offset_Op, Is_Constrained_Op,
                       Is_Constrained_Object_Op, Make_Constrained_Op,
                       Indirects_Appended_Op, Structure_Query_Op,
                       Reference_Makes_Copy_Op, Read_Variant_Op,
                       Read_Discriminant_Constraint_Op,
                       Field_Read_Op, Field_Write_Op,
                       Field_Reference_Op, Field_Append_Op,
                       Field_Type_Op, Field_Constrain_Op,
                       Set_Bounds_Op, Set_Variant_Op,
                       Field_Execute_Op, Entry_Call_Op,
                       Conditional_Call_Op, Timed_Call_Op,
                       Family_Call_Op, Family_Timed_Op,
                       Family_Conditional_Op, Guard_Write_Op,
                       Member_Write_Op, Write_Unchecked_Op,
                       Field_Read_Dynamic_Op, Field_Write_Dynamic_Op,
                       Field_Type_Dynamic_Op, Field_Execute_Dynamic_Op,
                       Field_Reference_Dynamic_Op,
                       Reference_Lex_1_Op, Convert_Op,
                       Convert_Unchecked_Op, Convert_To_Formal_Op,
                       Check_In_Type_Op, Check_In_Formal_Type_Op,
                       Check_In_Integer_Op, In_Type_Op,
                       Not_In_Type_Op, Address_Op, Size_Op,
                       Address_Of_Type_Op, Make_Root_Type_Op,
                       Set_Constraint_Op, Is_Value_Op,
                       Is_Scalar_Op, Is_Default_Op,
                       Make_Visible_Op, Make_Aligned_Op,
                       Run_Utility_Op, Change_Utility_Op,
                       Run_Initialization_Utility_Op,
                       Has_Default_Initialization_Op,
                       Has_Repeated_Initialization_Op,
                       Is_Initialization_Repeated_Op, Raise_Op, Reraise_Op,
                       Get_Name_Op, Is_Constraint_Error_Op,
                       Is_Numeric_Error_Op, Is_Program_Error_Op,
                       Is_Tasking_Error_Op, Is_Storage_Error_Op,
                       Is_Instruction_Error_Op, Instruction_Read_Op,
                       Spare14_Op, Spare15_Op, Spare16_Op,
                       Spare17_Op, Spare18_Op, Spare19_Op,
                       Spare20_Op, Spare21_Op, Spare22_Op,
                       Spare23_Op, Spare24_Op, Spare25_Op);


    type Type_Sort is (Defined, Constrained, Incomplete,
                       Defined_Incomplete, Constrained_Incomplete);

    type Type_Option_Set is
        record
            Is_Visible : Boolean;
            Unsigned : Boolean;
            With_Size : Boolean;
            Accesses_Protected : Boolean;
            Values_Relative : Boolean;
            Bounds_With_Object : Boolean;
            Not_Elaborated : Boolean;
        end record;


    type Type_Completion_Mode is
```

#4

```
        (By_Defining, By_Renaming, By_Constraining, By_Constraining_Incomplete,
         By_Completing_Constraint, By_Component_Completion);


    type Variable_Option_Set is
        record
            Is_Visible : Boolean;
            Duplicate : Boolean;
            By_Allocation : Boolean;
            With_Constraint : Boolean;
            With_Subtype : Boolean;
            With_Value : Boolean;
            As_Component : Boolean;
            On_Processor : Boolean;
            Choice_Open : Boolean;
        end record;


    type Subprogram_Sort is (For_Call, For_Outer_Call,
                             For_Accept, Null_Subprogram);

    type Subprogram_Option_Set is
        record
            Is_Visible : Boolean;
            Not_Elaborated : Boolean;
            With_Address : Boolean;
        end record;


    type Exit_Option_Set is
        record
            With_Result : Boolean;
            From_Utility : Boolean;
        end record;




    Field_Index_Size : constant Integer := 8;

    type Field_Index is new Integer range 0 .. 2 ** Field_Index_Size - 1;
    subtype Variant_Record_Index is Field_Index range 1 .. Field_Index'Last;

    No_Variants : constant Variant_Record_Index := Field_Index'Last;

    type Field_Sort is (Fixed, Variant);
    type Field_Mode is (Direct, Indirect);


    subtype Field_Op is Operation range Field_Read_Op .. Member_Write_Op;
    subtype Component_Op is Field_Op range Field_Read_Op .. Field_Append_Op;

    type Access_Spec (Op : Operation := Equal_Op) is
        record
            case Op is
                when Component_Op =>
                    Kind : Field_Sort;
                    Mode : Field_Mode;

                when others =>
```

```
                    null;
            end case;
        end record;

    type Field_Spec (Class : Operand_Class := Discrete_Class;
                     Op : Operation := Equal_Op) is
        record
            case Class is
                when Package_Class | Task_Class =>
                    Offset : Field_Index;

                when Record_Class | Select_Class =>
                    Number : Field_Index;

                when Variant_Record_Class =>
                    Index : Field_Index;
                    Component : Access_Spec (Op);

                when others =>
                    null;
            end case;
        end record;

    type Operator_Spec (Class : Operand_Class := Discrete_Class;
                        Op : Operation := Equal_Op) is
        record
            case Op is
                when Field_Op =>
                    Field : Field_Spec (Class, Op);

                when others =>
                    null;
            end case;
        end record;




    type Lexical_Level is new Integer range 0 .. 15;
    type Lexical_Delta is new Integer range -256 .. 511;

    subtype Scope_Delta is Lexical_Delta range 0 .. 511;
    subtype Frame_Delta is Lexical_Delta range -256 .. 255;

    type Object_Reference is
        record
            Level : Lexical_Level;
            Offset : Lexical_Delta;
        end record;


    type Inner_Frame_Delta is new Integer range 0 .. 2 ** 8 - 1;

    type Parameter_Count is new Integer range 0 .. 2 ** 8 - 1;

    type Pc_Offset is new Integer range -2 ** 10 .. 2 ** 10 - 1;

    subtype Loop_Offset is Pc_Offset range -2 ** 9 .. -1;

    type Case_Maximum is new Integer range 0 .. 2 ** 9 - 1;
    type Stack_Top_Offset is new Integer range -6 .. 0;

    type Stack_Pop_Count is new Integer range 1 .. 7;

    type Encached_Object_Number is new Integer range 0 .. 31;

    type Symbolic_Label is new Integer;


    type Segment_Index is new Integer range 0 .. 2 ** 3 - 1;
    type Segment_Displacement is new Integer range 0 .. 2 ** 12 - 1;

    type Segment_Reference is
        record
            Offset : Segment_Displacement;
            Index : Segment_Index;
        end record;



    type Short_Literal_Value is new Integer range -2 ** 10 .. 2 ** 10 - 1;

    type Partial_Structure is new Long_Integer range 0 .. 2 ** 32 - 1;

    type Structure_Literal_Value is
        record
            High : Partial_Structure;
            Low : Partial_Structure;
        end record;

    type Literal (Of_Kind : Operand_Class := Discrete_Class) is
        record
            case Of_Kind is
                when Discrete_Class =>
                    Discrete_Literal : Long_Integer;

                when Float_Class =>
                    Float_Literal : Float;

                when Any_Class =>
                    Structure_Literal : Structure_Literal_Value;

                when others =>
                    null;
            end case;
        end record;


    type Unsigned_Immediate_Value is new Integer range 0 .. 255;
    type Signed_Immediate_Value is new Integer range -128 .. 127;

    type Immediate_Value (Is_Signed : Boolean := True) is
        record
            case Is_Signed is
                when True =>
                    Signed_Value : Signed_Immediate_Value;

                when False =>
                    Unsigned_Value : Unsigned_Immediate_Value;
```

```
            end case;
        end record;


    type Extended_Literal is new Integer range 0 .. 2 ** 16 - 1;

    type Extension_Kind is (Is_Literal, Is_Location);

    type Instruction_Extension (Of_Kind : Extension_Kind := Is_Literal) is
        record
            case Of_Kind is
                when Is_Literal =>
                    Value : Extended_Literal;

                when Is_Location =>
                    Location : Segment_Reference;
            end case;
        end record;



    type Unclassed_Action is
       (Illegal, Idle,

        Elaborate_Subprogram, Check_Subprogram_Elaborated,

        Accept_Activation, Activate_Tasks, Activate_Heap_Tasks,
        Signal_Activated, Signal_Completion, Propagate_Abort,
        Set_Priority, Increase_Priority, Get_Priority,
        Initiate_Delay, Exit_Nullary_Function, Set_Block_Start,

        Make_Default, Make_Self, Make_Scope, Make_Parent, Name_Partner,

        Swap_Control, Mark_Auxiliary, Pop_Auxiliary,
        Pop_Auxiliary_Loop, Pop_Auxiliary_Range,

        Pop_Block, Pop_Block_With_Result,

        Break_Unconditional, Break_Optional, Exit_Break,
        Query_Break_Cause, Query_Break_Address, Query_Break_Mask,
        Alter_Break_Mask, Query_Frame, Establish_Frame,

        Loop_Increasing_Extended, Loop_Decreasing_Extended,
        Push_Discrete_Extended, Push_Float_Extended,
        Push_Structure_Extended, Push_String_Extended,
        Push_String_Extended_Indexed, Store_String_Extended,

        Jump_Extended, Jump_Zero_Extended, Jump_Nonzero_Extended,
        Jump_Dynamic, Jump_Zero_Dynamic, Jump_Nonzero_Dynamic,

        Load_Dynamic, Store_Dynamic, Call_Dynamic, Reference_Dynamic,

        Spare6_Action, Spare7_Action, Spare8_Action,
        Spare9_Action, Spare10_Action, Spare11_Action);



    type Op_Code is (Declare_Type, Complete_Type, Declare_Variable,
                     Declare_Subprogram, Load, Store, Store_Unchecked,
```

#4

```
                     Reference, Call, Exit_Subprogram, End_Rendezvous, Execute,
                     Action, Execute_Immediate, Jump, Jump_Zero, Jump_Nonzero,
                     Jump_Case, Loop_Increasing, Loop_Decreasing, Short_Literal,
                     Indirect_Literal, Pop_Control, Load_Top, Store_Top,
                     Store_Top_Unchecked, Load_Encached, Extension,
                     Literal_Value, Block_Begin, Block_Handler, End_Locals);


    type Instruction (For_Op : Op_Code := Action) is
        record
            case For_Op is
                when Declare_Type =>
                    Type_Class : Operand_Class;
                    Type_Kind : Type_Sort;
                    Type_Options : Type_Option_Set;

                when Complete_Type =>
                    Completion_Class : Operand_Class;
                    Completion_Mode : Type_Completion_Mode;

                when Declare_Variable =>
                    Variable_Class : Operand_Class;
                    Variable_Options : Variable_Option_Set;

                when Declare_Subprogram =>
                    Subprogram_Kind : Subprogram_Sort;
                    Subprogram_Options : Subprogram_Option_Set;

                when Load | Store | Store_Unchecked | Reference | Call =>
                    Object : Object_Reference;

                when Exit_Subprogram =>
                    New_Top_Offset : Inner_Frame_Delta;
                    Exit_Options : Exit_Option_Set;

                when End_Rendezvous =>
                    Return_Count : Parameter_Count;

                when Execute =>
                    Operator : Operator_Spec;

                when Action =>
                    To_Perform : Unclassed_Action;

                when Execute_Immediate =>
                    Discrete_Op : Operation;
                    With_Value : Immediate_Value;

                when Jump | Jump_Zero | Jump_Nonzero |
                     Loop_Increasing | Loop_Decreasing =>
                    Relative : Pc_Offset;

                when Jump_Case =>
                    Case_Max : Case_Maximum;

                when Pop_Control =>
                    Pop_Count : Stack_Pop_Count;

                when Load_Top =>
                    At_Offset : Stack_Top_Offset;
```

```
                when Store_Top | Store_Top_Unchecked =>
                    Target_Class : Operand_Class;
                    Target_Offset : Stack_Top_Offset;

                when Load_Encached =>
                    With_Number : Encached_Object_Number;

                when Short_Literal =>
                    Short_Value : Short_Literal_Value;

                when Indirect_Literal =>
                    Value_Class : Operand_Class;
                    Value_Relative : Pc_Offset;

                when Literal_Value =>
                    Value : Literal;

                when Extension =>
                    Argument : Instruction_Extension;

                when Block_Begin | Block_Handler =>
                    Location : Segment_Reference;

                when End_Locals =>
                    Offset : Extended_Literal;
            end case;
        end record;


    -- indices into the encaching registers for declarations from Standard

    Standard_Cache_Index : constant Encached_Object_Number := 0;
    Boolean_Cache_Index : constant Encached_Object_Number := 1;
    Integer_Cache_Index : constant Encached_Object_Number := 2;
    Natural_Cache_Index : constant Encached_Object_Number := 3;
    Positive_Cache_Index : constant Encached_Object_Number := 4;
    Long_Integer_Cache_Index : constant Encached_Object_Number := 5;
    Float_Cache_Index : constant Encached_Object_Number := 6;
    Duration_Cache_Index : constant Encached_Object_Number := 7;
    Character_Cache_Index : constant Encached_Object_Number := 8;
    String_Cache_Index : constant Encached_Object_Number := 9;
    Null_String_Cache_Index : constant Encached_Object_Number := 10;

    -- indices into the encaching registers for declarations from Diana

    Diana_Cache_Index : constant Encached_Object_Number := 11;
    Diana_Tree_Cache_Index : constant Encached_Object_Number := 12;
    Diana_Symbol_Rep_Cache_Index : constant Encached_Object_Number := 13;
    Diana_Seq_Type_Cache_Index : constant Encached_Object_Number := 14;
    Diana_Sequence_Cache_Index : constant Encached_Object_Number := 15;

    -- indices into the encaching registers for declarations from System
    Segment_Cache_Index : constant Encached_Object_Number := 16;

    -- more indices into the encaching registers for declarations from Diana

    Diana_Temp_Seq_Index : constant Encached_Object_Number := 17;
    Diana_Attr_List_Index : constant Encached_Object_Number := 18;
    Diana_Tree_Node_Index : constant Encached_Object_Number := 19;
    Diana_Seq_Type_Node_Index : constant Encached_Object_Number := 20;
    -- unused indices

    Unused_Index_21 : constant Encached_Object_Number := 21;
    Unused_Index_22 : constant Encached_Object_Number := 22;
    Unused_Index_23 : constant Encached_Object_Number := 23;
    Unused_Index_24 : constant Encached_Object_Number := 24;
    Unused_Index_25 : constant Encached_Object_Number := 25;
    Unused_Index_26 : constant Encached_Object_Number := 26;
    Unused_Index_27 : constant Encached_Object_Number := 27;
    Unused_Index_28 : constant Encached_Object_Number := 28;
    Unused_Index_29 : constant Encached_Object_Number := 29;
    Unused_Index_30 : constant Encached_Object_Number := 30;
    Unused_Index_31 : constant Encached_Object_Number := 31;

end Machine_Code;
```

INSTRUCTION SET - DEBUGGER

- You can display code and monitor PC values with the R1000 debugger.

  Memory_Display ("seg offset", count, "code")

  Memory_Display ("17634 250", 30, "code");

  ↑ hex segment name and offset

- Enable (Addresses)

  - Causes debugger to display PC and other machine information in debugger displays.

.Handi 1a

```
with Io;
with Job_Segment;

procedure Architecture_Example is
    package P1 is
        procedure Op (X : String);
    end P1;
    package body P1 is

        Local_Value : Integer := 33;
        procedure Local_Procedure (S : String) is
        begin
            Io.Put_Line (S & Integer'Image (Local_Value));
        end Local_Procedure;

        procedure Op (X : String) is
            Local_Variable : Integer;
            type A is array (1 .. 10) of Boolean;
            Ar : A;
            type Pa is access A;
            type Sa is access A;
            pragma Segmented_Heap (Sa);
            P1 : Pa := new A'(others => True);
            P2 : Sa := new A' (others => True);
            pragma Heap (Job_Segment.Get);

            procedure Nested (Parm : Natural) is
                Nested_Local : Integer;
            begin
                Local_Variable := Parm;
                Nested_Local := Local_Variable;
                Local_Value := Parm;
                P2 (3) := False;
                null;
            end Nested;

        begin
            Local_Procedure (X & "hi");
            Nested (23);
        end Op;

    end P1;
-- hi
begin
    P1.Op ("this is a test");
    null;
end Architecture_Example;
pragma Main;
```

Handout # 4b

```
Code for segment  1033472


    0 0000: 000F
    1 0001: 5800
    2 0002: 0000
    3 0003: 00B8
    4 0004: 0100
    5 0005: 00BB
    6 0006: 0000
    7 0007: 0000
```

```
    8 0008: 0010  BLOCK_BEGIN       2,  0
    9 0009: 0004  BLOCK_HANDLER      0,  4
   10 000A: 0001  END_LOCALS     1

;;; module ARCHITECTURE_EXAMPLE
   11 000B: 029A  DECLARE_SUBPROGRAM     FOR_OUTER_CALL, IS_VISIBLE,
                                                                     NOT_ELABORATED
   12 000C: 001B  EXTENSION      3,  3      ;;; ARCHITECTURE_EXAMPLE
   13 000D: 00BF  ACTION     ACCEPT_ACTIVATION
   14 000E: 1D0D  EXECUTE_IMMEDIATE     REFERENCE_LEX_1_OP, 13
   15 000F: 00C7  ACTION     ELABORATE_SUBPROGRAM
   16 0010: 00BC  ACTION     SIGNAL_ACTIVATED
   17 0011: 00BB  ACTION     SIGNAL_COMPLETION
   18 0012: 0000
   19 0013: 0000
   20 0014: 0000
   21 0015: 0000
   22 0016: 0000
   23 0017: 0000
   0018: 0029  BLOCK_BEGIN       5,  1
   0019: 0004  BLOCK_HANDLER       0,  4
   001A: 0002  END_LOCALS      2

;;; subprogram ARCHITECTURE_EXAMPLE
;;; package P1 is
   001B: 4800  SHORT_LITERAL     0
   001C: 0093  EXTENSION     147
   001D: 0033  EXTENSION      6,  3      ;;; push full address of a location in
                                                          current code segment
   001E: 02A0  DECLARE_SUBPROGRAM    NULL_SUBPROGRAM
   001F: 038E  DECLARE_TYPE     PACKAGE_CLASS, DEFINED      ;;; P1
   0020: 0387  DECLARE_VARIABLE     PACKAGE_CLASS      ;;; P1

;;; package body P1 is
   0021: E001  LOAD      0, 1      ;;; DEFAULT
   0022: E002  LOAD      0, 2      ;;; JOB_SEGMENT
   0023: E003  LOAD      0, 3      ;;; IO
   0024: 4803  SHORT_LITERAL     3
   0025: E402  LOAD      2, 2      ;;; P1      ;;; P1
   0026: 020E  EXECUTE      MODULE_CLASS, AUGMENT_IMPORTS_OP
   0027: 00D8  LOAD_TOP      0      ;;; ( 2, 2)      ;;; P1
   0028: 020F  EXECUTE      MODULE_CLASS, ACTIVATE_OP

;;; P1.OP ("this is a test");
;;; push parameters
   0029: 0092  ACTION      PUSH_STRING_EXTENDED
   002A: 00A8  EXTENSION      21,  0
   002B: E402  LOAD      2, 2      ;;; P1
   002C: 180D  EXECUTE      PACKAGE_CLASS, FIELD_EXECUTE_OP, 13      ;;; OP

;;; null;
   002D: 4501  EXIT_SUBPROGRAM      1
   002E: 0000
   002F: 0000
```

```
   0030: 003C  BLOCK_BEGIN       7,  4
   0031: 0004  BLOCK_HANDLER       0,  4
   0032: 000F  END_LOCALS      15

;;; module P1
;;; procedure OP (X : STRING);
   0033: 029A  DECLARE_SUBPROGRAM     FOR_OUTER_CALL, IS_VISIBLE,
                                                               NOT_ELABORATED
   0034: 0043  EXTENSION      8,  3      ;;; OP
   0035: 00BF  ACTION      ACCEPT_ACTIVATION

;;; LOCAL_VALUE : INTEGER := 33;
   0036: 00E2  LOAD_ENCACHED      2      ;;; INTEGER
   0037: 0621  EXECUTE_IMMEDIATE     SET_VALUE_UNCHECKED_OP, 33

;;; procedure LOCAL_PROCEDURE (S : STRING) is
   0038: 029D  DECLARE_SUBPROGRAM     FOR_OUTER_CALL
   0039: 008B  EXTENSION      17,  3      ;;; LOCAL_PROCEDURE

;;; procedure OP (X : STRING) is
   003A: 1D0D  EXECUTE_IMMEDIATE     REFERENCE_LEX_1_OP, 13
   003B: 00C7  ACTION      ELABORATE_SUBPROGRAM
   003C: 00BC  ACTION      SIGNAL_ACTIVATED
   003D: 00BB  ACTION      SIGNAL_COMPLETION
   003E: 0000
   003F: 0000
```

#4b

```
   64 0040: 0079  BLOCK_BEGIN      15,  1
   65 0041: 0004  BLOCK_HANDLER     0,  4
   66 0042: 0009  END_LOCALS     9

;;; subprogram OP
;;; LOCAL_VARIABLE : INTEGER;
   67 0043: 00E2  LOAD_ENCACHED     2      ;;; INTEGER

;;; type A is array (1 .. 10) of BOOLEAN;
   68 0044: 4801  SHORT_LITERAL     1      ;;; 1
   69 0045: 480A  SHORT_LITERAL     10      ;;; 10
   70 0046: 00E1  LOAD_ENCACHED     1      ;;; BOOLEAN
   71 0047: 02A0  DECLARE_SUBPROGRAM    NULL_SUBPROGRAM
   72 0048: 4801  SHORT_LITERAL     1
   73 0049: 035D  DECLARE_TYPE    ARRAY_CLASS, DEFINED      ;;; A

;;; AR : A;
   74 004A: 00D8  LOAD_TOP     0      ;;; ( 2, 3)      ;;; A
   75 004B: 0337  DECLARE_VARIABLE    ARRAY_CLASS

;;; type PA is access A;
   76 004C: E403  LOAD     2, 3      ;;; A
   77 004D: 02A0  DECLARE_SUBPROGRAM    NULL_SUBPROGRAM
   78 004E: 4818  SHORT_LITERAL     24
   79 004F: 03D5  DECLARE_TYPE    ACCESS_CLASS, DEFINED      ;;; PA

;;; type SA is access A;
   80 0050: E403  LOAD     2, 3      ;;; A
   81 0051: 02A0  DECLARE_SUBPROGRAM    NULL_SUBPROGRAM
   82 0052: 03AD  DECLARE_TYPE    HEAP_ACCESS_CLASS, DEFINED      ;;; SA

;;; pragma SEGMENTED_HEAP (SA);
   83 0053: 0007  ACTION     BREAK_OPTIONAL

;;; P1 : PA := new A'(others => TRUE);
   84 0054: E405  LOAD     2, 5      ;;; PA
   85 0055: E403  LOAD     2, 3      ;;; A
   86 0056: 0337  DECLARE_VARIABLE    ARRAY_CLASS
   87 0057: 480A  SHORT_LITERAL     10
   88 0058: 4801  SHORT_LITERAL     1
   89 0059: 4801  SHORT_LITERAL     1      ;;; TRUE
   90 005A: E40A  LOAD     2, 10
   91 005B: E408  LOAD     2, 8
   92 005C: 01D6  EXECUTE    VECTOR_CLASS, FIELD_WRITE_OP
   93 005D: 3FFB  LOOP_INCREASING    -5
   94 005E: E403  LOAD     2, 3      ;;; A
   95 005F: 01C3  EXECUTE    VECTOR_CLASS, CHECK_IN_TYPE_OP      ;;; A'(others =>
                                                                          TRUE)
   96 0060: E405  LOAD     2, 5      ;;; PA
   97 0061: 03B6  DECLARE_VARIABLE    ACCESS_CLASS, BY_ALLOCATION, WITH_VALUE
                                                          ;;; new A'(others => TRUE)
   98 0062: C407  STORE     2, 7

;;; P2 : SA := new A'(others => TRUE);
   99 0063: E406  LOAD     2, 6      ;;; SA
  100 0064: E403  LOAD     2, 3      ;;; A
  101 0065: 0337  DECLARE_VARIABLE    ARRAY_CLASS
  102 0066: 480A  SHORT_LITERAL     10
  103 0067: 4801  SHORT_LITERAL     1
```

```
  104 0068: 4801  SHORT_LITERAL     1      ;;; TRUE
  105 0069: E40B  LOAD     2, 11
  106 006A: E409  LOAD     2, 9
  107 006B: 01D6  EXECUTE    VECTOR_CLASS, FIELD_WRITE_OP
  108 006C: 3FFB  LOOP_INCREASING    -5
  109 006D: E403  LOAD     2, 3      ;;; A
  110 006E: 01C3  EXECUTE    VECTOR_CLASS, CHECK_IN_TYPE_OP      ;;; A'(others =>
                                                                          TRUE)
  111 006F: E406  LOAD     2, 6      ;;; SA

;;; push parameters
  112 0070: E001  LOAD     0, 1      ;;; DEFAULT
  113 0071: 1811  EXECUTE    PACKAGE_CLASS, FIELD_EXECUTE_OP, 17      ;;; PROCESS
  114 0072: E002  LOAD     0, 2      ;;; JOB_SEGMENT
  115 0073: 1811  EXECUTE    PACKAGE_CLASS, FIELD_EXECUTE_OP, 17      ;;; GET
  116 0074: 0396  DECLARE_VARIABLE    HEAP_ACCESS_CLASS, BY_ALLOCATION,
                                       WITH_VALUE      ;;; new A'(others => TRUE)
  117 0075: C408  STORE     2, 8

;;; pragma HEAP (JOB_SEGMENT.GET);
  118 0076: 0007  ACTION     BREAK_OPTIONAL

;;; procedure NESTED (PARM : NATURAL) is
  119 0077: 029F  DECLARE_SUBPROGRAM    FOR_CALL
  120 0078: 009B  EXTENSION     19,  3      ;;; NESTED

;;; LOCAL_PROCEDURE (X & "hi");
;;; push parameters
  121 0079: E5FF  LOAD     2, -1      ;;; X
  122 007A: 0092  ACTION     PUSH_STRING_EXTENDED
  123 007B: 00A9  EXTENSION     21,  1
  124 007C: 01CC  EXECUTE    VECTOR_CLASS, CATENATE_OP
  125 007D: 820F  CALL     1, 15      ;;; LOCAL_PROCEDURE

;;; NESTED (23);
;;; push parameters
  126 007E: 4817  SHORT_LITERAL     23      ;;; 23
  127 007F: 8409  CALL     2, 9      ;;; NESTED
  128 0080: 4502  EXIT_SUBPROGRAM     2
  129 0081: 0000
  130 0082: 0000
  131 0083: 0000
  132 0084: 0000
  133 0085: 0000
  134 0086: 0000
  135 0087: 0000
```

.b

```
  136 0088: 008B  BLOCK_BEGIN     17,  3
  137 0089: 0004  BLOCK_HANDLER     0,  4
  138 008A: 0001  END_LOCALS    1

;;; subprogram LOCAL_PROCEDURE
;;; IO.PUT_LINE (S & INTEGER'IMAGE (LOCAL_VALUE));
;;; push parameters
  139 008B: E5FF  LOAD    2, -1     ;;; S
  140 008C: E20E  LOAD    1, 14     ;;; LOCAL_VALUE
  141 008D: 00E0  LOAD_ENCACHED    0     ;;; Standard
  142 008E: 1818  EXECUTE    PACKAGE_CLASS, FIELD_EXECUTE_OP, 24
  143 008F: 01CC  EXECUTE    VECTOR_CLASS, CATENATE_OP
  144 0090: E003  LOAD    0, 3     ;;; IO
  145 0091: 1864  EXECUTE    PACKAGE_CLASS, FIELD_EXECUTE_OP, 100     ;;;
                                                                        PUT_LINE
  146 0092: 4502  EXIT_SUBPROGRAM    2
  147 0093: 0000
  148 0094: 0000
  149 0095: 0000
  150 0096: 0000
  151 0097: 0000
```

```
  152 0098: 009C  BLOCK_BEGIN     19,  4
  153 0099: 0004  BLOCK_HANDLER     0,  4
  154 009A: 0002  END_LOCALS    2

;;; subprogram NESTED
;;; NESTED_LOCAL : INTEGER;
  155 009B: 00E2  LOAD_ENCACHED    2     ;;; INTEGER

;;; LOCAL_VARIABLE := PARM;
  156 009C: E7FF  LOAD    3, -1     ;;; PARM
  157 009D: A402  STORE_UNCHECKED    2, 2     ;;; LOCAL_VARIABLE

;;; NESTED_LOCAL := LOCAL_VARIABLE;
  158 009E: E402  LOAD    2, 2     ;;; LOCAL_VARIABLE
  159 009F: A602  STORE_UNCHECKED    3, 2     ;;; NESTED_LOCAL

;;; LOCAL_VALUE := PARM;
  160 00A0: E7FF  LOAD    3, -1     ;;; PARM
  161 00A1: A20E  STORE_UNCHECKED    1, 14     ;;; LOCAL_VALUE

;;; P2.all (3) := FALSE;
  162 00A2: 4800  SHORT_LITERAL    0     ;;; FALSE
  163 00A3: 4803  SHORT_LITERAL    3     ;;; 3
  164 00A4: E408  LOAD    2, 8     ;;; P2
  165 00A5: 0217  EXECUTE    HEAP_ACCESS_CLASS, ALL_REFERENCE_OP     ;;; P2.all
  166 00A6: 01D6  EXECUTE    VECTOR_CLASS, FIELD_WRITE_OP     ;;; P2.all (3)

;;; null;
  167 00A7: 4502  EXIT_SUBPROGRAM    2
```

#4b

```
;;; String Pointer table
  168 00A8: 0006  0012  0012

;;; String Literal Pool
  171 00AB: 7468  6973  2069  7320  6120  7465  7374  6869  |this is a testhi|
  179 00B3: 0000
  180 00B4: 0000
  181 00B5: 0000
  182 00B6: 0000
  183 00B7: 0000
```

```
;;; debug table start
  184 00B8: 0004  0000  5397  0000  0001  0007  000B  0000
  192 00C0: 8000  0000  001B  00DA  8001  0000  0033  00E0
  200 00C8: 0000  0000  0036  00E3  0000  0000  0043  00E8
  208 00D0: 0001  0000  008B  00F6  0001  0000  009B  00F9
  216 00D8: 0001  0000  0002  0002  001B  0021  0029  002D
  224 00E0: 0001  0000  0033  0003  0000  0036  0038  003A
  232 00E8: 000A  0002  0043  0044  004A  004C  0050  0053
  240 00F0: 0054  0063  0076  0077  0079  007E  0000  0001
  248 00F8: 008B  0001  0005  009B  009C  009E  00A0  00A2
  256 0100: 00A7
```

```
R1000 Ada Debugger (Version 9.1.6 - 7/11/87)
Beginning to debug: !USERS.PHIL.TEST_AREA % !USERS.PHIL.TEST_AREA.TASKY
Stop at: .command_procedure,  Root task: [Task : ROOT_TASK, #794DD].

Enable (ADDRESSES, TRUE);
The ADDRESSES flag has been set to TRUE.

Break (".TASKY.JUNK.CHUCK.1S", 1, "all", "TRUE");
The breakpoint has been created and activated:
Active Permanent Break 1 at .TASKY.JUNK.CHUCK.1S [any task]

Execute ("all");
Break  1:  .TASKY.JUNK.CHUCK.1s  [Task : #7C8DD].
Pc = #21D03, #9B.
Break  1:  .TASKY.JUNK.CHUCK.1s  [Task : #7C4DD].
Pc = #21D03, #9B.
Break  1:  .TASKY.JUNK.CHUCK.1s  [Task : #7CCDD].
Pc = #21D03, #9B.

Stack ("%#7CCDD", 0, 0);
Stack of task #7CCDD:
_1: CHUCK.1s
      Pc = #21D03, #9B, Frame = #7CCDD, #18, Lex = 2, Outer = #7C0DD, #0
_2: BUSY2.2s
      Pc = #21D03, #8D, Frame = #7CCDD, #12, Lex = 2, Outer = #7C0DD, #0
_3: BUSY.6s
      Pc = #21D03, #B9, Frame = #7CCDD, #0, Lex = 1, Outer = #7CCDD, #0
```

- lower mark word

```
Information (Space, "all");

Job:  221, Root task: #794DD
  Space use for task #7C4DD:
  Current control space:  1632,   Max control space:  3056   (bytes).
  Current data space:  514,   Max data space:  2047   (bytes).
  Address space sizes (in bits):
segment       current      max
Control        13056     24448
Data            4112     16383
Type               0         0
Queue                        0

  Space use for task ROOT_TASK, #794DD:
  Current control space:  752,   Max control space:  8160   (bytes).
  Current data space:  24,   Max data space:  2138111   (bytes).
  Address space sizes (in bits):
segment       current      max
Control         3328     57216
Data             193  17104895
Type            2688      8064
Queue                        0

  Space use for task #7C8DD:
  Current control space:  1632,   Max control space:  3056   (bytes).
  Current data space:  514,   Max data space:  2047   (bytes).
  Address space sizes (in bits):
segment       current      max
Control        13056     24448
Data            4112     16383
Type               0         0
Queue                        0
```

```
  Space use for task #7CCDD:
  Current control space:  1632,   Max control space:  3056   (bytes).
  Current data space:  514,   Max data space:  2047   (bytes).
  Address space sizes (in bits):
segment       current      max
Control        13056     24448
Data            4112     16383
Type               0         0
Queue                        0


Task_Display ("all", ALL_TASKS);

Job:  221, Root task: #794DD
#7C4DD (.TASKY.BUSY): At a breakpoint at .TASKY.JUNK.CHUCK.1s [Pri =  1,
    Memory cond = 0]
    Stopped in debugger.
ROOT_TASK, #794DD: Running, waiting for delay [Pri =  1, Memory cond = 0]
#7C8DD (.TASKY.BUSY): At a breakpoint at .TASKY.JUNK.CHUCK.1s [Pri =  1,
    Memory cond = 0]
    Stopped in debugger.
#7CCDD (.TASKY.BUSY): At a breakpoint at .TASKY.JUNK.CHUCK.1s [Pri =  1,
    Memory cond = 0]
    Stopped in debugger.
```

INSTRUCTION SET - REVIEW

- Code Segments
- Basic flavor of instructions
- Getting code segment listings for a unit
- Using the R1000 debugger to look at code segments

STORAGE HIERARCHY

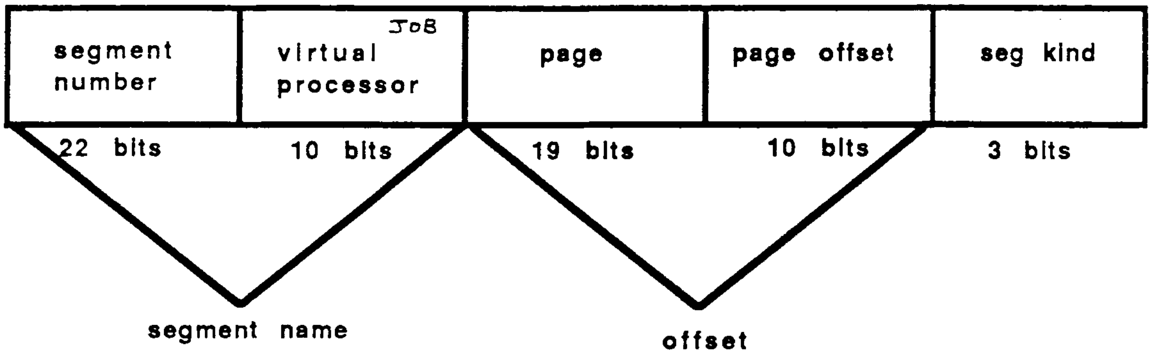
- Cache memory (RAM) and disk interact closely for ALL memory use by the R1000 CPU.

- No separation of program execution virtual space and file space.

- If the disks get full, you cannot execute instructions on the machine, and you are hosed.

- Full virtual address

page address



segment number	JOB virtual processor	page	page offset	seg kind
22 bits	10 bits	19 bits	10 bits	3 bits

segment name

offset

- The segment kind specifies which of control, type, data, import,code,queue, and system segments are to be referenced.

- The virtual processor is part of the segment name. More later.

- Segments are divided into 1 Kbyte pages.

STORAGE HIERARCHY - KERNEL

- The kernel subsystem manages segments on the disk.

- The microcode manages segments in the main memory (cache).

- Segments can be created, used, and deleted without ever reaching disk.

- Tasks cannot take page faults until the virtual memory system is up.

STORAGE HIERARCHY - SEGMENT NAMES

- Each module (package or task) has a name.
- This name is the segment name of the module's control, type and data stacks.
- The import space for a module has a different name.
- Code segments also have different names.
- Independent data objects (files and Ada units, for example) exist in their own data stacks. Each of these segments has a name.

EXECUTION ENVIRONMENT

- Each active thread of control (task or package during elaboration) has a control stack.

- The control stack contains a Task Control Block (TCB) and the procedure call stack.

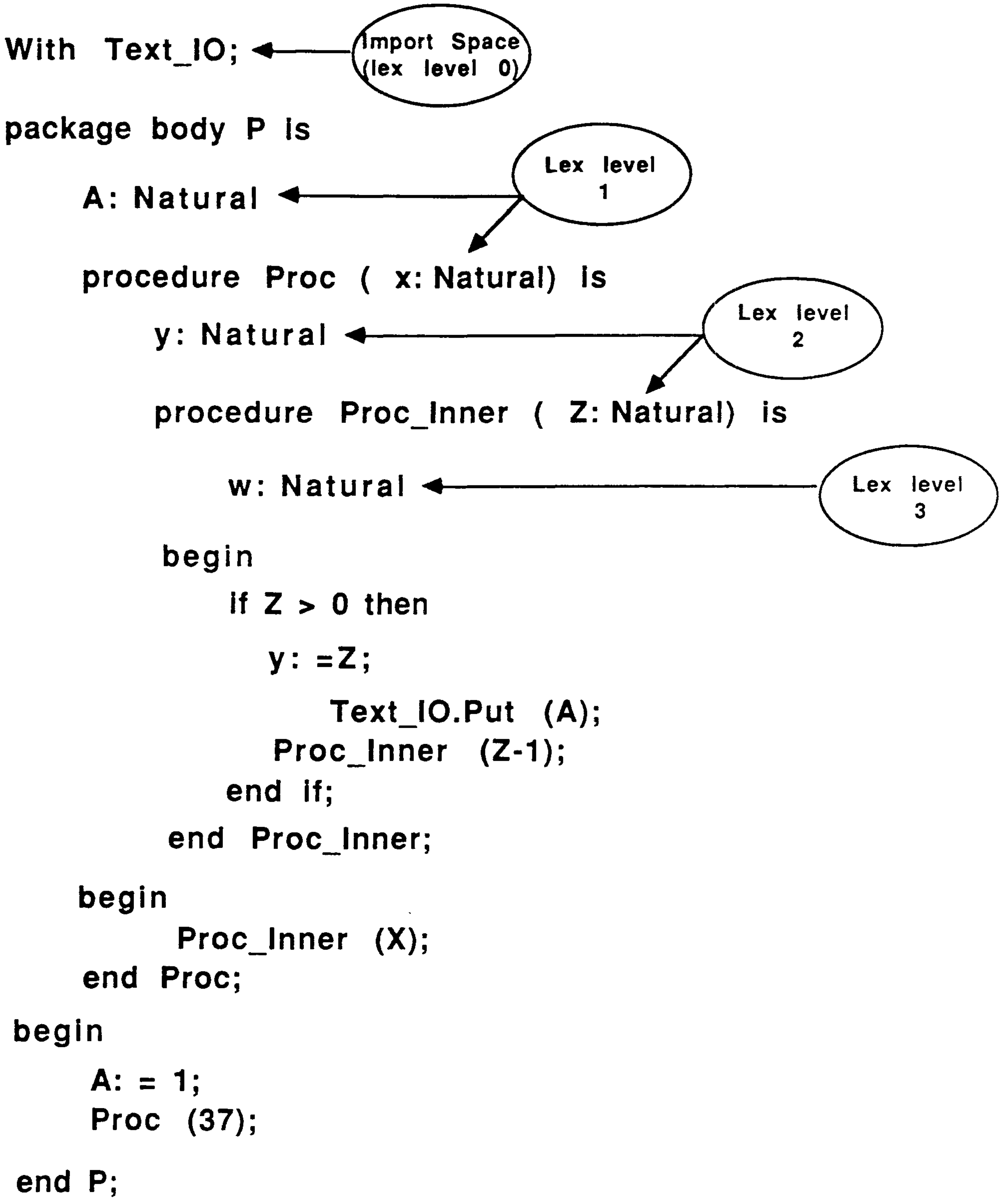
- The stack is divided into frames. Each frame contains the information for a particular procedure activation.

- The bottom frame represents the task or package outer block.

EXECUTION ENVIRONMENT - ADA ADDRESSING

```
With  Text_IO;

package  body  P  is

     A: Natural

     procedure  Proc  (  x: Natural)  is
          y: Natural

          procedure  Proc_Inner  (  Z: Natural)  is

               w: Natural

           begin
               if Z > 0 then
                  y: =Z;
                      Text_IO.Put  (A);
                  Proc_Inner  (Z-1);
               end  if;
           end  Proc_Inner;

     begin
           Proc_Inner  (X);
     end  Proc;
begin
     A: = 1;
     Proc  (37);

end  P;
```

EXECUTION ENVIRONMENT - ADA ADDRESSING



```
With  Text_IO;

package  body  P  is

    A: Natural

    procedure  Proc  (  x: Natural)  is

        y: Natural

        procedure  Proc_Inner  (  Z: Natural)  is

            w: Natural

         begin
            if Z > 0 then
              y: =Z;
                  Text_IO.Put  (A);
              Proc_Inner  (Z-1);
            end  if;
         end  Proc_Inner;

    begin
          Proc_Inner  (X);
    end  Proc;

begin
     A: = 1;
     Proc  (37);

end  P;
```

Import Space
(lex level 0)

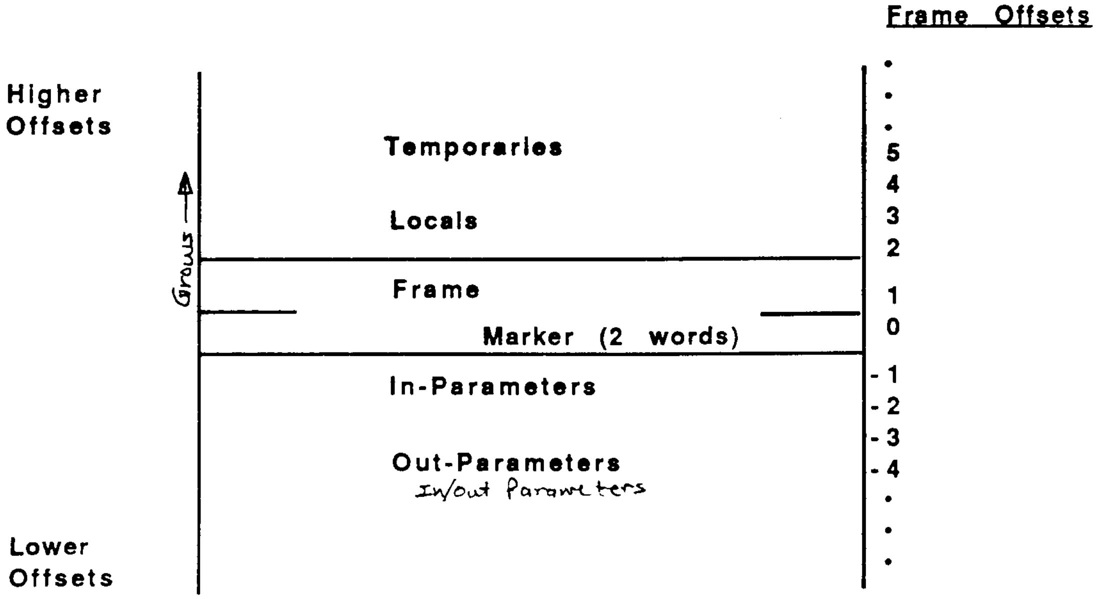
Lex level
1

Lex level
2

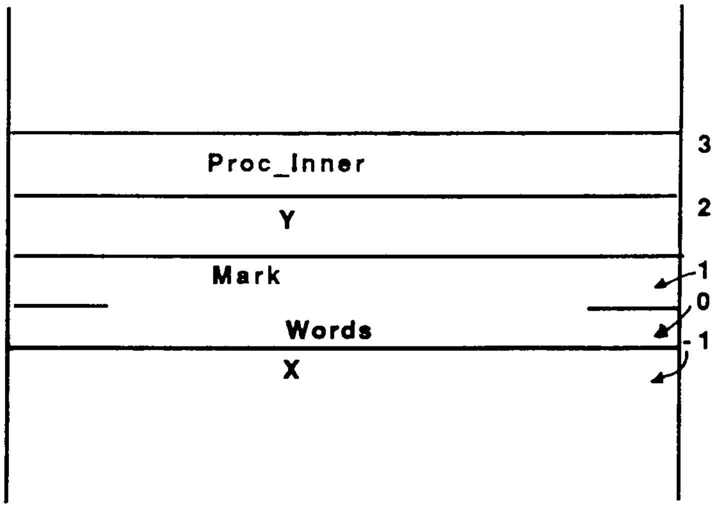
Lex level
3

EXECUTION ENVIRONMENT - ADA ADDRESSING (CONT'D.)

- Stack frame for a subprogram invocation.



- Stack frame for procedure Proc. Lex Level = 2



- Thus the call to Proc_Inner is:

  Load 2,-1 —— push X
  Call 2, 3 —— invoke Proc_Inner

EXECUTION ENVIRONMENT - ADA ADDRESSING (CONT'D.)

	LEX LEVEL	OFFSET	
P.A	1,	13	12 for TCB
P.Proc	1,	14	
Proc.X	2,	-1	
Proc.Y	2,	2	Ø&1 mark Words
Proc.Proc_Inner.Z	3,	-1	below mark words
Proc.Proc_Inner.W	3,	2	
Text_IO	0, (import space	1 (probably)	

R1ØØØ

① uses Display

② dynamic Links
    -once, then put into display

EXECUTION ENVIRONMENT - ADA ADDRESSING (CONT'D.)

- Lexical Level is the static level of nesting of an object from its containing package.

library subprograms get package to surround them (needs module)

- Machine addressing limits lex level to 4 bits - max procedure static nesting is 15 levels.

- Lex level 0 refers to import space.

- Each module is a closed scope. Can only address internal objects and objects imported to it by its parent.

EXECUTION ENVIRONMENT - FRAME INFORMATION

- Return address
- Data and type stack marks
- "Outer frame" pointer - name of enclosing package
- Lex level
- Flag
  - utility
  - rendezvous
- Children
  - tasks, packages

EXECUTION ENVIRONMENT - RETURNING VALUES

- Out parameters are on the stack

- Function return is a control word left on the top-of-stack after return

- Data and type stack are popped to the pre-call values when the subprogram returns except:

  - functions returning structured value: data stack not popped because return value is up there.

  - Implications: Returning from a subprogram that consumes a lot of data stack space may not free that space. Loops with such calls could be trouble spots causing Storage_Errors.

- The switch R1000_CG.Reclaim_Space cause the code generator to emit code in each loop to mark and pop the data stack to avoid run-away allocation. This switch defaults to true and is invisible unless set to false.

  doesn't appear in switch file
  – can't be edited until set false

- This applies only to loops.

EXECUTION ENVIRONMENT - RETURNING VALUES (CONT'D.)

- Thus, In

```
(A)       for i in 1..5 loop
                F (G(i));
          end loop;
```

and

```
(B)       F(G(1));
          F(G(2));
          F(G(3));
          F(G(4));
          F(G(5));
```

If G is a function returning a structure, (B) consumes 5 times the data stack space as (A).

EXECUTION ENVIRONMENT - DEBUGGER

- A number of debugger operations are available to display the execution environment:

Enable (Addresses)	causes display of machine information by a number of commands
Memory_Display	shows contents of the different kinds of memory segments
Stack	shows a module's call stack with frame addresses and information
Information (Space)	shows sizes of various memory segments
Task_Display	shows state of various tasks in the program

should follow Section

EXECUTION ENVIRONMENT - REVIEW

- Control stack frames
- Lex level, offset addressing
- Import spaces
- Function returns and data stack
- Debugger and kernel commands
- Package addressing

EXECUTION ENVIRONMENT - KERNEL COMMAND INTERPRETER

Show_Task_States	displays task names and state based on various queries
LMR	Logical Memory Read. Hex display of memory segments. Especially useful if debugger is not runnable.
LMW	Logical Memory Write
Jobs	show information about jobs. Covered more later.
Job	Same as Jobs, but for one job

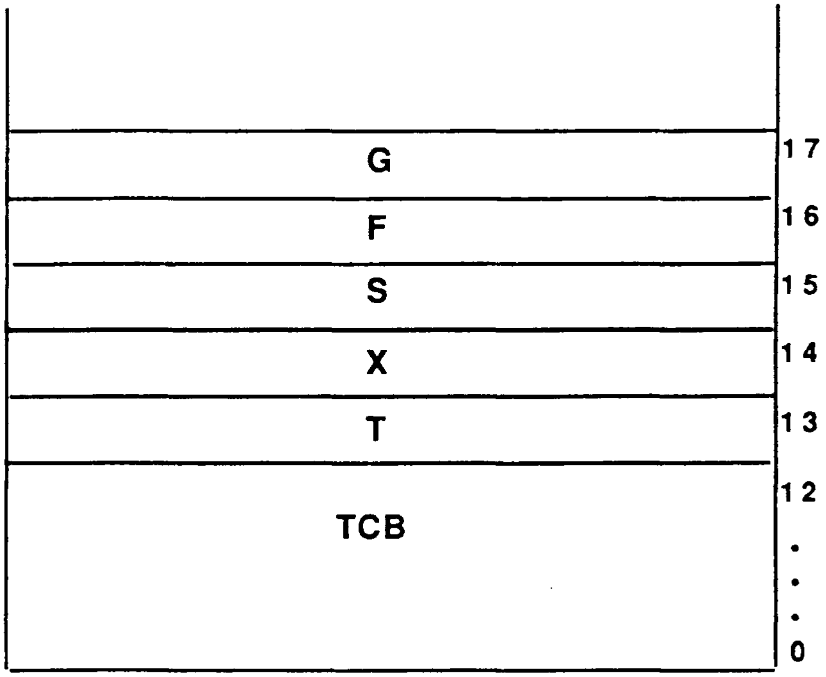
EXECUTION ENVIRONMENT - OTHER COMMANDS

Show_Tasks

displays stack dumps of specified task or jobs. More later.

EXECUTION ENVIRONMENT - PACKAGE ADDRESSING

```
package  P  is

 Type  T  is  private;

  x: Natural;

 Subtype  S  is  integer;

 function  F  returns  T;

 package  IO  renames  Text_IO;     - no offset for package renames

 function  G  renames  F;     - generates offset

private

  . . .

end  P;
```



EXECUTION ENVIRONMENT - PACKAGE ADDRESSING (CONT'D.)

to get P.X from outside P

```
LOAD       0,4                      --     suppose P is at import space
                                           offset 4

Execute    Package, Field_Read, 14        --X
```

Similarly to call P.F

```
LOAD       0,4

Execute    Package, Field_Execute, 16
```

- Each visible object has an offset in the module.
- Visible objects are marked specially as visible.
- Package renames occupy no runtime offset.
- Some constants also occupy no runtime offset.

Review

ADA FEATURES - EXCEPTIONS

- Exceptions are represented by a 48 bit number.

- An exception can be raised explicitly by loading the exception number and executing:

  Execute, Exception, Raise

- An exception can be raised implicitly when some Ada rule is violated. This is built into the R1000 instruction set. Checks are done in parallel with the operation.

- Exception propagation is handled largely by the microcode. The header of the subprogram points to the exception handler. If there is no handler, it points to location 4 in the code segment. — instruction there will propagate the exception ~~handler~~

- Once raised, an exception variable is pushed on the stack. This control word contains the exception number and raise PC.

- Functions are available in the environment to get an image of the exception variable

  Debug_Tools.Get_Exception_Name

  returns a string. Parameters control display of raise PC and machine representation of the exception number.

ADA FEATURES - EXCEPTIONS (CONT'D)

- Some values of the exception number are pre-defined. Others are encodings of the Diana pointer to the exception declaration.

- Some values are grouped into a contiguous range that represents different flavors of the same exception. This matters only in the image function which displays the flavor:

Constraint_Error	(type range)
Constraint_Error	(null access)

  etc.

- See handout for ranges.

```
 0           =>  "Unknown (0)";
 1           =>  "Constraint_Error (Type Range)";
 2           =>  "Constraint_Error (Case Range)";
 3           =>  "Constraint_Error (Exponent)";
 4           =>  "Constraint_Error (Null Access)";
 5           =>  "Constraint_Error (Array Index)";
 6           =>  "Constraint_Error (Length_Error)";
 7           =>  "Constraint_Error (Discriminant)";
 8           =>  "Constraint_Error (Variant)";
 9           =>  "Constraint_Error (Entry Family)";
 10 .. 31    =>  "Constraint_Error";
 32          =>  "Numeric_Error (zero divide)";
 33          =>  "Numeric_Error (overflow)";
 34 .. 47    =>  "Numeric_Error";
 48          =>  "Program_Error (elaboration order)";
 49          =>  "Program_Error (function exit)";
 50          =>  "Program_Error (select)";
 51          =>  "Program_Error (prompt executed)";
 52 .. 63    =>  "Program_Error";
 64          =>  "Storage_Error (control)";
 65          =>  "Storage_Error (type)";
 66          =>  "Storage_Error (data)";
 67          =>  "Storage_Error (import)";
 68          =>  "Storage_Error (queue)";
 69          =>  "Storage_Error (program)";
 70          =>  "Storage_Error (allocation)";
 71          =>  "Storage_Error (job page limit)";
 72          =>  "Storage_Error (oversize object)";
 73          =>  "Storage_Error (name)";
 74 .. 79    =>  "Storage_Error";
 80          =>  "Tasking_Error (activation)";
 81          =>  "Tasking_Error (completed task)";
 82          =>  "Tasking_Error (abnormal task)";
 83 .. 95    =>  "Tasking_Error";
 96          =>  "!Lrm.System.Operand_Class_Error";
 97          =>  "!Lrm.System.Type_Error";
 98          =>  "!Lrm.System.Visibility_Error";
 99          =>  "!Lrm.System.Capability_Error";
 100         =>  "!Lrm.System.Machine_Restriction";
 101         =>  "!Lrm.System.Illegal_Instruction";
 102         =>  "!Lrm.System.Illegal_Reference";
 103         =>  "!Lrm.System.Illegal_Frame_Exit";
 104         =>  "!Lrm.System.Record_Field_Error";
 105         =>  "!Lrm.System.Utility_Error";
 106         =>  "!Lrm.System.Unsupported_Feature";
 107         =>  "!Lrm.System.Illegal_Heap_Access";
 108         =>  "!Lrm.System.Select_Use_Error";
 109 .. 127  =>  "Instruction_Error";
 128         =>  "Task_Aborted";
 129         =>  "!Lrm.System.Frame_Establish_Error";
 130         =>  "Unimplemeneted_Microcode_Error";
 131         =>  "!Lrm.System.Nonexistent_Space_Error";
 132         =>  "!Lrm.System.Nonexistent_Page_Error";
 133         =>  "!Lrm.System.Write_To_Read_Only_Page";
 134         =>  "!Lrm.System.Heap_Pointer_Copy_Error";
 135         =>  "!Lrm.System.Assertion_Error";
 136         =>  "!Lrm.System.Microcode_Assist_Error";
 137 .. 223  =>  not used
 224         =>  "Constraint_Error";
 225         =>  "Numeric_Error";
```

```
 226         =>  "Program_Error";
 227         =>  "Storage_Error";
 228         =>  "Tasking_Error";
 229 .. 255  =>  not used
 256         =>  "!Io.Io_Exceptions.Status_Error";
 257         =>  "!Io.Io_Exceptions.Status_Error (already open)";
 258         =>  "!Io.Io_Exceptions.Status_Error (not open)";
 259 .. 271  =>  "Io_Exceptions.Status_Error";
 272         =>  "!Io.Io_Exceptions.Mode_Error";
 273         =>  "!Io.Io_Exceptions.Mode_Error (illegal on infile)";
 274         =>  "!Io.Io_Exceptions.Mode_Error (illegal on outfile)";
 275 .. 287  =>  "!Io.Io_Exceptions.Mode_Error";
 288         =>  "!Io.Io_Exceptions.Name_Error";
 289         =>  "!Io.Io_Exceptions.Name_Error (illformed name)";
 290         =>  "!Io.Io_Exceptions.Name_Error (nonexistent directory)";
 291         =>  "!Io.Io_Exceptions.Name_Error (nonexistent object)";
 292         =>  "!Io.Io_Exceptions.Name_Error (nonexistent version)";
 293         =>  "!Io.Io_Exceptions.Name_Error (ambiguous name)";
 294 .. 303  =>  "!Io.Io_Exceptions.Name_Error";
 304         =>  "!Io.Io_Exceptions.Use_Error";
 305         =>  "!Io.Io_Exceptions.Use_Error (access restriction)";
 306         =>  "!Io.Io_Exceptions.Use_Error (not checked out)";
 307         =>  "!Io.Io_Exceptions.Use_Error (wrong class)";
 308         =>  "!Io.Io_Exceptions.Use_Error (object frozen)";
 309         =>  "!Io.Io_Exceptions.Use_Error (lock_error)";
 310         =>  "!Io.Io_Exceptions.Use_Error (file/device full)";
 311         =>  "!Io.Io_Exceptions.Use_Error (line/page length exceeded)";
 312         =>  "!Io.Io_Exceptions.Use_Error (reset failed)";
 313         =>  "!Io.Io_Exceptions.Use_Error (operation unsupported)";
 314 .. 319  =>  "!Io.Io_Exceptions.Use_Error";
 320         =>  "!Io.Io_Exceptions.Device_Error";
 321         =>  "!Io.Io_Exceptions.Device_Error (nonexistent page)";
 322         =>  "!Io.Io_Exceptions.Device_Error (write to read-only page)";
 323         =>  "!Io.Io_Exceptions.Device_Error (illegal reference)";
 324         =>  "!Io.Io_Exceptions.Device_Error (illegal heap access)";
 325         =>  "!Io.Io_Exceptions.Device_Error (device data error)";
 326 .. 335  =>  "!Io.Io_Exceptions.Use_Error";
 336 .. 351  =>  "!Io.Io_Exceptions.End_Error";
 352         =>  "!Io.Io_Exceptions.Data_Error";
 353         =>  "!Io.Io_Exceptions.Data_Error (bad input syntax)";
 354         =>  "!Io.Io_Exceptions.Data_Error (bad input value)";
 355         =>  "!Io.Io_Exceptions.Data_Error (bad output value)";
 356         =>  "!Io.Io_Exceptions.Data_Error (bad output type)";
 357 .. 367  =>  "!Io.Io_Exceptions.Data_Error";
 368         =>  "!Io.Io_Exceptions.Layout_Error";
 369         =>  "!Io.Io_Exceptions.Layout_Error (bad column)";
 370         =>  "!Io.Io_Exceptions.Layout_Error (bad position)";
 371         =>  "!Io.Io_Exceptions.Layout_Error (bad item length)";
 372 .. 383  =>  "!Io.Io_Exceptions.Layout_Error";
```

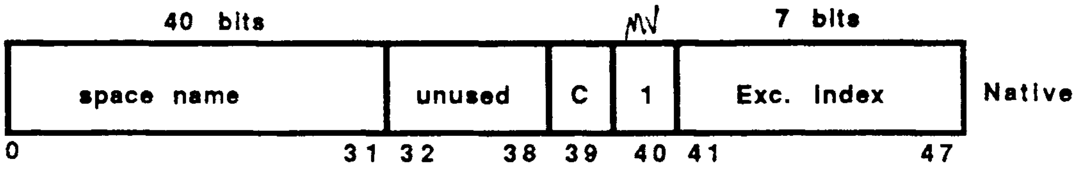
Handout #5c

```
package Debug_Tools is

    procedure Debug_On;
    procedure Debug_Off;

    -- Enable or disable debugging for the calling task's job.  When enabled,
    -- only tasks that are descendents of the caller can be debugged.
    -- When debugging is disabled, the task is released to execute, and all
    -- active debugger "hooks" are deactivated (eg, breakpoints, etc).

    function Debugging return Boolean;

    -- return true if calling task is being debugged.

    procedure Message (Info : String);

    -- Print the message string in the debugger window.  No operation if
    -- the debugger is not activated

    procedure User_Break (Info : String);

    -- "Break" in the debugger.  The calling task stops as though it
    -- encountered a breakpoint.  If the debugger is not active, no action
    -- is performed.  Otherwise, the task remains stopped until the
    -- debugger user explicitly continues its execution.

    procedure Set_Task_Name (Name : String);
    function Get_Task_Name return String;

    -- Set or retrieve a string "synonym" for the calling task.  This name
    -- is used within the debugger to make identifying task easier.
    -- It is also useful for multiple instances of the same task type
    -- to distinguish themselves in the debugger.
    -- No operation if the debugger is not activate.


    function Ada_Location (Frame_Number : Natural := 0;
                           Fully_Qualify : Boolean := True;
                           Machine_Info : Boolean := False) return String;

    -- Return a string name for the Ada location of execution in the
    -- specified stack frame.  Frame_Number = 0 refers to the caller
    -- of Ada_Location.  Frame_Number = 1 refers to its caller, and so
    -- on.  The null string is returned if the frame is nonexistent or
    -- its location cannot be found for some other serious reason.

    -- This procedure works independent of whether there is an active
    -- debugger for the calling tasks, but it may return less information
    -- if there is not.

    function Get_Exception_Name (Fully_Qualify : Boolean := True;
                                 Machine_Info : Boolean := False) return String;

    -- return a string representation of the exception most recently
    -- executed by the calling task.  Get_Exception_Name must be called
    -- either directly or indirectly from an exception handler.  If
    -- no exception is found, the null string is returned.


    function Get_Raise_Location (Fully_Qualify : Boolean := True;
```

```
                                 Machine_Info : Boolean := False) return String;

    -- return a string representation of the location of the exception
    -- most recently executed by the calling task.  Get_Raise_Location
    -- must be called either directly or indirectly from an exception
    -- handler.  If  no exception is found or other problems encountered,
    -- the null string is returned.


    generic
        type T is limited private;
        -- Type that will be displayed using Image procedure as part
        -- of debugger's object display procedure.

        with function Image (Value : T;
                             Level : Natural;
                             Prefix : String;
                             Expand_Pointers : Boolean) return String;

        -- Given the Value, return its image.  Level specifies the number
        -- of levels to detail to be displayed.  Expand_Pointers indicates
        -- that internal pointers should be expanded in the image.
        -- Level = 0 => entire object should be elided.

        -- If any ASCII.LF's are emitted, the following line should
        -- start with Prefix as an "indent" value.

    procedure Register;
    -- Make this special display procedure known to the debugger
    -- of the session making the call.


    generic
        type T is limited private;
    procedure Un_Register;
    -- Remove the display procedure from the calling session's
    -- database of special types for the type T given.

    pragma Subsystem (Native_Debugger);
    pragma Module_Name (4, 3802);

end Debug_Tools;
```

ADA FEATURES - EXCEPTIONS (CONT'D.)

- Exception numbers representing native exceptions are interpreted differently from those representing exception in cross-compiled (MV) code.

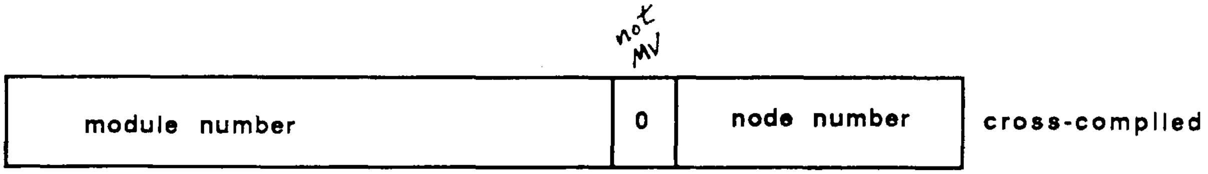


40 bits | MV | 7 bits

space name	unused	C	1	Exc. Index	Native
0 – 31	32 – 38	39	40	41 – 47	

R1000 Native <Space = 19230979, Index = 2> - includes bit at 40?

MV <Module = 3174, Ord = 46>



not MV

module number	0	node number	cross-compiled

C - Is_Code_Archived bit

ADA FEATURES - EXCEPTIONS

- Getting exception names

  - Cross-compiled exceptions: exceptions that are defined in the universe are built into a special table and the names of these are, thus, known. When the name of another cross-compiled exception is requested, the MV-cross debugger is called if one is connnected. If not connected, or the answer takes more than 20 seconds, the numeric form

    <module = n, ord = n >

    is returned.

  - Native code exceptions: since the Diana pointer is encoded into the exception number, the tree can be accessed to get the exception name if the tree exists. Cases where it doesn't:

    * loaded main programs
    * subsystem code views
    * exceptions defined in command windows

ADA FEATURES - EXCEPTIONS

- Getting exception information from the debugger

  Information (Exceptions)

- If addresses is enabled, shows control offset of exception variable. Use Memory_Display to look at this - it give the true information about the exception.

- When an exception is raised in the environment, a loaded main subprogram, or code view, the debugger provides the view that the exception was actually raised at the statement in Native code that called whoever actually raised the exception.

- The debugger keeps this information in a cache that can be displayed with:

  Debug.Show (Exception_Cache)

- In obscure cases, the cache information can be lost and the debugger will be vague about the raise location.

- If putting in a problem report about an environment exception flying out, it is helpful to include the true raise address if the problem is not easily reproducible.

ADA FEATURES - UNCHECKED CONVERSION

- Ada generic function (of unchecked conversion)
  - checks type for legality on every call. This makes it slow.

- Package Unchecked_Conversions
  - package Unchecked_Conversion_Package checks type for legality when instantiation is elaborated. Faster, but requires a module (several pages) for each instantiation.
  - conversions to/from byte string; good performance and very useful

Basic Rule

- The bits you put in are the bits you get out.

- See handout

Hc . #6

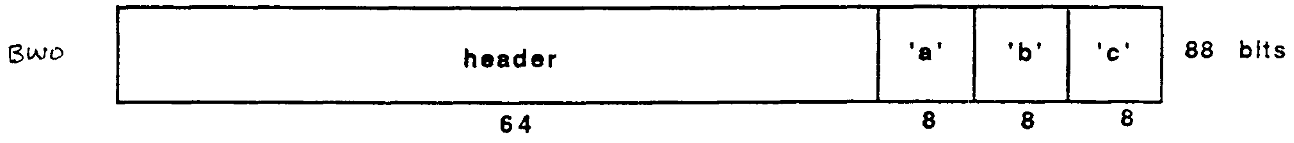
```
with System;

package Unchecked_Conversions is


    generic
        type Source is limited private;
        type Target is limited private;
    package Unchecked_Conversion_Package is

        function Convert (S : Source) return Target;
        -- Package form of LRM Unchecked_Conversion.
        -- Type-specific calculations are made during package elaboration, making
        -- calls to this convert faster than to an equivalent Unchecked_Conversi\
on
        -- instantiation.  Speed improvement depends on the type involved.
    end Unchecked_Conversion_Package;


    generic
        type Source is limited private;
    function Convert_To_Byte_String (S : Source) return System.Byte_String;
    -- Convert from Source to a byte string.  The byte string may contain
    -- more bits than the object.

    generic
        type Target is limited private;
    function Convert_From_Byte_String (S : System.Byte_String) return Target;
    -- Convert from a byte string to a Target type.  The string should
    -- have been produced by an instantiation of Convert_To_Byte_String
    -- with the same type.  A constrained object is always returned.


    pragma Subsystem (Miscellaneous);
    pragma Module_Name (4, 3543);
end Unchecked_Conversions;
```

ADA FEATURES - UNCHECKED CONVERSION (CONT'D.)

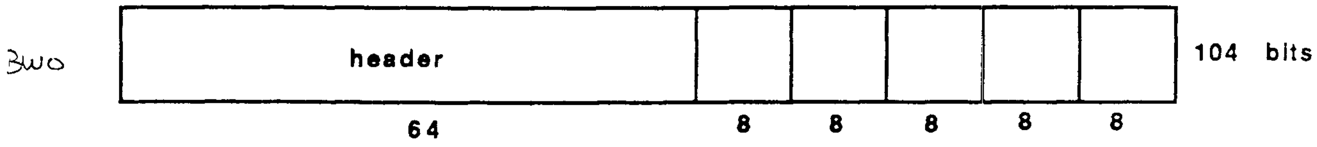
BWO Bounds with Object
BWT Bounds with Type

- Some examples of object layout :

String constant: "abc"

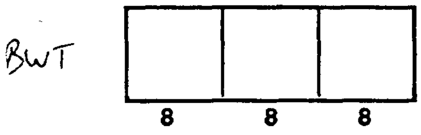


S: string (1..5)



Subtype Fixed_String is string (2..4);

S2: Fixed_String



24 bits - no header

```
with Display;  -- a simple generic that dumps an object in hex.
with System, Io;
with Unchecked_Conversion;  -- the slow unchecked conversion

procedure Array_Example is

    type Block is array (0 .. 7) of System.Byte;  -- This is the dest type
    Test_Block : Block;

    S : String (1 .. 5) := "abcde";
    subtype Fixed_String is String (2 .. 4);
    S2 : Fixed_String := "123";


    generic
        type E is private;
        type A is array (Positive range <>) of E;
    function Convert (Source : A) return Block;
    -- This generic converts "any" array type into a block.  Raises
    -- nasty exceptions of the data doesn't fit.  Only actual data is
    -- stored, not descriptor bits.

    function Convert (Source : A) return Block is
        Size : Natural := Source'Length * E'Size;  -- Number of data bits

        type Bit_Vector is array (Integer range <>) of Boolean;
        subtype Bit_Vect is Bit_Vector (1 .. 1024); -- bit image of Source
        subtype Block_Bits is Bit_Vector (1 .. Block'Size);  -- of Block

        Bits : Bit_Vect;          -- Gets binary image of source object
        Real_Bits : Block_Bits;   -- Gets binary image of value to return
        Data_Offset : Natural;    -- Offset in Bits of start of real data

        function C1 is new Unchecked_Conversion (A, Bit_Vect);
        function C2 is new Unchecked_Conversion (Block_Bits, Block);
    begin
        Io.Put_Line ("source size     = " & Natural'Image (Size));
        Io.Put_Line ("source obj size = " & Natural'Image (Source'Size));

        -- Form binary image of source parameter
        Bits := C1 (Source);

        -- Extract real data bits
        Data_Offset := Source'Size - Size + 1;
        Real_Bits (1 .. Size) := Bits (Data_Offset .. Source'Size);

        -- return final value
        return C2 (Real_Bits);
    end Convert;

    function Cvt is new Convert (Character, String);
    procedure D is new Display (Block);
begin
    Io.Put_Line ("String constant");
    Test_Block := Cvt ("abc");
    D (Test_Block);
    Io.New_Line;

    Io.Put_Line ("String variable");
    Test_Block := Cvt (S);
```

```
    D (Test_Block);
    Io.New_Line;

    Io.Put_Line ("String expression");
    Test_Block := Cvt ("A" & S & "A");
    D (Test_Block);
    Io.New_Line;

    Io.Put_Line ("Constrained string variable");
    Test_Block := Cvt (S2);
    D (Test_Block);
end Array_Example;
```

Output from this program:

```
String constant
source size     =  24
source obj size =  88
Object size =  64 bits.
Object fits in  8 bytes.
616263AB0829AA09
valid
String variable
source size     =  40
source obj size =  104
Object size =  64 bits.
Object fits in  8 bytes.
6162636465090128

String expression
source size     =  56
source obj size =  120
Object size =  64 bits.
Object fits in  8 bytes.
4161626364654138

Constrained string variable
source size     =  24
source obj size =  24
Object size =  64 bits.
Object fits in  8 bytes.
31323307278F6374
```

Handout #6a

```
with Unchecked_Conversion;  -- Language-defined function from !LRM
with Unchecked_Conversions;  -- package form from !tools
with String_Utilities, Io, System;

procedure Conversion is

    type Vstring (Max_Length : Positive) is
        record
            Length : Positive;
            Contents : String (1 .. Max_Length);
        end record;

    subtype V20 is Vstring (20);  -- A constrained subtype.
    type Bits is array (0 .. 400) of Boolean;

    S : V20;  -- We will be converting between these two objects
    B : Bits;

    function Convert1 is new Unchecked_Conversion (V20, Bits);
    package Convert2 is new Unchecked_Conversions.Unchecked_Conversion_Package
                           (Bits, V20);

    generic
        type T is private;
    procedure Display (Object : T);
    -- This procedure produces a binary dump of the Object.
    procedure Display (Object : T) is
        function Convert_To_Bytes is
           new Unchecked_Conversions.Convert_To_Byte_String (Source => T);

        procedure Display_Bytes (Value : System.Byte_String) is
            Bytes_Output : Natural := 0;
        begin
            Io.Put_Line ("Object fits in " & Natural'Image (Value'Length) &
                         " bytes.");
            for I in Value'First .. Value'Last loop
                Io.Put (String_Utilities.Number_To_String (Integer (Value (I)),
                                                           Base => 16,
                                                           Width => 2,
                                                           Leading => '0'));
                Bytes_Output := Bytes_Output + 1;
                if Bytes_Output mod 32 = 0 then
                    Io.New_Line;
                elsif Bytes_Output mod 8 = 0 then
                    Io.Put (" ");
                end if;
            end loop;
        end Display_Bytes;
    begin
        Io.Put_Line ("Object size = " & Integer'Image (Object'Size) & " bits.");
        Display_Bytes (Convert_To_Bytes (Object));
        Io.New_Line;
    end Display;


    procedure Dump is new Display (V20);  -- Instantiate dump for each
    procedure Dump is new Display (Bits);  -- of the two types.

begin
    S.Length := 15;  -- Initialize S
```

```
    S.Contents (1 .. 15) := "000011112222abc";

    -- Display initial object
    Io.Put_Line ("initial VString object:");
    Dump (S);

    -- Convert and display result
    Io.Put_Line ("converted bits object");
    B := Convert1 (S);
    Dump (B);

    -- Now, modify the bits in
    Io.Put_Line ("Modified original object");
    B (256) := True;     -- This changes one of the characters
    S := Convert2.Convert (B);  -- Convert bits back into record
    Dump (S);
    Io.Put_Line (S.Contents (1 .. 15));

end Conversion;

Generated output:

initial VString object:
Object size =  351 bits.
Object fits in  44 bytes.
800000140000001E 00000080000001C0 0000000200000028 6060606062626262
64646464C2C4C600 00000000
converted bits object
Object size =  401 bits.
Object fits in  51 bytes.
800000140000001E 00000080000001C0 0000000200000028 6060606062626262
64646464C2C4C600 0000000100000028 000000
Modified original object
Object size =  351 bits.
Object fits in  44 bytes.
800000140000001E 00000080000001C0 0000000200000028 6060606062626262
E4646464C2C4C600 00000001
00001111r222abc
```

```
generic
    type T is private;
procedure Display (Object : T);
 - This procedure produces a binary dump of the Object.

with Unchecked_Conversions;
with Io;
with System;
with String_Utilities;

procedure Display (Object : T) is
    function Convert_To_Bytes is
       new Unchecked_Conversions.Convert_To_Byte_String (Source => T);

    procedure Display_Bytes (Value : System.Byte_String) is
        Bytes_Output : Natural := 0;
    begin
        Io.Put_Line ("Object fits in " &
                     Natural'Image (Value'Length) & " bytes.");
        for I in Value'First .. Value'Last loop
            Io.Put (String_Utilities.Number_To_String (Integer (Value (I)),
                                                       Base => 16,
                                                       Width => 2,
                                                       Leading => '0'));
            Bytes_Output := Bytes_Output + 1;
            if Bytes_Output mod 32 = 0 then
                Io.New_Line;
            elsif Bytes_Output mod 8 = 0 then
                Io.Put (" ");
            end if;
        end loop;
    end Display_Bytes;

begin
    Io.Put_Line ("Object size = " & Integer'Image (Object'Size) & " bits.");
    Display_Bytes (Convert_To_Bytes (Object));
    Io.New_Line;

end Display;
```

```
with System;
with Unchecked_Conversion;
with Text_Io;
with Display;

procedure Unch is
    Base_B : constant := 31;
    Base : constant Long_Integer := 2 ** Base_B;
    subtype Universal_Digit is Long_Integer range -(Base - 1) .. (Base - 1);
    type Vector is array (Positive range <>) of Universal_Digit;
    X : Vector (1 .. 1) := (1 => 6);
    subtype Int is Vector (1 .. X'Length);
    Y : Int := X;  -- make sure we have a real object of type Int

    subtype Str is System.Byte_String
                  (1 .. X'Length * Universal_Digit'Size / System.Byte'Size);

    function Xbytes is new Unchecked_Conversion (Int, Str);
    Result : constant System.Byte_String := Str'(Xbytes (Int'(X)));
    Result2 : constant System.Byte_String := Str'(Xbytes (Y));
    Result3 : constant System.Byte_String := Str'(Xbytes (X));


    procedure Dump is new Display (Int);
    procedure Dump is new Display (Str);

    procedure Show (Result : Str) is
    begin
        Text_Io.Put_Line ("result'first = " & Integer'Image (Result'First));
        Text_Io.Put_Line ("result'last = " & Integer'Image (Result'Last));
        for I in Result'Range loop
            Text_Io.Put (System.Byte'Image (Result (I)) & " ");
        end loop;
    end Show;
begin
    Dump (X);
    Dump (Y);
    Dump (Result);
    Show (Result);
    Dump (Result2);
    Show (Result2);
    Dump (Result3);
    Show (Result3);
end Unch;
```

Output from this program:

```
Object size =  96 bits.        -- Dump of X : Vector (1..1) := (1=>6)
Object fits in  12 bytes.
0000000100000001 00000006

Object size =  32 bits.         -- Dump of Y : Int := X;  -- constrained
Object fits in  4 bytes.
00000006

Object size =  32 bits.         -- Result : conversion of Int'(X)
```

```
Object fits in  4 bytes.
00000001                        -- ended up with array bounds bits!
result'first =  1
result'last =  4
 0  0  0  1

Object size =  32 bits.         -- Result2 : conversion of Y
Object fits in  4 bytes.
00000006                        -- correct answer
result'first =  1
result'last =  4
 0  0  0  6

Object size =  32 bits.         -- Result3 : conversion of X
Object fits in  4 bytes.
00000001                        -- gets array bounds bits
result'first =  1
result'last =  4
 0  0  0  1
```

UNCHECKED - DEALLOCATION

- Normally, calls to instantiations of Unchecked_ Deallocation do not reclaim storage.

- For certain access types deallocation can be enabled and Unchecked_Deallocation will actually reclaim storage.

- When deallocation is enabled, storage requirements in the collection are increased.
  - small amount on (a per page basis) for headers
  - twice the 'size of the access type per allocated object (used to maintain the free list).

- Enabling deallocation for an access type, enables deallocation for all of its subtypes and all types derived from it (since they all share the same collection).

see next page

ENABLING UNCHECKED_ DEALLOCATION

- Pragma Enable_Deallocation (Access_Type_Name)

  - Enables deallocation, if possible, only for the named access type. Wanrings are produced when the unit is coded if the compiler can determinte that deallocation is not possible for ths type.

  - This pragma may be applied to generic formal access types and the compiler will produce warnings for instantiations that provide actual access types for which deallocation is enabled.

- Library switch R1000_Cg.Enable_Deallocation:=True

  - Enables deallocaiton for all access types for which deallocation is possible. Deallocation may be disabled for specific access types using pragma Disable_ Deallocation (Access_Type_Name).

ALLOCATION STRATEGY

- First fit

- Items that are larger than required are split, if possible (remaining fragment must be big enough).

- Deallocation returns items to the head of the free list.

RESTRICTIONS ON UNCHECKED_ DEALLOCATION

- Designated type may not be a task or contain task components

- Designated type may not be access to task or contain access to task components

- Access type may not be a segmented heap pointer

- Designated type may not be a segmented heap pointer or contain segmented heap pointers

- The compiler cannot always detect these cases at compile time to produce warnings. To determine if Unchecked_Deallocation will actually reclaim storage, instantiate the following generic in !Tools:

```
Generic
      type Object is limited private;
      type Name is access object;
function Allows_Deallocation return Boolean;
```

PROGRAM STRUCTURE

- When a command is run or main program is built, information on the entire program is gathered.
  - the unit state of all units is checked
  - the code for all units is located
  - an elaboration order for library units is computed
  - specs in spec views are matched with bodies in load views using the current activity
  - an elaboration module is generated

- The elaboration module is like a package that contains all units in the program. It:

  1) Declares each package type
  2) Sets-up import spaces for the modules
  3) Creates instances for each package
  4) Elaborates packages in order
  5) Calls the command procedure

PROGRAM STRUCTURE - ELABORATION SEGMENT

- When a main subprogram (pragma main) is promoted from installed to coded, its elaboration segment is built.

- This will take a long time if the closure is large. Thus, one shouldn't put code likely to change in such a unit.

- There is no Ada source, or Diana tree for the elaboration segment, and a minimal debug table.

- Occasionally an exception will be raised in an elaboration segment. It is difficult to pinpoint this with the debugger because there is no source. One must take the PC and look at the code to see if it is an elaboration segment. The segment listing can be helpful here.

- A special attribute called "units in program" is attached to the elaboration module. It lists all the Ada units in the program closure. See !Compiler_Interface.Units_In_Program.

Hando 79

```
Code for segment  1035520


    0 0000: 000F
    1 0001: 5800
    2 0002: 0000
    3 0003: 0068
    4 0004: 0100
    5 0005: 00BB
    6 0006: 0048
    7 0007: 0000
```

```
    8 0008: 000E  BLOCK_BEGIN      1,  6
    9 0009: 0004  BLOCK_HANDLER      0,  4
   10 000A: 0001  END_LOCALS     1

;;; module Main
   11 000B: 029C  DECLARE_SUBPROGRAM    FOR_OUTER_CALL, IS_VISIBLE
   12 000C: 0013  EXTENSION      2,  3
   13 000D: 00BF  ACTION    ACCEPT_ACTIVATION
   14 000E: 00BC  ACTION    SIGNAL_ACTIVATED
   15 000F: 00BB  ACTION    SIGNAL_COMPLETION
```

#9

```
   0010: 0015  BLOCK_BEGIN     2,  5
   0011: 001A  BLOCK_HANDLER     3,  2
   0012: 0002  END_LOCALS    2

;;; subprogram ARCHITECTURE_EXAMPLE
   0013: 029F  DECLARE_SUBPROGRAM    FOR_CALL
   0014: 0023  EXTENSION     4,  3
   0015: 8402  CALL     2, 2
   0016: 7801  JUMP     1
   0017: 0003  EXTENSION    3
   0018: 006F  ACTION    BREAK_UNCONDITIONAL
   0019: 4501  EXIT_SUBPROGRAM    1
   001A: 7801  JUMP     1
   001B: 0003  EXTENSION    3
   001C: 006F  ACTION    BREAK_UNCONDITIONAL
   001D: 00D8  LOAD_TOP    0     ;;; ( 2, 3)
   001E: 0100  EXECUTE    EXCEPTION_CLASS, RAISE_OP
   001F: 0000
```

```
   0020: 004A  BLOCK_BEGIN     9,  2
   0021: 0004  BLOCK_HANDLER     0,  4
   0022: 0006  END_LOCALS    6

;;; subprogram ARCHITECTURE_EXAMPLE
   0023: 7801  JUMP    1
   0024: 0002  EXTENSION    2
   0025: 006F  ACTION    BREAK_UNCONDITIONAL
   0026: 6039  INDIRECT_LITERAL    DISCRETE_CLASS, 57     ;;; page limit -
                                                         actual value is patched
   0027: 6034  INDIRECT_LITERAL    DISCRETE_CLASS, 52     ;;; module name of
                                                              (Runtime_Utilities)
   0028: E002  LOAD    0, 2     ;;; Proto Package Type
   0029: E001  LOAD    0, 1     ;;; Ucode Assist Package
   002A: 1835  EXECUTE    PACKAGE_CLASS, FIELD_EXECUTE_OP, 53
   002B: 180D  EXECUTE    PACKAGE_CLASS, FIELD_EXECUTE_OP, 13
   002C: 4800  SHORT_LITERAL    0
   002D: 602A  INDIRECT_LITERAL    DISCRETE_CLASS, 42
   002E: 02A0  DECLARE_SUBPROGRAM    NULL_SUBPROGRAM
   002F: 038E  DECLARE_TYPE    PACKAGE_CLASS, DEFINED       ;;;
                                                              ARCHITECTURE_EXAMPLE
   0030: 0387  DECLARE_VARIABLE    PACKAGE_CLASS     ;;; ARCHITECTURE_EXAMPLE
   0031: 00D8  LOAD_TOP    0
   0032: 190D  EXECUTE    PACKAGE_CLASS, FIELD_REFERENCE_OP, 13     ;;; Local
                            copy of lib unit subprogram ARCHITECTURE_EXAMPLE
   0033: 6020  INDIRECT_LITERAL    DISCRETE_CLASS, 32     ;;; module name of
                                                                            IO
   0034: E002  LOAD    0, 2     ;;; Proto Package Type
   0035: E001  LOAD    0, 1     ;;; Ucode Assist Package
   0036: 1835  EXECUTE    PACKAGE_CLASS, FIELD_EXECUTE_OP, 53
   0037: 6018  INDIRECT_LITERAL    DISCRETE_CLASS, 24     ;;; module name of
                                                                   JOB_SEGMENT
   0038: E002  LOAD    0, 2     ;;; Proto Package Type
   0039: E001  LOAD    0, 1     ;;; Ucode Assist Package
   003A: 1835  EXECUTE    PACKAGE_CLASS, FIELD_EXECUTE_OP, 53
   003B: 6010  INDIRECT_LITERAL    DISCRETE_CLASS, 16     ;;; module name of
                                                                       DEFAULT
   003C: E002  LOAD    0, 2     ;;; Proto Package Type
   003D: E001  LOAD    0, 1     ;;; Ucode Assist Package
   003E: 1835  EXECUTE    PACKAGE_CLASS, FIELD_EXECUTE_OP, 53
   003F: 00D8  LOAD_TOP    0     ;;; ( 3, 6)     ;;; DEFAULT
   0040: E605  LOAD    3, 5     ;;; JOB_SEGMENT
   0041: E604  LOAD    3, 4     ;;; IO
   0042: 4803  SHORT_LITERAL    3
   0043: E602  LOAD    3, 2     ;;; ARCHITECTURE_EXAMPLE
   0044: 020E  EXECUTE    MODULE_CLASS, AUGMENT_IMPORTS_OP
   0045: E602  LOAD    3, 2     ;;; ARCHITECTURE_EXAMPLE
   0046: 020F  EXECUTE    MODULE_CLASS, ACTIVATE_OP
   0047: 7802  JUMP    2     ;;; code segment reference table start
   0048: 002D  EXTENSION     5,  5
   0049: 0000  EXTENSION    0
   004A: 8603  CALL    3, 3     ;;; ARCHITECTURE_EXAMPLE
   004B: 4501  EXIT_SUBPROGRAM    1
   004C: 0000  LITERAL_VALUE    DISCRETE_CLASS, 931844
   004D: 0000
   004E: 000E
   004F: 3804
   0050: 0000  LITERAL_VALUE    DISCRETE_CLASS, 935940
   0051: 0000
```

```
  82 0052: 000E
  83 0053: 4804
  84 0054: 0000  LITERAL_VALUE    DISCRETE_CLASS, 3590148
  85 0055: 0000
  86 0056: 0036
  87 0057: C804
  88 0058: 000F  LITERAL_VALUE    DISCRETE_CLASS, 1033472
  89 0059: C500
  90 005A: 0000
  91 005B: 00B0
  92 005C: 0000  LITERAL_VALUE    DISCRETE_CLASS, 1641476
  93 005D: 0000
  94 005E: 0019
  95 005F: 0C04
  96 0060: 0000  LITERAL_VALUE    DISCRETE_CLASS, 0
  97 0061: 0000
  98 0062: 0000
  99 0063: 0000
 100 0064: 0000
 101 0065: 0000
 102 0066: 0000
 103 0067: 0000
```

```
;;; debug table start
  104 0068: 0004  0000  5397  0000  0001  0001  000B  0000
  112 0070: 0009  0000
```

PROGRAM STRUCTURE - LIBRARY UNITS

- Each library package is a child of the elaboration module.

- Library unit subprograms are wrapped in packages. Thus, each library unit is a module.

- Consequently, many library subprograms occupy far more pages than one package with many subprograms. There may be closure size issues with this approach.

PROGRAM STRUCTURE - MAIN UNITS

- Main subprograms serve as roofs over entire programs. When a main subprogram is called, a new set of library units are elaborated.

```
package P is . . . end p

with P; procedure M is . . . end m; pragma main;
```

- A command:

```
M;
M;
M;
```

will elaborate 3 independent copies of package P, not one shared copy. If M were not a main subprogram, only one shared copy of P would be elaborated.

* beware of package state!

- In addition to the semantic implications, this also has debugger implications since there can be more than one elaborated copy of a package in a program.

- If any unit in the closure of a main unit is uncoded, the main unit will be demoted as well.

PROGRAM STRUCTURE - LOADED MAIN UNITS

- A loaded main unit is built by the Compilation.Load command and is a single object containing a full copy of the entire closure of the main unit.

- It does not include any trees or source and cannot be debugged. <unknown location>

- It is independent of the units from which it was built - they can be demoted or deleted independent of the loaded main unit.

PROGRAM STRUCTURE - PAGE LIMITS

intended to catch runaway jobs

- When a page limit pragma appears, the limit value for the job is set to that value if that value is bigger.

- Every unit has a limit associated with it. The value from the switch file is used if there is no explicit pragma.

- The largest limit is set <u>before</u> program elaboration begins.

- An explicit call in the program to set the limit will set the limit independent of any value imposed by pragmas or switches.

  System_Utilities.Set_Job_Page_Limit

PROGRAM STRUCTURE - PAGE LIMITS (CONT'D.)

EXAMPLE 1

Session Default: 8000

Library Default: 9000

Unit A: Pragma Page_Limit (7000)

Unit B: No Pragma

Unit C: Pragma Page_Limit (10000)

Result: Elaboration begins with limit 10000

EXAMPLE 2

Session Default: 8000

Library Default: 12000

Unit A: Pragma Page_Limit (7000)

Unit B: No Pragma

Unit C: Pragma Page_Limit(10000)

Result: Elaboration begins with limit 12000

EXAMPLE 3

Session Default: 8000

Library Default: 8000

Unit A: Pragma Page_Limit (9000)

Unit B: Main subprogram with its limit 12000

Result: Elaboration begins with limit 9000. Limit raised to 12000 when B is called. (N.B. - This currently does not work.)

Stays at 12000 when B returns

4-Aug-1987

PROGRAM STRUCTURE - REVIEW

- Elaboration modules
- Library units
- Setting-up import spaces
- Main unit semantics
- Loaded main subprograms
- Page limits

TASKING - BASICS

- Each task has a name. This is the segment name of the task's control, type, and data stacks.

- Recall that each package is a "task", too.

- Tasks have state and other information the system keeps about them.

- Task creation, activation, termination, rendezvous, etc., are handled by the R1000 microcode.

- Major steps in a task:
  - creation of the task type
  - creation of the task
  - activation of the task - elaboration of declarative part
    - * occurs when "begin" reached; Tasking_Error if exception during activation; or when allocated
  - execution of the task; interactions with other tasks; delays
  - completion of the task -- T' callable = false
  - termination of the task -- T' terminated = true
  - aborts make tasks "abnormal"

    at next synchronization point, task will terminate

TASKING - TASK CONTROL BLOCK

- The first 13 words of each control stack are the Task Control Block (TCB).

- This has lots of state information concerning the task.

- See handout for format.

- Many debugger and diagnostic commands extract information from the TCB

- Summary

Word	
0	Outer block mark words
1	" " " "
* 2	Current PC, TOS, priority
3	Data TOS, Type TOS
* 4	Microcode temp storage
* 5	" " " "
* 6	Task type, import space, and declarer
7	Task dependence information
8	Size of control stack, break mask
9	Size of type, queue, and data spaces
A	Delay, scheduler, and debugger information
B	Debugger interface subprogram
C	Microcode temp storage for slice information

* These tend to be the most interesting during problem diagnosis

Handout #10 - TCB format

MAR and TAGSTORE

```
0       8       16      24      32      40      48      56      64      72      80      88      96      104     112     120
+---------------------------------------------------------------------------------------------------------------------------+
| refresh       | refresh       |  flags  | flu    |              |     segment         | vpid  |    page       |word | bit |
|   interval    |     window    |         | length |              |                     |       |               |     |     |
|               |               |         | reg    |              |                     |       |               |     |     |
+---------------------------------------------------------------------------------------------------------------------------+
flags : 32 - scavenger trap     36 - fill mode         40 - incomplete
        33 - cs out of range    37 - physical last
        34 - page crossing      38 - write last
        35 - cache miss         39 - mar modified
                                                                                                       dirty
                                                                                                         V
                                                                +------------------------------------------------------------+
                                                                |     segment         | vpid  |    page       ||lru|p|f |b |
                                                                |                     |       |               ||   | |l |p |
                                                                |                     |       |               ||   |s|g |c |
                                                                +------------------------------------------------------------+
                                                                page state : 00 - invalid        flags : 58 - wired
                                                                             01 - r/w                    59 - permanent
                                                                             10 - r/o                    60 - writable
                                                                             11 - loading
```

Space kinds

1 - Control　　4 - Data　　7 - System
2 - Type　　5 - Import
3 - Queue　　6 - Code

SOME USEFUL TAGS

Discrete	00 (80)	Record	44 (C4)	Subprogram	06 (86)	Null Subprogram	76	Seg Heap	38 (B8)
Access	10 (90)	Variant Record	4C (CC)	(Elaborated)	16 (96)	Utility	68 (E8)		
Task	18 (98)	Vector	6C (EC)	(Visible)	26 (A6)	Accept "	46 (C6)		
Package	58 (D8)	Matrix	74 (F4)	(Visible & Elaborated)	36 (B6)	Interface "	56 (D6)		
Float	08 (88)	Array	7C (FC)			Exception Var	7E (FE)		

BLOCKED STATES

Unblocked	00	Terminable At End	07	In FS Rendezvous	0E	Blocking On Accept	18
Declaring Module	01	Blocking On Entry	08	In Wait Svc	0F	Blocking On Select	19
Awaiting Activation	02	Delaying On Entry	09	Delay In Wait Svc	10	Delaying On Select	1A
Activating Module	03	Attempting Entry	0A	Blocking On Abort	11	Await Children Select	1B
Activating Tasks	04	Delaying	0B	Deleted	12	Terminable In Select	1C
Awaiting Task Activ	05	Aborting Module	0C	Aborted While In MTS	13		
Awaiting Children	06	Terminated	0D	In_MTS_RendezVous	14		

offset

```
     0       8       16      24      32      40      48      56      64      72      80      88      96      104     112     120
     +---------------------------------------------------------------------------------------------------------------------------+
C    |                 slice stuff                           |      |                                                            |600
     |                                                       | (09) |                                                            |
     +---------------------------------------------------------------------------------------------------------------------------+
B    |                                                       |subprg|                                                            |580
     |            debug interface subprogram                 | var  |                                                            |
     |                                                       | (36) |                                                            |
     +---------------------------------------------------------------------------------------------------------------------------+

     +---------------------------------------------------------------------------------------------------------------------------+
A    |  delay days     |        delay ticks          |m|f|sched |    scheduling_group            | debugging  |                 |500
     |  (18 bits)      |        (36 bits)            |t|l|alloc |                                |   state    |                 |
     |                 |                             |s|g| (49) |                                |            |                 |
     +---------------------------------------------------------------------------------------------------------------------------+
9    |     breakpoint scope         |    | type extent   | aux  |    |     queue extent          |       data extent              |480
     |                              |    |               |alloc |    |                           |                                |
     |                              |    |               | (3f) |    |                           |                                |
     +---------------------------------------------------------------------------------------------------------------------------+
8    |    distributor's name        |flgs| control extent|contrl| breakpoint  |                                                 |400
     |                              |    |               |alloc |   mask      |                                                 |
     |                              |    |               | (41) |             |                                                 |
     +---------------------------------------------------------------------------------------------------------------------------+
7    |   dependence site name       |    |dependence site|depend|                                                              |380
     |                              |    |    offset     | link |                                                              |
     |                              |    |               | (19) |                                                              |
     +---------------------------------------------------------------------------------------------------------------------------+
6    |      our type name           |flgs|our type offset|static| our import stack name     |      declarer's name           |300
     |                              |    |               | conn |                           |                                |
     |                              |    |               | (11) |                           |                                |
     +---------------------------------------------------------------------------------------------------------------------------+
5    |                                                   |micro |                                                              |280
     |                                                   |state2|                                                              |
     |                                                   | (29) |                                                              |
     +---------------------------------------------------------------------------------------------------------------------------+
4    |                                                   |micro |                                                              |200
     |                                                   |state1|                                                              |
     |                                                   | (21) |                                                              |
     +---------------------------------------------------------------------------------------------------------------------------+
3    |   queue successor name       |    |   type tos    | aux  | inner frame  |            |          data tos                  |180
     |                              |    |               |state |              |            |                                    |
     |                              |    |               | (3f) |              |            |                                    |
     +---------------------------------------------------------------------------------------------------------------------------+
2    | current         |blkd |m|pri|flgs| control tos   |cntrl |                 current macro pc                    |lx|100
     | slice           |state|d|ori|    |               |state |                                                     |  |
     | time            |     | |ty |    |               | (01) |                                                     |  |
     +---------------------------------------------------------------------------------------------------------------------------+
1    |    outer frame name          |flgs|  type frame   |activ | control pred | block      |          data frame                |
     |                              |    |               |link  |              | start      |                                    |
     |                              |    |               | (3f) |              |            |                                    |
     +---------------------------------------------------------------------------------------------------------------------------+
0    |   enclosing frame name       |flgs|  enclosing    |activ |      |     return          |children |  return       |lx|000
     |                              |    |    frame      |state |      |     address         | start   |  address      |  |
     |                              |    |    offset     | (7f) |      |     segment         | offset  | offset/index  |  |
     +---------------------------------------------------------------------------------------------------------------------------+
     0       8       16      24      32      40      48      56      64      72      80      88      96      104     112     120
```

TASKING - STATES

- See handout (next page)

- Described in debugger manual

- In bad situations, a task may get "stuck" in an unusual state.

- Examples of this include:
  - unrecoverable disk error
  - microcode bug
  - I/O problems

- Diagnosis
  - Kernel Show_Task_States command. Normally shows "interesting" tasks.
  - Show_Tasks environment command. This is in the System_Maintenance subsystem.
  - Debugger Task_Display command if job is running under the debugger
  - more shortly

Handout # 12 – Task states

- Unblocked—“running”—The task is ready to execute or is currently executing. (Higher-priority tasks may prevent the task from running, but there is no condition associated with the task to keep it from running.)
- Declaring_Module—“waiting for child elaboration”—The module is in the process of declaring a task or package.
- Awaiting_Activation—“waiting for parent elaboration”—The module is waiting for a package or task it has declared to become activated.
- Activating_Module—“activating child packages”—The module is waiting for children packages to become elaborated.
- Activating_Tasks—“activating child tasks”—The module is telling its child tasks to become activated.
- Awaiting_Task_Activation—“waiting for task activation”—The module has told its child tasks to become activated and is waiting for confirmation that each has been activated.
- Awaiting_Children—“waiting for children”—The task is waiting for its children tasks to be terminated.
- Terminable_At_End—“package completed”—The task has reached its end and will terminate according to Ada rules when its parent decides it's time to terminate.
- Blocking_On_Entry—“waiting at entry for accept”—The task is blocked waiting for an entry call to be accepted.
- Delaying_On_Entry—“waiting at timed entry for accept”—The task is blocked at a timed entry call waiting for the called task to accept the call.
- Attempting_Entry—“attempting entry call”—Reserved for future use.
- Delaying—“waiting for delay”—The task is blocked at a delay statement.
- Aborting_Module—“begin aborted”—The module is in the process of aborting a task it has declared.
- Terminated—“terminated”—The task is terminated.
- In_FS_Rendezvous—“state presently unknown”—The task is processing a page fault. This transient state is unlikely to be seen.
- In_Wait_Service—“in wait service”—The task is blocked in an Environment service.
- Delaying_In_Wait_Service—“delaying in wait service”—The task is blocked in an Environment service and has a maximum wait time.
- Blocking_in_Abort—“being aborted”—The module is in the process of making another task abnormal (see *Ada Language Reference Manual*, Section 9.10).
- Blocking_On_Accept—“waiting at accept for entry call”—The task is blocked at an accept statement, waiting for an entry call.
- Blocking_On_Select—“waiting at select for entry call”—The task is blocked at a select statement, waiting for an entry call.
- Delaying_On_Select—“waiting at select-delay for entry call”—The task is blocked in a select with a delay alternative waiting for an entry call.
- Awaiting_Children_In_Select—“waiting at select-terminate for entry call”—The task is executing at a select, with a terminate alternative, and has dependent tasks.
- Terminable_In_Select—“waiting at select-terminate for entry call”—The task is blocked in a select, with a terminate alternative, waiting for an entry call and can terminate.

TASKING - KERNEL SHOW_TASK_ STATES

- Kernel CI command

- By default, shows names and states of "interesting" tasks

- Simple use: on IOA console, type ↑Z until the kernel prompt appears, then enter:

```
Show_Task_States
```

(example next page)

```
EEDB: kernel
Kernel: show_task_states
CACHE/DISK [CACHE]:
16#4388004#; IN_WAIT_SERVICE X25_WAIT; PRI 1
16#27805#; UNBLOCKED; PRI 15
16#488D9#; IN_WAIT_SERVICE TCP_IP_INPUT_WAIT; PRI 14
16#127CD804#; DELAYING_IN_WAIT_SERVICE U031; PRI 3
16#1245A804#; IN_WAIT_SERVICE TCP_IP_INPUT_WAIT; PRI 3
16#5B8D3#; UNBLOCKED; PRI 14
16#5B4D3#; UNBLOCKED; PRI 14
16#52CEE#; UNBLOCKED; PRI 8
16#5BCD3#; UNBLOCKED; PRI 14
16#5C4D7#; IN_WAIT_SERVICE TCP_IP_INPUT_WAIT; PRI 4
16#7C8DD#; DELAYING_IN_WAIT_SERVICE NATIVE_DEBUGGING_WAIT; PRI 8
16#7C4DD#; DELAYING_IN_WAIT_SERVICE NATIVE_DEBUGGING_WAIT; PRI 8
16#7CCDD#; DELAYING_IN_WAIT_SERVICE NATIVE_DEBUGGING_WAIT; PRI 8
Kernel: enable_priv_cmds
You are enabling a set of commands which must be used
with extreme care.  They should be used only by
knowledgeable support personnel.  These commands can
easily crash/hang the machine; some can competely trash
the state of the machine such that you must recover the
machine from backup tapes.
Proceed [FALSE]: yes
Password: secret
*Kernel: lmr
KIND [1]:
NAME [0]: 16#794dd@
EXPECTED VALUES ARE:  0 ..  4294967295
NAME [0]: 16#794dd#
OFFSET [0]:
COUNT [0]: 5
  not WIRED  not IN_TRANSIT  DIRTY  not PERMANENT  READ_WRITE
16#00#   16#000794DD0000007F# 16#0000000000180001#
16#01#   16#000794DD9800003F# 16#FFFF00200000000#
16#02#   16#FFB802D810000D01# 16#DD021D0300000422#
16#03#   16#0007C8DD00000ABF# 16#00014000000000C1#
16#04#   16#0000000000000021# 16#0000063E00000000#
*Kernel: job
VPID [4]: 16#dd#
 Job Pri Stat CPU%  ModCt Cache   Disk  PgLim DskWts D/S JSegSz WsSiz WsLim
---- --- ---- ---- ------ ----- ------ ------ ------ --- ------ ----- -----
 221   6 I,AT    0      8    63      0   8000     88   0     50    50    50
*Kernel: quit
EEDB:
```

TASKING - EXCEPTION PROPAGATION

- If an exception propagates out of a package that package terminates and the exception is re-raised at the point of declaration of the package. The raise PC is set to the point of declaration. This has debugging implications.

- If you did not cath the original raise in the debugger, it may be difficult to locate the source of the exception without re-reunning the program.

TASKING - WAIT SERVICE

- Wait service is a wait/signal mechanism used by the environment.

- It allows a task to block until another task signals (rousts) it.

- The wait service is used as part of normal operation to wait for I/O transfers and resource availability.

- The wait service is also used when a serious error is detected. Then, the affected task may be blocked in the wait service so someone can look at it and evaluate the situation.

- The kernel Show_Task_States command shows stopped in the waiter service and displays the reason they are stopped

  ```
  ENVIRONMENT_DEBUGGING_WAIT
  ```

  is the reason that appears for such fatal conditions.

- Tasks stopped in the wait service can be manually rousted by using a kernel command.

Next page

TASKING - WAIT SERVICE (CONT'D.)

- This is generally dangerous and should only be done if you know what you are doing or have advice from someone who does. Permanent, fatal system damage can result if a task is rousted at the wrong time, although this is rare

```
*Kernel:        ROUST

 Task id:       16#127FB04#
```

- Kernel always takes decimal numeric parameters. Used based notation (16#...#) for hex addresses and names.

TASKING - SHOW_TASKS COMMAND

- Displays stack of specific or group of tasks

- Useful to monitor progress or diagnose hangs

- Return dump of this in hung editor or session case

- When given job id, attempts to locate as many tasks as possible in that job

- !Commands.System_Maintenance.Current_Combined_ Mload provides source information for the environment itself. This is packages level information only.

- There are parameters to specify the type of stock display, and to filter the task search.

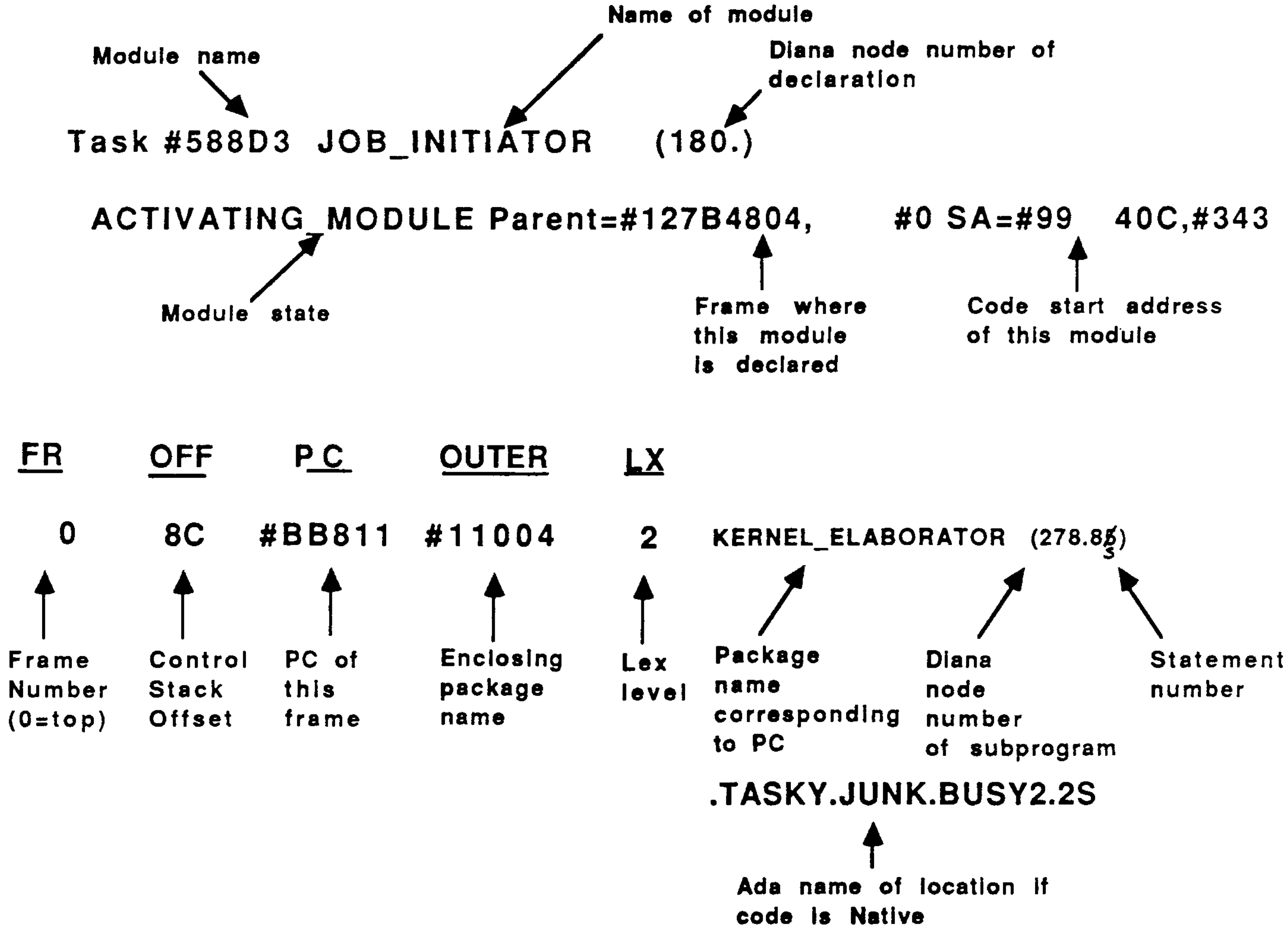
TASKING - SHOW_TASKS COMMAND (CONT'D.)

- Parameters

Task_or_Job_ID	
Show_Stack	if false, only list task names and states
Show_Full_Stack	show each stack word one way or another, not just frames
Show_Hex_Stack	show control words in hex
Show_Packages	display packages as well as tasks
Name_Filter	skip entries that don't include this as a name fragment
Max_Tasks	stop after this many tasks have been found. You many need to increase this to get all tasks in a job.
Names_File	file containing environment package/ segment map
Debug_File	not used

TASKING - SHOW_TASKS COMMAND (CONT'D.)

- Output
  - First, symbol segment files are read
  - Next, stacks are scanned to locate as many modules as possible
  - For each module



Name of module

Module name

Diana node number of declaration

```
Task #588D3  JOB_INITIATOR    (180.)

 ACTIVATING_MODULE Parent=#127B4804,     #0 SA=#99   40C,#343
```

Module state

Frame where this module is declared

Code start address of this module

```
FR     OFF    PC      OUTER     LX

  0     8C   #BB811  #11004     2    KERNEL_ELABORATOR  (278.8~~5~~s)
```

Frame Number (0=top)

Control Stack Offset

PC of this frame

Enclosing package name

Lex level

Package name corresponding to PC

Diana node number of subprogram

Statement number

```
.TASKY.JUNK.BUSY2.2S
```

Ada name of location if code is Native

- No Ada names available for elaboration code, loaded main programs or code views, or environment code if file is missing.

```
User Status on August 2, 1987 at 2:13:54 PM

     User         Line  Job   S      Time    Job Name

============== ==== === ==== ========
                                            ========================================
PHIL           242   223  IDLE   1:26:18       [PHIL.S_1 Command]
                     238  RUN    1:26:27       [PHIL.S_1 Editor]
                     251  RUN      4.382       !USERS.PHIL % WHAT.USERS



    Show_Tasks (Task_Or_Job_Id => 238,
                Show_Stack => True,
                Show_Full_Stack => False,
                Show_Hex_Stack => False,
                Show_Packages => False,
                Name_Filter => "",
                Max_Tasks => 20,
                Names_File => "!Commands.System_Maintenance.@_Mload",
                Debug_File => "!Commands.System_Maintenance.@_Rdebug");

--------------------------------------------------------------------------------
!USERS.PHIL.GURU_COURSE % SHOW_TASKS                        STARTED 2:15:36 PM
--------------------------------------------------------------------------------

Processing file: !COMMANDS.SYSTEM_MAINTENANCE.A_9_18_0_COMBINED_MLOAD'V(1) ...
                                                                 1326 defined.
Processing file: !COMMANDS.SYSTEM_MAINTENANCE.LATEST_COMBINED_MLOAD'V(1) ...
                                                                 1349 defined.
Processing file: !COMMANDS.SYSTEM_MAINTENANCE.UOC931_MLOAD'V(2) ...   33 defined.

Max_Tasks reached;  not all tasks will be displayed


Task    #1A0EE    USER (210.)
    BLOCKING_ON_ENTRY Parent=#198EE, #0 SA=#6B300B, #B
Fr Off         Pc          Outer  Lx
-- --- -------------- -------- --
 0  26  #6B1C0B, #293#12717804  2 ROUTER (225.2s)
 1  1E  #6B300B, #136   #1A0EE  2 USER (643.3s)
 2   0   #6B300B, #59   #1A0EE  1 USER (24.4s)


Task    #198EE    CORE_EDITOR (444.)
    BLOCKING_ON_SELECT Parent=#1F5404, #0 SA=#6B380B, #B
Fr Off         Pc          Outer  Lx
-- --- -------------- -------- --
 0  76  #6B380B, #E53   #198EE  2 CORE_EDITOR (1785.2s)
 1   0  #6B380B, #239   #198EE  1 CORE_EDITOR (30.5s)


Task    #1A4EE    RUN_MACRO (230.)
    BLOCKING_ON_ENTRY Parent=#198EE, #0 SA=#6B2C0B, #B
Fr Off         Pc          Outer  Lx
-- --- -------------- -------- --
 0  1A  #6B1C0B, #35A#12717804  2 ROUTER (319.1s)
 1  15   #6B2C0B, #40   #1A4EE  2 RUN_MACRO (94.1s)
 2   0   #6B2C0B, #28   #1A4EE  1 RUN_MACRO (10.4s)
```

```
Task    #19CEE    INPUT (190.)
    IN_WAIT_SERVICE Parent=#198EE, #0 SA=#6B280B, #B
Fr Off         Pc          Outer  Lx
-- --- -------------- -------- --
 0  AF  #B82C11, #18F   #19C04  2 PROCESSOR_MANAGER (1272.5s)
 1  AB  #B82C11, #143   #19C04  2 PROCESSOR_MANAGER (638.2s)
 2  A7   #18F408, #4F   #67804  2 WAIT_SERVICE (85.1s)
 3  A3 #715008, #1000   #6FC04  2 TCP_IP_DRIVER (541.2s)
 4  97  #715008, #AEF   #6FC04  2 TCP_IP_DRIVER (295.16s)
 5  8F  #546808, #23E#12459404  2 TCP_IP_COUPLER (338.1s)
 6  88  #55B008, #37E  #273404  2 TRANSPORT (372.1s)
 7  81  #718C08, #803  #272804  4 TELNET_PROTOCOL (2155.1s)
 8  76  #718C08, #731  #272804  3 TELNET_PROTOCOL (2129.4s)
 9  6D  #718C08, #572  #272804  2 TELNET_PROTOCOL (2072.2s)
10  63  #718C08, #6CD  #272804  3 TELNET_PROTOCOL (3216.2s)
11  5C  #718C08, #375  #272804  2 TELNET_PROTOCOL (384.9s)
12  4F  #71D808, #19E  #276404  2 TELNET_SERVER (121.11s)
13  46    #6EC09, #F8#12490804  2 TELNET_COUPLER (282.1s)
14  3D  #6C1008, #5F4  #36B004  3 TERMINAL_IO (2100.3s)
15  36  #6C1008, #326  #36B004  2 TERMINAL_IO (300.7s)
16  2C  #6C1008, #37B  #36B004  2 TERMINAL_IO (370.1s)
17  22  #43E008, #42B   #FBC04  2 TERMINAL_MANAGER (412.2s)
18  1A   #6B280B, #A7   #19CEE  2 INPUT (309.2s)
19   0   #6B280B, #38   #19CEE  1 INPUT (18.4s)


Task    #1BCEE    SCREEN_TASK (289.)
    DELAYING_ON_SELECT Parent=#12703004, #0 SA=#68E00B, #B
Fr Off         Pc          Outer  Lx
-- --- -------------- -------- --
 0 1F2 #68E00B, #4E71   #1BCEE  2 SCREEN_TASK (5077.2s)
 1   0  #68E00B, #775   #1BCEE  1 SCREEN_TASK (16.5s)


Task    #1F8EE    COMMAND_EDITOR (419.)
    BLOCKING_ON_SELECT Parent=#12768004, #0 SA=#68A80B, #1713
Fr Off         Pc          Outer  Lx
-- --- -------------- -------- --
 0  5C #68A80B, #23EB   #1F8EE  2 COMMAND_EDITOR (2737.2s)
 1   0 #68A80B, #18B0   #1F8EE  1 COMMAND_EDITOR (30.4s)


Task    #1E4EE    TEXT_OBJECT_EDITOR (311.)
    BLOCKING_ON_SELECT Parent=#22A804, #0 SA=#61000B, #2A43
Fr Off         Pc          Outer  Lx
-- --- -------------- -------- --
 0  6E #61000B, #38DF   #1E4EE  2 TEXT_OBJECT_EDITOR (1205.2s)
 1   0 #61000B, #2BBF   #1E4EE  1 TEXT_OBJECT_EDITOR (6.4s)


Task    #1D4EE    BUFFER_TASK (121.)
    BLOCKING_ON_SELECT Parent=#12717804, #0 SA=#6B180B, #B
Fr Off         Pc          Outer  Lx
-- --- -------------- -------- --
 0  36  #6B180B, #23F   #1D4EE  2 BUFFER_TASK (1102.2s)
 1   0   #6B180B, #9E   #1D4EE  1 BUFFER_TASK (24.4s)


Task    #230EE    ADA_EDITOR (174.)
```

```
    BLOCKING_ON_SELECT Parent=#12766804, #0 SA=#5C080B, #158B
Fr Off         Pc          Outer   Lx
-- --- -------------- -------- --
 0  5C #5C080B, #1E63    #230EE  2 ADA_EDITOR (5041.2s)
 1   0 #5C080B, #1729    #230EE  1 ADA_EDITOR (46.4s)


-----
-------------------------------------------------------------------------------
!USERS.PHIL.GURU_COURSE % SHOW_TASKS                      STARTED 2:31:05 PM
-------------------------------------------------------------------------------

Processing file: !COMMANDS.SYSTEM_MAINTENANCE.LATEST_COMBINED_MLOAD'V(1) ...
                                                                        1349 defined.

scanning stack of #588D3  tos =  147
scanning stack of #5A0D3  tos =  13
scanning stack of #5B0D3  tos =  16
scanning stack of #5B8D3  tos =  42
scanning stack of #5B4D3  tos =  46
scanning stack of #5ACD3  tos =  13
scanning stack of #59CD3  tos =  26
scanning stack of #5BCD3  tos =  51


Task     #588D3    JOB_INITIATOR (180.)
    ACTIVATING_MODULE Parent=#127B4804, #0 SA=#9940C, #343
Fr Off         Pc          Outer   Lx
-- --- -------------- -------- --
 0  8C  #BBB811, #384    #11004  2 KERNEL_ELABORATOR (278.8s)
 1  81   #27009, #813#1243F804  2 ENVIRONMENT_ELABORATOR_DATABASE (1083.1s)
 2  7C  #DD3C16, #1D4   #192004  2 EXECUTION (131.1s)
 3  76  #DD3C16, #12B   #192004  2 EXECUTION (81.4s)
 4  70  #56EC0B, #5A9#1278B004  2 ADA_INTERPRETER (1628.2s)
 5  68  #56EC0B, #102#1278B004  2 ADA_INTERPRETER (18.10s)
 6  59 #68A80B, #1038#12768004  2 COMMAND_EDITOR (4110.34s)
 7  48  #68A80B, #29A#12768004  2 COMMAND_EDITOR (80.4s)
 8  37  #5C080B, #956#12766804  2 ADA_EDITOR (1138.5s)
 9  32  #6B100B, #25B#12798004  2 REGISTRATION (191.1s)
10  2E  #6B140B, #339   #1FA004  2 OE_COUPLER (306.1s)
11  24 #56500B, #2577#127B3C04  2 LIBRARY_OBJECT_EDITOR (820.9s)
12  1F  #AB40C, #1107#127B4004  2 LINE_EDITOR (664.3s)
13   0   #9940C, #3B2    #588D3  1 JOB_INITIATOR (20.6s)


Task     #5B8D3    .TASKY.BUSY
    UNBLOCKED Parent=#59CD3, #14 SA=#21D03, #AB
Fr Off         Pc          Outer   Lx
-- --- -------------- -------- --
 0  31  #2ED808, #167    #4B404  2 BOUNDED (145.2s)
 1  2D   #8DC08, #56F    #4BC04  3 STRING_UTILITIES (644.16s)
 2  29   #8DC08, #553    #4BC04  3 STRING_UTILITIES (644.11s)
 3  21   #8DC08, #232    #4BC04  2 STRING_UTILITIES (120.6s)
 4  1B   #8DC08, #208    #4BC04  2 STRING_UTILITIES (96.1s)    environment (cross)
 5  12    #21D03, #8D    #5B0D3  2 .TASKY.JUNK.BUSY2.2s        native
 6   0    #21D03, #BA    #5B8D3  1 .TASKY.BUSY.7s


Task     #5B4D3    .TASKY.BUSY
    UNBLOCKED Parent=#59CD3, #14 SA=#21D03, #AB
```

```
Fr Off         Pc          Outer   Lx
-- --- -------------- -------- --
 0  2A  #2ED808, #2AD    #4B404  2 BOUNDED (326.1d)
 1  21   #8DC08, #223    #4BC04  2 STRING_UTILITIES (120.1s)
 2  1B   #8DC08, #208    #4BC04  2 STRING_UTILITIES (96.1s)
 3  12    #21D03, #8D    #5B0D3  2 .TASKY.JUNK.BUSY2.2s
 4  3D   #8DC08, #553    #4BC04  3 STRING_UTILITIES (644.11s)
 5  39   #8DC08, #553    #4BC04  3 STRING_UTILITIES (644.11s)
 6  35   #8DC08, #553    #4BC04  3 STRING_UTILITIES (644.11s)
 7  31   #8DC08, #553    #4BC04  3 STRING_UTILITIES (644.11s)
 8  2D   #8DC08, #553    #4BC04  3 STRING_UTILITIES (644.11s)


Package #59CD3    unknown - no table
    DELAYING Parent=#588D3, #8C SA=#43502, #13
Fr Off         Pc          Outer   Lx
-- --- -------------- -------- --
 0  14    #21D03, #42    #5A0D3  2 .TASKY.3s
 1  12    #43102, #1C    #5ACD3  2 unknown - no table
 2   0    #43502, #39    #59CD3  1 unknown - no table


Task     #5BCD3    .TASKY.BUSY
    UNBLOCKED Parent=#59CD3, #14 SA=#21D03, #AB
Fr Off         Pc          Outer   Lx
-- --- -------------- -------- --
 0  29   #8DC08, #56A    #4BC04  3 STRING_UTILITIES (644.16s)
 1  21   #8DC08, #232    #4BC04  2 STRING_UTILITIES (120.6s)
 2  1B   #8DC08, #208    #4BC04  2 STRING_UTILITIES (96.1s)
 3  12    #21D03, #8D    #5B0D3  2 .TASKY.JUNK.BUSY2.2s
 4   0    #21D03, #BA    #5BCD3  1 .TASKY.BUSY.7s

-------------------------------------------------------------------------------
!USERS.PHIL.GURU_COURSE % SHOW_TASKS                      STARTED 2:32:03 PM
-------------------------------------------------------------------------------

Processing file: !COMMANDS.SYSTEM_MAINTENANCE.LATEST_COMBINED_MLOAD'V(1) ...
                                                                        1349 defined.

scanning stack of #588D3  tos =  147
scanning stack of #5A0D3  tos =  13
scanning stack of #5B0D3  tos =  16
scanning stack of #5B8D3  tos =  79
scanning stack of #5B4D3  tos =  63
scanning stack of #5ACD3  tos =  13
scanning stack of #59CD3  tos =  26
scanning stack of #5BCD3  tos =  48


Task     #588D3    JOB_INITIATOR (180.)
    ACTIVATING_MODULE Parent=#127B4804, #0 SA=#9940C, #343


Package #5A0D3    .TASKY
    TERMINABLE_AT_END Parent=#59CD3, #0 SA=#21D03, #B


Package #5B0D3    .TASKY.JUNK'SPEC.1d
    TERMINABLE_AT_END Parent=#59CD3, #14 SA=#21D03, #63
```

```
Task    #5B8D3    .TASKY.BUSY
    UNBLOCKED Parent=#59CD3, #14 SA=#21D03, #AB


Task    #5B4D3    .TASKY.BUSY
    UNBLOCKED Parent=#59CD3, #14 SA=#21D03, #AB


Package #5ACD3    unknown - no table
    TERMINABLE_AT_END Parent=#59CD3, #0 SA=#43102, #B


Package #59CD3    unknown - no table
    DELAYING Parent=#588D3, #8C SA=#43502, #13


Task    #5BCD3    .TASKY.BUSY
    UNBLOCKED Parent=#59CD3, #14 SA=#21D03, #AB
```

TASKING - REVIEW

- Task lifecycle
- TCB
- Task state
- Commands:

  Show_Task_States

  Show_Tasks

  Roust
- Packages and tasks
- Wait service

VIRTUAL PROCESSORS

- A Virtual Processor (VP) is a grouping of resources that the envrionment deals with as a unit.

- All objects on a particular VP exist on the same disk volume.

- All tasks running on a particular VP are scheduled as a unit.

- Part of the segment name of each segment identifies the VP. The bottom 10 bits of the segment name are the VPID number.

- Of the 1024 VPIDs, VP 1 through 255 are used for jobs. Others are used for data objects.

VIRTUAL PROCESSORS - SPECIAL VPs

0 - 3	Reserved for kernel
4	System
5	Daemons
6 - 7	Reserved for System
8 - 25	Cross-compiled code segments
26 - 255	User jobs
256 - 1023	Storage of "permanent" data

VIRTUAL PROCESSORS - JOBS

- The VP number for a Job VP is the same as the job id.

- Information associated with a job

  - Priority
  - Name
  - Class - Attached, Detached, Server, CE, OE
  - State - Run, Wait, Idle, Disabled, Queued
  - Resource use
  - Access control identity
  - Session
  - Root task
  - Start time
  - Elapsed time
  - Termination message
  - Accounting information

- Task Name/VP conversion

Task id (hex)	Job id
4388004	4
1C8E7	E7 = 231
220DF	DF = 223

- Take right two hex digits. Next two higher bits always are zeros. (Why?)

always on VP Ø..255

VIRTUAL PROCESSORS - JOB INFORMATION

- **Resource Use**

CPU MS/S or Percentage	computed by scheduler
Module Count	Number of modules (tasks and packages). Computed by the microcode.
Cache	Number of pages in cache occupied by the jobs modules. Computed by Microcode.
Disk	Number of pages of disk occupied by the job's modules. Computed by the kernel.
Page Limit	Page limit - set by the job.
Disk Waits	Number of times job has waited for disk I/O. Roughly, the number of page faults (including I/O).
Disk Waits/Second	Rate of disk waits.
Job Segment Size	Number of pages in the job's garbage heap.
Working Set Size	Current job working set size in pages. Computed by scheduler.
Working Set Limit	Maximum size working set can be before scheduler discriminates against the job.

VIRTUAL PROCESSORS - JOB INFORMATION (CONT'D.)

- Identity

  Root Task

  The task id of the root module of the job. May be zero for system jobs and editor jobs.

  Job Name

  A string name of the job. Usually the context and command.

  Job Segment Name

  Segment name of the job garbage heap. Environment code allocates space here on behalf of the job.

  Group Identity

  The set of groups the job is a member of. Used for access control.

- Priority

  The priority of the job. User jobs start at 6. Bigger numbers are lower priorities.

VIRTUAL PROCESSORS - GETTING INFORMATION

- From the Environment

What.Users	Shows user, session, elapsed time, state, and name
What.Jobs	Updated OE display of jobs. Can be set for different thresholds.
Show_Jobs	Shows, for individual, active, or all jobs, priority, CPU, status, module, cache, and disk page counts, etc. Very useful for watching resources go.
Show_Job_Names	Similar to Show_Jobs, but displays names, root task, and job segment name.
Show_Identity	Shows access control and user identity of a job.

- From the kernel (or kernel via EEDB)

Jobs	Same as Show_Jobs. Threshold =0 for interesting jobs only, = 1 for all.
Job	Same as Show_Jobs, but for a specific job.
Job_Names	Same as Show_Job_Names
Job_Name	Same as Show_Job_Names, but for a specific job.

VIRTUAL PROCESSORS - GETTING INFORMATION (CONT'D.)

Jobs_MTS	Similar to Show_Jobs, includes scheduler information
Job_MTS	Jobs_MTS for a specific job
Show_VPs	Shows for each VP, the volume the VP is assigned to
MTSq	Show medium term scheduler queues
Hogs	Find big objects on specified VPs

- From programs

```
package    !.Commands.Scheduler           --    get information

package    !.Tools.System_Utilities       --    get information
                                          --      page limits

package    !.Commands.Program             --    create jobs
```

```
What.Users

User Status on August 2, 1987 at 11:57:05 PM

     User       Line  Job   S      Time          Job Name

=============  ====  ===  ====  =========
                                ==================================================

*SYSTEM          -     4  RUN     1/05:41  System
                       5  RUN     1/05:41  Daemons
                     225  IDLE    1/05:32  Print_Spooler
                     227  IDLE    1/05:32  Console Command Interpreter
                     228  IDLE    1/05:31  Problem Report Transfer Server
                     231  IDLE    1/05:32  Ftp Server
                     236  IDLE    1/05:32  Archive Server

NETWORK_PUBLIC   -   217  IDLE    6:30:35  Print Queue Server

PHIL            245  211  RUN     9:26:11  !USERS.PHIL.TEST_AREA %
                                                     !USERS.PHIL.TEST_AREA.TASKY
                     213  IDLE  13:12.090  [PHIL.S_1 Debugger]
                     215  RUN     1:10:41  [PHIL.S_1 Editor]
                     219  IDLE    1:10:36  [PHIL.S_1 Command]
                     220  RUN       1.016  !USERS.PHIL.GURU_COURSE % WHAT.USERS
                     221  IDLE  13:16.808  !USERS.PHIL.TEST_AREA %
                                                     !USERS.PHIL.TEST_AREA.TASKY


    Show_Jobs (Job => 0, Active_Jobs_Only => True);


 Job Pri Stat CPU%  ModCt Cache   Disk  PgLim DskWts D/S JSegSz WsSiz WsLim
---- --- ---- ---- ------ ----- ------ ------ ------ --- ------ ----- -----
   4   0 R,AT    1   4052  9184  12101  65536 116128   0    777 11007 11000
 211   0 R,DT   98      8    17     13   8000     12   0     78    26    50
 213   0 I,SV    0    170   831      7  16000    499   0     18    58    75
 215   6 R,CE    0     47   438     14  16000    400   0     67   164   150
 228   0 I,SV    0      5    12     17   8000    200   0    178    42    75

    Show_Job_Names;


 Job CPU%     Root   Job Seg   Name
---- ---- -------- ---------  -------------------------
   4    0        0   14F9903 System
 211   95    588D3   1277902 !USERS.PHIL.TEST_AREA % !USERS.PHIL.TEST_AREA.T
 213    0    BB4D5   1566503 [PHIL.S_1 Debugger]
 215    0        0   1283D02 [PHIL.S_1 Editor]
 228    0    2B4E4   1133901 Problem Report Transfer Server
```

Hando ·14

```
with Directory;
with Machine;
with Calendar;
with System;

package System_Utilities is

    pragma Subsystem (Tools, Closed);
    pragma Module_Name (4, 3973);

    subtype Job_Id is Machine.Job_Id range 4 .. 255;
    subtype Session_Id is Machine.Session_Id;

    subtype Version is Directory.Version;
    subtype Object is Directory.Object;

    subtype Port is Natural range 0 .. 4 * 16 * 16;
    subtype Tape is Natural range 0 .. 4;

    subtype Byte_String is System.Byte_String;

    -- Job (Process) characteristics
    function Get_Job return Job_Id;
    function Priority
                (For_Job : Job_Id := System_Utilities.Get_Job) return Natural;
    function Elapsed
                (For_Job : Job_Id := System_Utilities.Get_Job) return Duration;
    function Cpu (For_Job : Job_Id := System_Utilities.Get_Job) return Duration;
    function Job_Name
                (For_Job : Job_Id := System_Utilities.Get_Job) return String;


    -- Active Session characteristics
    function Get_Session return Session_Id;
    function Get_Session (For_Job : Job_Id) return Session_Id;

    function Session_Name
                (For_Session : Session_Id := System_Utilities.Get_Session)
                return String;
    function User_Name
                (For_Session : Session_Id := System_Utilities.Get_Session)
                return String;
    function Terminal (For_Session : Session_Id := System_Utilities.Get_Session)
                      return Port;
    function Terminal (For_Session : Session_Id := System_Utilities.Get_Session)
                      return Version;
    function Terminal (For_Session : Session_Id := System_Utilities.Get_Session)
                      return Object;
    function Session (For_Session : Session_Id := System_Utilities.Get_Session)
                     return Version;
    function Session (For_Session : Session_Id := System_Utilities.Get_Session)
                     return Object;
    function User (For_Session : Session_Id := System_Utilities.Get_Session)
                  return Version;
    function User (For_Session : Session_Id := System_Utilities.Get_Session)
                  return Object;

    -- Inactive Session characteristics

    function Home_Library (User : String := User_Name) return String;
```

```
    function Last_Login (User : String; Session : String := "")
                        return Calendar.Time;
    function Last_Logout (User : String; Session : String := "")
                         return Calendar.Time;
    function Logged_In (User : String; Session : String := "") return Boolean;

    -- Names for Text_IO/Simple_Text_IO standard file names

    function Output_Name
                (For_Session : Session_Id := System_Utilities.Get_Session)
                return String;
    function Input_Name
                (For_Session : Session_Id := System_Utilities.Get_Session)
                return String;
    function Error_Name
                (For_Session : Session_Id := System_Utilities.Get_Session)
                return String;
    function Tape_Name (Drive : Tape := 0) return String;

    -- Terminal characteristics

    subtype Stop_Bits_Range is Integer range 1 .. 2;
    subtype Character_Bits_Range is Integer range 5 .. 8;
    type Parity_Kind is (None, Even, Odd);

    function Terminal_Name
                (Line : Port := System_Utilities.Terminal) return String;
    function Terminal_Type
                (Line : Port := System_Utilities.Terminal) return String;

    function Input_Count
                (Line : Port := System_Utilities.Terminal) return Long_Integer;
    function Output_Count
                (Line : Port := System_Utilities.Terminal) return Long_Integer;
    -- The number of characters input/output from/to the specified terminal
    -- since the machine was booted.  Input from the terminal that has not
    -- been read by a session or user program will not be counted as input.

    function Input_Rate
                (Line : Port := System_Utilities.Terminal) return String;
    function Output_Rate
                (Line : Port := System_Utilities.Terminal) return String;

    function Parity (Line : Port := System_Utilities.Terminal)
                    return Parity_Kind;
    function Stop_Bits (Line : Port := System_Utilities.Terminal)
                       return Stop_Bits_Range;
    function Character_Size (Line : Port := System_Utilities.Terminal)
                            return Character_Bits_Range;

    function Xon_Xoff_Characters
                (Line : Port := System_Utilities.Terminal) return String;
    function Xon_Xoff_Bytes
                (Line : Port := System_Utilities.Terminal) return Byte_String;

    function Receive_Xon_Xoff_Characters
                (Line : Port := System_Utilities.Terminal) return String;
    function Receive_Xon_Xoff_Bytes
                (Line : Port := System_Utilities.Terminal) return Byte_String;
    -- returns a 2-element string consisting of Xon followed by Xoff
```

#14

```
    function Flow_Control
                (Line : Port := System_Utilities.Terminal) return String;
    function Receive_Flow_Control
                (Line : Port := System_Utilities.Terminal) return String;
    -- return one of NONE, XON_XOFF, RTS, DTR

    function Enabled (Line : Port := System_Utilities.Terminal) return Boolean;

    function Disconnect_On_Disconnect
                (Line : Port := System_Utilities.Terminal) return Boolean;
    function Logoff_On_Disconnect
                (Line : Port := System_Utilities.Terminal) return Boolean;
    function Disconnect_On_Logoff
                (Line : Port := System_Utilities.Terminal) return Boolean;
    function Disconnect_On_Failed_Login
                (Line : Port := System_Utilities.Terminal) return Boolean;
    function Log_Failed_Logins
                (Line : Port := System_Utilities.Terminal) return Boolean;
    function Login_Disabled
                (Line : Port := System_Utilities.Terminal) return Boolean;
    function Detach_On_Disconnect
                (Line : Port := System_Utilities.Terminal) return Boolean;

    -- Iterate over all active sessions

    type Session_Iterator is private;
    procedure Init (Iter : out Session_Iterator);
    function Value (Iter : Session_Iterator) return Session_Id;
    function Done (Iter : Session_Iterator) return Boolean;
    procedure Next (Iter : in out Session_Iterator);

    -- Iterate over all jobs for a session

    type Job_Iterator is private;
    procedure Init (Iter : out Job_Iterator;
                    For_Session : Session_Id := Get_Session);
    function Value (Iter : Job_Iterator) return Job_Id;
    function Done (Iter : Job_Iterator) return Boolean;
    procedure Next (Iter : in out Job_Iterator);

    type Terminal_Iterator is private;
    procedure Init (Iter : out Terminal_Iterator);
    function Value (Iter : Terminal_Iterator) return Natural;
    function Done (Iter : Terminal_Iterator) return Boolean;
    procedure Next (Iter : in out Terminal_Iterator);

    function System_Up_Time return Calendar.Time;
    function System_Boot_Configuration return String;

    function Image (Version : Directory.Version) return String;



    procedure Set_Page_Limit (Max_Pages : Natural;
                              For_Job : Job_Id := System_Utilities.Get_Job);

    -- Set the upper limit for pages created by the specified job.
    -- Attempts to create additional pages result in Storage_Error.
    -- Requires operator capability if For_Job specifies a job belonging
    -- to a user different from the caller.  If For_Job parameter defaults,
```

```
    -- then the limit applies to the calling job.  In the worst case,
    -- the job can allocate twice Max_Pages pages before getting a storage
    -- error.  Raises constraint_error if the job_id is illegal.

    procedure Get_Page_Counts (Cache_Pages : out Natural;
                               Disk_Pages : out Natural;
                               Max_Pages : out Natural;
                               For_Job : Job_Id := System_Utilities.Get_Job);

    -- Return the counts for the specified job.  Cache_Pages is the number of
    -- pages presently in main memory; Disk_Pages is the number of pages that
    -- have disk space allocated for them.  Max_Pages is the current page
    -- limit.


    -- Operations for reading machine information:

    type Bad_Block_Kinds is new Long_Integer range 0 .. 7;
    Manufacturers_Bad_Blocks : constant Bad_Block_Kinds := 1;
    Retargeted_Blocks : constant Bad_Block_Kinds := 2;
    All_Bad_Blocks : constant Bad_Block_Kinds := 3;

    type Block_List is array (Natural range <>) of Integer;

    function Bad_Block_List
                (For_Volume : Natural;
                 Kind : Bad_Block_Kinds := Retargeted_Blocks) return Block_List;
    -- Return the list of bad blocks of the specified kind on the specified
    -- disk.  Return null array if Kind or volume are illegal.

    function Get_Board_Info (Board : Natural) return String;
    -- return information about the specified board in the machine.  The
    -- string identifies the information.
    --  Board specifies the particular board:
    --      0 : IOA/IOC
    --      1 : SYS
    --      2 : SEQ
    --      3 : VAL
    --      4 : TYP
    --      5 : FIU
    --      100 : MEM0
    --      101 : MEM1
    --      102 : MEM2
    --      103 : MEM3
    --    etc.
end System_Utilities;
```

VIRTUAL PROCESSORS - COMMANDS TO CONTROL THEM

- Environment

  - Job.Kill
  - Scheduler.Set_Job_Attribute
  - Scheduler.Set_WSL_Limits

  Scheduler.Disable
  , Enable

- Kernel

  - Disable_Job — good to stop runaways and look around at them
  - Enable_Job
  - Abort_Task — use on root task to kill job

- If a program has runaway and is using unexpected CPU or disk space (or job segment size large), you can disable it from the kernel and then use Show_ Tasks to dump the job state.

VIRTUAL PROCESSORS - REVIEW

- Volumes
- VPIDs and job ids
- Job information
- Job resources
- Commands:

  What.Users

  What.Jobs

  Show_Jobs

  Show_Jobs_Names

  Show_Identity

  etc.

ndout #15

```
with Machine;

package Scheduler is

    subtype Job_Id is Machine.Job_Id;
    subtype Cpu_Priority is Natural range 0 .. 6;
    subtype Milliseconds is Long_Integer;


    procedure Disable (Job : Job_Id);
    procedure Enable (Job : Job_Id);
    function Enabled (Job : Job_Id) return Boolean;

    function Get_Cpu_Priority (Job : Job_Id) return Cpu_Priority;

    type Job_Kind is (Ce, Oe, Attached, Detached, Server, Terminated);
    function Get_Job_Kind (Job : Job_Id) return Job_Kind;

    type Job_State is (Run, Wait, Idle, Disabled, Queued);
    function Get_Job_State (Job : Job_Id) return Job_State;

    -- returns the current state of job.
    --   RUN:      the job is currently runnable
    --   WAIT:     the job is runnable but being withheld by the scheduler.
    --   IDLE:     the job isn't using cpu time and has no unblocked tasks.
    --   DISABLED: an external agent has disabled the job from running.
    --   QUEUED:   the job is DETACHED and must wait for another to complete.

    function Get_Cpu_Time_Used (Job : Job_Id) return Milliseconds;

    -- returns the number of milliseconds of cpu time used by the job.
    -- belongs on the previous page, put here for compatability reasons

    function Disk_Waits (Job : Job_Id) return Long_Integer;

    -- returns the number of disk_waits the job has done since last initialized

    function Working_Set_Size (Job : Job_Id) return Natural;

    -- returns the number of pages in the job's working set.

    subtype Load_Factor is Natural;

    -- for run queues, number of tasks * 100

    procedure Get_Run_Queue_Load (Last_Sample : out Load_Factor;
                                  Last_Minute : out Load_Factor;
                                  Last_5_Minutes : out Load_Factor;
                                  Last_15_Minutes : out Load_Factor);
    -- number of runnable tasks * 100

    procedure Get_Disk_Wait_Load (Last_Sample : out Load_Factor;
                                  Last_Minute : out Load_Factor;
                                  Last_5_Minutes : out Load_Factor;
                                  Last_15_Minutes : out Load_Factor);
    -- number of tasks waiting for a page on the disk wait queue * 100

    procedure Get_Withheld_Task_Load (Last_Sample : out Load_Factor;
                                      Last_Minute : out Load_Factor;
                                      Last_5_Minutes : out Load_Factor;
```

```
                                      Last_15_Minutes : out Load_Factor);
    -- returns the average number of tasks withheld from running by
    -- the scheduler * 100. In this call LAST_SAMPLE is the number of tasks
    -- held at the last 100ms scheduling cycle.

    procedure State;

    -- print scheduler state

    procedure Display (Show_Parameters : Boolean := True;
                       Show_Queues : Boolean := True);
    -- display current scheduler state and queues

    type Job_Descriptor is
        record
            The_Cpu_Priority : Cpu_Priority;
            The_State : Job_State;
            The_Disk_Waits : Long_Integer;
            The_Time_Consumed : Milliseconds;
            The_Working_Set_Size : Natural;
            The_Working_Set_Limit : Natural;
            The_Milliseconds_Per_Second : Natural;
            The_Disk_Waits_Per_Second : Natural;
            The_Maps_To : Job_Id;
            The_Kind : Job_Kind;
            The_Made_Runnable : Long_Integer;
            The_Total_Runnable : Long_Integer;
            The_Made_Idle : Long_Integer;
            The_Made_Wait : Long_Integer;
            The_Wait_Disk_Total : Long_Integer;
            The_Wait_Memory_Total : Long_Integer;
            The_Wait_Cpu_Total : Long_Integer;
            The_Min_Working_Set_Limit : Long_Integer;
            The_Max_Working_Set_Limit : Long_Integer;
        end record;

    function Get_Job_Descriptor (Job : Job_Id) return Job_Descriptor;

    -- use to get a consistent snapshot of a job's statistics.

    generic
        with procedure Put (Descriptor : Job_Descriptor);
    procedure Traverse_Job_Descriptors (First, Last : Job_Id);

    -- use to get a consistent, efficient snapshot of a range of
    -- job's statistics.


    procedure Set (Parameter : String := ""; Value : Integer);
    function Get (Parameter : String) return Integer;

    -- Programmatic versions of set and display
    --      initial parameters          Default      Units
    --      CPU_Scheduling                 1          1 or 0 (true or false)
    --      Percent_For_Background        10          %
    --      Min_Foreground_Budget       -250          milliseconds (-5000..0)
    --      Max_Foreground_Budget        250          milliseconds (0..5000)
    --      Withhold_Run_Load            130          load * 100
    --      Withhold_Multiple_Jobs         0          1 or 0 (true or false)
```

#15

```
    --      Memory_Scheduling             1          1 or 0 (true or false)
    --      Environment_Wsl           11000          pages
    --      Min_Ce_Wsl                  400          pages
    --      Max_Ce_Wsl                 1000          pages
    --      Min_Oe_Wsl                  250          pages
    --      Max_Oe_Wsl                 2000          pages
    --      Min_Attached_Wsl             50          pages
    --      Max_Attached_Wsl           2000          pages
    --      Min_Detached_Wsl             50          pages
    --      Max_Detached_Wsl           4000          pages
    --      Min_Server_Wsl              400          pages
    --      Max_Server_Wsl             1000          pages
    --      Daemon_Wsl                  200          pages
    --      Wsl_Decay_Factor             50          pages
    --      Wsl_Growth_Factor            50          pages
    --      Min_Available_Memory       2048          pages
    --      Page_Withdrawl_Rate           1          n*640 pages/sec (n in 0..64)

    --      Disk_Scheduling               1          1 or 0 (true or false)
    --      Max_Disk_Load               250          Load_Factor
    --      Min_Disk_Load               200          Load_Factor

    --      Foreground_Time_Limit        60          seconds
    --      Background_Streams            3
    --      Stream_Time N            2,5,20          minutes
    --      Stream_Jobs N             3,0,0          jobs
    --      Strict_Stream_Policy          0          1 or 0 (true or false)


    procedure Set_Job_Attribute (Job : Job_Id;
                                 Attribute : String := "Kind";
                                 Value : String := "Server");
    function Get_Job_Attribute
               (Job : Job_Id; Attribute : String := "Kind") return String;

    -- These interfaces exist to deal with ongoing changes to scheduler
    -- characteristics without requiring new procedures.
    --
    -- The default parameters to Set_Job_Attributes make the indicated job
    -- a server.
    --
    -- See the documentation for other attributes.


    procedure Set_Wsl_Limits (Job : Job_Id; Min, Max : Natural);
    procedure Get_Wsl_Limits (Job : Job_Id; Min, Max : out Natural);
    procedure Use_Default_Wsl_Limits (Job : Job_Id);

    -- Each class of job has a default for working set min and max.
    -- Set_Parameter lets you change the default value. Set_Wsl_Limits lets
    -- you override the default for a specific job.  Use_Default_Wsl_Limits
    -- restores the values to the defaults, cancelling any prior Set_Wsl_Limits
    -- call.
    -- Get_Wsl_Limits returns the current values for a specific job.
    -- Min and Max specify the range (in number of pages) in which the
    -- working set limit is set.  The scheduler chooses the working set
    -- limit based on prevailing conditions on the machine.  If Min and
    -- Max are the same, the a fixed limit is specified.
    -- Min must be less than or equal to Max and Max less than the memory size.
    -- Error messages are sent to an output window in the case of errors.
```

```
    -- No message of any kind if success.


    pragma Subsystem (Os_Commands);
    pragma Module_Name (4, 3923);

end Scheduler;
```

OBJECT MANAGEMENT SYSTEM

- Data stack segments can be created that are not associated with a control stack. The data segments are called heaps or segmented heaps. The are used to store permanent objects in the environment.

- Segmented heaps are managed by a variety of mechanisms at several levels:

Kernel Level: basic mechanisms for creating, copying, and deleting heaps; page differential copies; snapshot coordination.

Object Management Level: specific protocols and uses of heaps for storage of different types of objects; Action Semantics.

Directory Level: association of objects with names; versions; Ada semantics.

OBJECT MANAGEMENT SYSTEM - HEAP MECHANISMS

- Segmented heaps behave as Ada collections. Typed objects are placed in them using special allocator pragmas.

```
type Heap_Ptr is access T;
pragma Segmented_Heap (Heap_Ptr);

X:    Heap_Ptr;

X:=   New T;
pragma Heap (Job_Segment.Get);
```

- Not for customer use. Similar things can be done with Direct_IO, Sequential_IO, Polymorphic_IO, etc.

- The first word in a heap indicates the next offset for allocation.

OBJECT MANAGEMENT SYSTEM - TEMP HEAPS

- Internal structure similar to temp files.
- Each has a marker saying who owns it.
- Temp heaps are deleted during system boot.
- They usually are deleted explicitly by their creator, but this may not happen due to bugs or aborting of the creator.
- Recovery of this space is one reason to reboot on occasion.

OBJECT MANAGEMENT SYSTEM - MANAGERS

- Each object manager provides some type-specific operations to support its object type. Managers generally coordinate access to their objects using an open/close protocol.

- Atomic transactions make it possible to read and update several objects atomically.

- The most significant object managers are:

Ada	provides storage and access for all Ada objects in the system
File	provides storage and access for files
Directory	stores mapping betwen string names and objects. Also stores information about objects.

- Each manager stores its global information in a heap. During operation, new space is allocated in this heap and old space discarded.

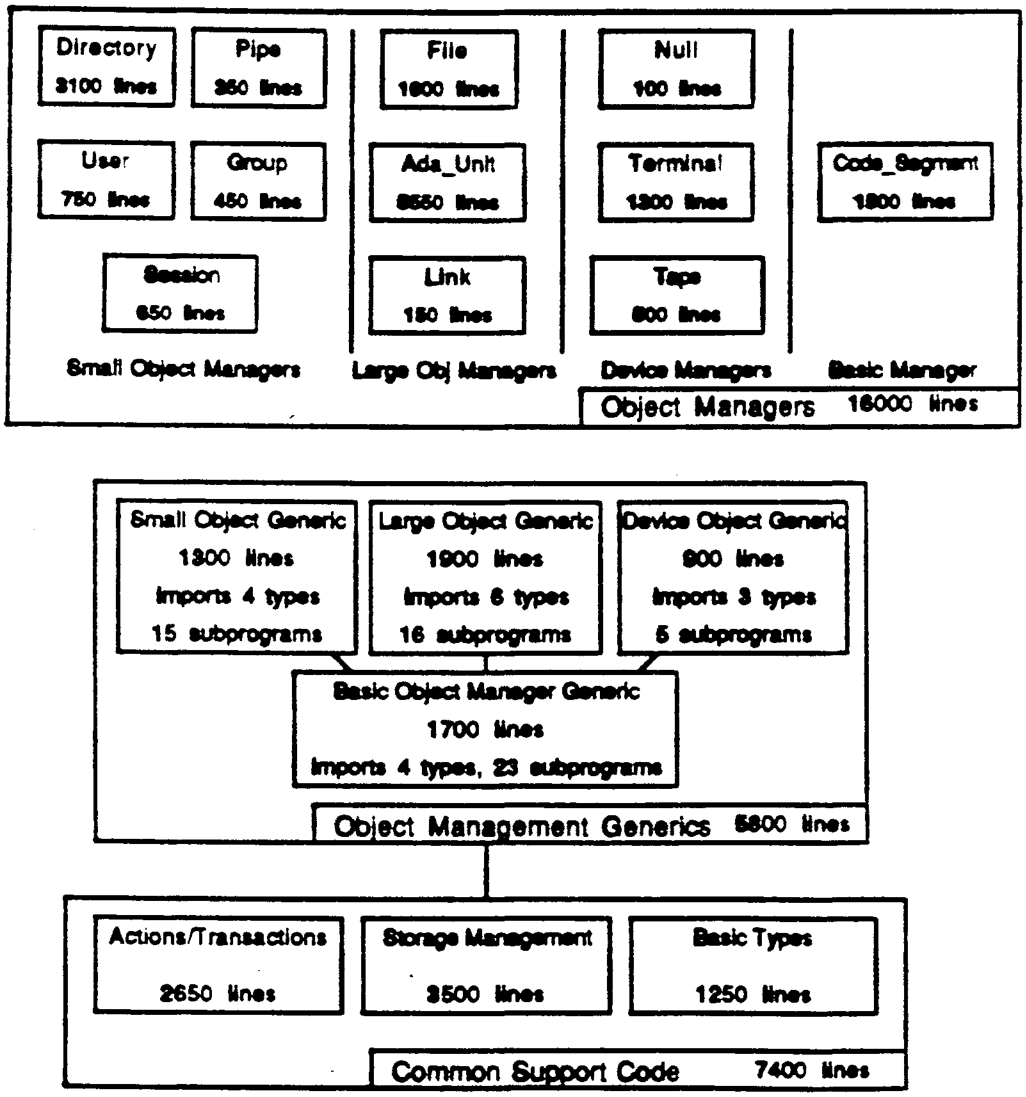
- The daemon makes a copy of the heap, copying only the currently valid information. This is how compaction works.

OBJECT MANAGEMENT SYSTEM - MANAGERS (CONT'D.)

- In addition, large objects that are managed by the object manager are each stored in their own heap or heaps.

- Small objects are stored in the manager heap and do not have their own heaps.

Figure 4-1: Object Management system structure and information. Each of the managers in the Object Managers layer instantiates one of the Object Management Generics. The number of types and subprograms that appear as generic parameters are indicated. All code is based on the common support code.



OBJECT MANAGEMENT - OBJECT IDs

- Each object has a unique object id.

- The object id is a triple:

  <manager id, object number, machine id>

- The manager ids are:

0	Null	6	Session	12	Link
1	Ada	7	Tape	13	Null_Device
2	DDB	8	Terminal	14	Pipe
3	File	9	Directory	15	Archived_Code
4	User	10	Configuration		
5	Group	11	Code_Segment		

- Each manager assigns object numbers to their objects. This is usually done sequentially starting with one.

- The machine id is not really used and is always 1. (Except in a few cases.)

- Given an object id, you can find the name (if any) using:

  Action_Utilities.Display_Object (Class,Instance,Host)

  or

  Lib.Resolve ("<[Class,Instance,Host]>")

- From the directory point of view, each version of an object has an object id and is, from the OM viewpoint, a separate object.

OBJECT MANAGEMENT - ACTIONS

- Actions serve two purposes:

  1) as atomic transactions that can be committed or abandoned
  2) as locks that are held on objects

- An action
  - has a unique action id
  - has an owner task
  - holds locks on zero or more objects
  - tracks changes in those objects

- When committed, the changes take effect and become "permanent" (at the next snapshot).

- When abandoned, the changes are all undone.

- Storing information to undo changes if actions are abandoned is one reason lots of disk garbage is produced doing routine things.

- If a task dies without explicitly committing or abandoning its actions, they are automatically abandoned the next time the action daemon runs or the next time someone attempts to update the affected object.

- Action "compaction" does not compact anything - it only abandons actions owned by dead tasks. The state size will only decrease when the system is shutdown and rebooted (not crashed and rebooted).

- To manually run the action manager daemon:

```
Daemon.Run ("Action");
```

OBJECT MANAGEMENT - SNAPSHOTS

- When a snapshot occurs, the state of each object manager and the action manager are saved. Each object modified since the last snapshot is flushed to disk and marked as committed.

- If the system crashes and is rebooted, during elaboration, each action that was in progress as of the last snapshot is abandoned. Thus, objects being modified when the snapshot occurred are restored to their pre-modified state.

- If there were bad objects lying around, they would be noticed at this point, probably when the action manger stopped in wait service due to a fatal error.

- During the snapshot, for part of the time, all object managers must save their state. During this time, not much else can happen. During the rest of the time, normal operations can proceed.

- To take a snapshot:

  ```
  Daemon.Run ("Snap")
  ```

  From EEDB

  ```
  EEDB: Snap
  ```

- There is a kernel command called snapshot. Never use this. It will irrecoverably destroy the system (most likely).

- To be included, actions must be committed <u>before</u> the snapshot begins.

OBJECT MANAGEMENT - BACKUP AND RECOVERY

- Backup saves address spaces in raw form. Individual objects would be difficult to recover as they are spread across segments and object manager state.

- When backup starts, a snapshot is taken. This is called a retained snapshot because it is saved until the backup completes. It is the contents of this snapshot that is saved.

OBJECT MANAGEMENT - LOCKS

- **The Show_Locks command can be used to display lock information and diagnose lock problems.**

- **There are several other commands to get information about locked objects:**

What.Locks (M-F20) Gives simple information about a locked object, or objects.

Action_Utilities.Lock_Information

More detailed information about locked objects given either object id, string name, or Directory. Version (which is really an Object.Id)

OBJECT MANAGEMENT - ARCHIVE ON SHUTDOWN

- Some software updates require changing the data structures stored by the object managers in their manager heap.

- "Archive on Shutdown" provides a mechanism to do this. When enabled, during shutdown, all managers convert their state to string form.

- On boot, the strings are converted back to internal form.

- Hopefully, the new software was loaded before re-booting.

- Operator.Archive_On_Shutdown (true);

```
Operator.Shutdown
- - -
after booting turn it off

Operator.Archive_On_Shutdown (false)
Daemon.Run ("Snap");
- - -
Op.Show_Shutdown_Settings displays state
```

OBJECT MANAGEMENT - LOCKS

- There is a limit of 1024 actions. If a job starts actions and leaves them pending, the system could run out, or run low.

- Messages and errors "out of action ids" appear in this case.

- Show_Locks can show who has what actions if it is possible to run commands. Parameters:

Show_Job_Action_Summary		
Show_All_Actions	–	show what each action has locked
Job_ID_Filter	–	limit to this job
Action_ID_Filter	–	limit to this action

OBJECT MANAGEMENT - REVIEW

- Segmented heaps
- Object managers
- Object ids
- Actions
- Snapshots
- Backup and recovery
- Locks
- Commands:

```
Action_Utilities.Display_Object

Daemon.Run ("Action")

Daemon.Run ("Snap")

Show_Locks
```