Multics

PL/I Programming with Multics Subroutines

Reference Handbook

Course Code F15C

ISSUE DATE: April 1, 1981

REVISION: 1

REVISION DATE: September, 1983

COURSE DESCRIPTION

F15C PL/I Programming with Multics Subroutines

Duration: Five Days

Intended For: Advanced Multics PL/I programmers who need to use Multics subroutines to perform I/O, manipulate files in the storage system, and/or write commands and active functions.

Synopsis: This course introduces the student to the system subroutine repertoire to include subroutines that: create, delete, develop pointers to, and return status information about storage system entities (hcs_); perform stream and record I/O to files and devices via I/O switches (iox_); enable command and active function procedures to properly interface to the standard command processing environment (cu_). Interactive workshops are included to reinforce the material presented.

Objectives: Upon completion of this course, the student should be able to: write PL/I programs containing calls to system subroutines which:

1. Create, destroy, and obtain status information on segments, directories, and links.
2. Address and manipulate data directly in the virtual memory (without input/output statements).
3. Interface directly with the Multics I/O System (ioa_, iox_).
4. Implement "system standard" commands and active functions.

Prerequisites: Advanced Multics PL/I Programming (F15B) or equivalent experience.

Major Topics:
Advanced Use of Based Variables
Subroutine Interfaces to the Storage System and ACL
Multics Implementation of Condition Handling
The Multics I/O System
Writing Commands and Active Functions

F15C TOPIC MAP

DAY	MORNING TOPICS	AFTERNOON TOPICS
1	WELCOME/ADMINISTRATION REVIEW OF PL/I ATTRIBUTES PL/I STORAGE MANAGEMENT WORKSHOP #1	BASED STORAGE WORKSHOP #2
2	INTRODUCTION TO SUBROUTINES ADVANCED BASED VARIABLE USAGE WORKSHOP #3	MULTICS CONDITION MECHANISM WORKSHOP #4
3	THE MULTICS I/O SYSTEM WORKSHOP #5	THE MULTICS iox_ SUBROUTINE THE MULTICS ioa_ SUBROUTINE WORKSHOP #6
4	STORAGE SYSTEM SUBROUTINES WORKSHOP #7	STORAGE SYSTEM SUBROUTINES (CONTINUED) WORKSHOP #8
5	COMMANDS & ACTIVE FUNCTIONS WORKSHOP #9	REVIEW, QUESTIONS AND WORKSHOP COMPLETION

CONTENTS (con't)

STUDENT BACKGROUND

PL/I Programming with Multics Subroutines (F15C)

NAME: _____ PHONE: _____

TITLE: _____

COMPANY ADDRESS: _____

_____

_____

MANAGER: _____ OFFICE PHONE: _____

INSTRUCTOR'S NAME: _____

1. Do you meet the prerequisite as stated in the "Course Description" of the student text? If yes, check "a" or "b". If no, check "c" or "d".

   a [ ] Prerequisite satisfied by attending course indicated in "Course Description".

   b [ ] Meet prerequisite by equivalent experience (explain briefly)

   _____

   c [ ] Elected or instructed to attend course anyway.

   d [ ] Was not aware of prerequisite.

2. What related Honeywell courses have you attended? Furnish dates and instructors if possible.

   _____

   _____

   _____

(PLEASE TURN OVER)

STUDENT BACKGROUND

PL/I Programming with Multics Subroutines (F15C)

3. Check the boxes for which you have <u>any</u> related experience. (May be other than Honeywell's)

[ ] PL1	[ ] COBOL	[ ] FORTRAN	[ ] ASSEMBLY
[ ] JCL	[ ] OPERATIONS	[ ] GCOS	[ ] MULTICS
[ ] NPS	[ ] GRTS	[ ] CP6	[ ] OTHER

4. Detail any experience you have had which is related to the material in this course.

5. Objectives for attending this course (May check more than one).

[ ] Require information to provide support for a system

[ ] To maintain an awareness of this product

[ ] To evaluate or compare its potentials

[ ] Required to use or implement

[ ] Need update from a previous release

[ ] Require a refresher

[ ] Other:

HONEYWELL MARKETING EDUCATION

COURSE AND INSTRUCTOR EVALUATION FORM

INSTRUCTOR _____

COURSE _____

START DATE _____

LOCATION _____

STUDENT NAME _____ (OPTIONAL)

In the interest of developing training courses of high quality, and then improving on that base, we would like you to complete this questionnaire. Your information will aid us in making future revisions and improvements to this course. Both the instructor and his/her manager will review these responses.

Please complete the form and return it to the instructor upon the completion of the course. In questions 1 through 14, check the appropriate box and feel free to include additional comments. Attach additional sheets if you need more room for comments. Be objective and 'concrete' in your comments -- be critical when criticism is appropriate.

TOPIC I

Review of PL/I Attributes

OBJECTIVES:

Upon completion of this topic, students should be able to:

1. Declare variables in PL/1 using full range of variable attributes.

2. Determine which instance of a variable is being referenced at any given point in a program.

3. Manipulate storage aggregates (arrays and structures).

4. Write and use external procedures.

5. Set up the proper entry declarations to use external procedures.

CLASSIFICATION OF ATTRIBUTES

- A REVIEW LIST OF ATTRIBUTES. STARRED ATTRIBUTES ARE COVERED IN DETAIL IN TOPICS 2, 3 AND 4. THIS CHAPTER PRESENTS USAGE EXAMPLES TO REVIEW/CLARIFY SOME OF THE NON-STARRED ATTRIBUTES

```
storage description
   storage type
      data type
         computational
            arithmetic
               mode: real  complex
               scale: fixed  float
               base: binary  decimal
               precision: precision(p,q)
            string
               string type: character(n)  bit(n)  picture"ps"
               variability: varying  nonvarying
         non-computational
            address
               statement: label  entry  format
               data
                  locator: pointer*  offset*
                  file: file
            area: area(n)*
      aggregate type
         array: dimension(bp,...)
         structure: structure  member
      alignment: aligned  unaligned
   management class
      storage class
         allocation: automatic  static  controlled*  based(lq)*
         sharing: based(lq)*  defined(r)*  position(i)*  parameter
      scope: internal  external
      category: variable  constant
      initial: initial (x,...)
usage description
   entry: entry(d,...)  returns(d,...)  options(variable)
   offset: offset(a)*
   file constant
      operation: input output update
      organization
         stream: stream  print  environment(interactive)
         record: record  sequential  direct keyed
                 environment(stringvalue)
non-valued names
   compile time: like r
   intrinsic names: builtin  condition*
```

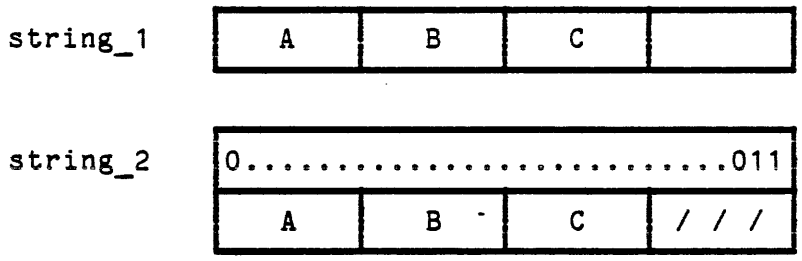
USAGE EXAMPLES OF SELECTED ATTRIBUTES

- ARITHMETIC DATA TYPES

```
dcl x real fixed binary precision (17,0) aligned;

dcl x;          /* SAME AS PREVIOUS DECLARATION */

dcl salary float decimal (6);
```

- STRING DATA TYPES

```
dcl string_1 char(4) init ("ABC");

dcl string_2 char(4) varying init ("ABC");
```



USAGE EXAMPLES OF SELECTED ATTRIBUTES

- STATEMENT LABEL PREFIX (DECLARED BY USAGE, NOT IN FORMAL DECLARATION)

```
continue_1:  x = x + 1;              /* label internal constant */

output_1:  format (a(9),f(6,2));  /* format internal constant */

prog_1: proc;                        /* entry constant */

alternate: entry (a,b);              /* entry constant */
```

- ALIGNMENT

```
dcl string char(4) aligned;       /* DEFAULT IS unaligned */

dcl number fixed bin unaligned;   /* DEFAULT IS aligned */
```

- STATIC VS. AUTOMATIC

```
dcl a init(0);                    /* automatic BY DEFAULT */
dcl b init(0) static;
    a = a + 1;
    b = b + 1;
    put skip list (a,b);
```

USAGE EXAMPLES OF SELECTED ATTRIBUTES

- AGGREGATES

  - ARRAY

    ```
    dcl array_1 (10);
    dcl array_2 (-6:4);
    dcl array_3 (10,3);
    dcl array_4 dimension (5);
    ```

  - STRUCTURE

    ```
    dcl 01 x structure,
           02 y char(8) member,
           02 z fixed bin(35) member;
    ```

    ```
    dcl 01 x, 02 y char(8), 02 z fixed bin(35);
    ```

  - LIKE ATTRIBUTE

    ```
    dcl 1 record_1,
          2 employee_info,
            3 name char(10),
            3 salary fixed dec(10,2);
    ```

    ```
    dcl 1 record_2 like record_1;
    ```

    ```
    dcl 1 employee like record_1.employee_info.name;
    ```

- PARAMETER

  ```
  sub_1:  proc (a,b);

     dcl a char(3) parameter;
     dcl b char(6);     /* parameter ATTRIBUTE USUALLY OMITTED */
  ```

• USAGE EXAMPLES OF SELECTED ATTRIBUTES

• SCOPE OF VARIABLES

```
A:  proc;                  /* SOURCE SEGMENT A.pl1 */
    dcl x external;  /* static by default */
    dcl y;
     .
     .
     .
    B:  proc;
        dcl x;
         .
         .
         .
    end B;
end A;
```

```
C:  proc;                  /* SOURCE SEGMENT C.pl1 */
    dcl x external;
     .
     .
     .
end C;
```

```
D:  proc;                  /* SOURCE SEGMENT D.pl1 */
    dcl x;
    dcl y;
     .
     .
     .
end D;
```

USAGE EXAMPLES OF SELECTED ATTRIBUTES

- VARIABLE VS. CONSTANT

```
dcl x internal static init (125) options (constant);
dcl (file_1, file_2) file;
dcl file_out file variable;

file_out = file_2;
put file (file_out) list ("Test line");
```

- TYPES OF IDENTIFIERS THAT ARE USUALLY USED AS CONSTANTS, BUT MAY BE DECLARED AND USED AS VARIABLES: label, entry, format, file

- INITIALIZATION

```
dcl array_1(5) init(1,2,3,4,5);
dcl array_2(5) init(1,2,(3)*);  /* LAST 3 ELEMENTS UNDEFINED */
dcl array_3(3,2) init(1,2,3,4,5,6);
```

- ENVIRONMENT ATTRIBUTES

```
open file (sysprint) stream output environment (interactive);

put list ("line 1");     /* LINEFEED ADDED AT END AUTOMATICALLY */
put list ("line 2");
```

```
dcl line char(150) varying;
dcl stream_file file;

open file (stream_file) environment (stringvalue) record input
     title ("record_stream_ user_input");

read file (stream_file) into (line);

/* MAKES POSSIBLE TAKING ENTIRE LINE FROM TERMINAL WITH EMBEDDED
   BLANKS WITHOUT USING QUOTES */
```

AGGREGATE DESCRIPTORS

- DESCRIPTORS DESCRIBE THE DATA TYPE AND LAYOUT OF AN IDENTIFIER WITHOUT REFERENCE TO ANY VARIABLE NAMES OR IDENTIFIERS

- DESCRIPTORS ARE USED IN "PARAMETER DESCRIPTOR" LISTS, AND IN "RETURNS DESCRIPTOR" LISTS

  - EXAMPLES

    - declare foo$bar entry (fixed bin, ptr, char(*));

    - declare how_many entry (fixed bin) returns (fixed dec(3,0));

AGGREGATE DESCRIPTORS

- DESCRIPTORS ARE FORMED FOR AGGREGATES AS FOLLOWS:
  - ARRAY DESCRIPTORS
    - ARE DERIVED BY ELIMINATING THE IDENTIFIER FROM THE DECLARATION
    - THE ARRAY BOUNDS MAY BE PRECEDED BY THE 'dimension' OR 'dim' KEYWORD, OR THE KEYWORD MAY BE OMITTED IF THE ARRAY BOUNDS PRECEDE THE DATA TYPE
    - EXAMPLES
      - dcl X(12,3) fixed dec(7);
        - dcl get_X entry ((12,3) fixed dec(7));
        - dcl return_X entry() returns (dim(12,3) fixed dec(7));
  - STRUCTURE DESCRIPTORS
    - ARE DERIVED FROM THE DECLARATION AS FOLLOWS:
      - ELIMINATING ALL IDENTIFIERS
      - NORMALIZING THE LEVEL NUMBERS
    - THE KEYWORDS 'structure' AND 'member' MAY BE OMITTED FROM THE DESCRIPTORS

- EXAMPLE

```
dcl 1 A aligned,
      2 C(3) fixed bin,
      2 F ptr;

dcl get_A entry (1 structure aligned, 2 dim(3) fixed bin
                 member, 2 ptr member);

dcl returns_A entry () returns (1 aligned, 2 (3) fixed bin,
                                2 ptr);

dcl get_A entry (1 like A);

dcl returns_A entry () returns (1 like A);
```

1. Considering the stated objectives of this course, rate the overall length of the course.

CAN'T JUDGE	TOO SHORT				ABOUT RIGHT				TOO LONG
0	1	2	3	4	5	6	7	8	9

COMMENTS_____

_____

_____

2. Considering the objectives, rate the technical level at which the course was taught.

CAN'T JUDGE	NOT TECH ENOUGH				ABOUT RIGHT				TOO TECH
0	1	2	3	4	5	6	7	8	9

COMMENTS_____

_____

_____

3. Considering the objectives, rate the emphasis placed on the more important topics.

CAN'T JUDGE	POOR				GOOD				EXCELLENT
0	1	2	3	4	5	6	7	8	9

COMMENTS_____

_____

_____

4. Rate the sequence in which the topics were presented.

CAN'T JUDGE	POOR				GOOD				EXCELLENT
0	1	2	3	4	5	6	7	8	9

COMMENTS_____

_____

_____

5. Rate the format and quality of the learning materials (slides, student handbooks, supplementary handouts, etc.).

CAN'T JUDGE	POOR				GOOD				EXCELLENT
0	1	2	3	4	5	6	7	8	9

COMMENTS_____

_____

_____

6. Rate the amount of time given for the completion of the workshops.

CAN'T JUDGE	TOO LITTLE TIME				ABOUT RIGHT				TOO MUCH TIME
0	1	2	3	4	5	6	7	8	9

COMMENTS_____

_____

_____

7. Rate the workshops' ability to relate back to and reinforce the material presented.

CAN'T JUDGE	POOR				GOOD				EXCELLENT
0	1	2	3	4	5	6	7	8	9

COMMENTS_____

_____

_____

8. Rate the physical condition of the classroom (space available, temperature, lighting, etc.).

CAN'T JUDGE	POOR				GOOD				EXCELLENT
0	1	2	3	4	5	6	7	8	9

COMMENTS_____

_____

_____

9. Rate the physical condition of the lab or workshop room. (systems configuration, space available, learning tools, terminals, tables, etc.).

CAN'T JUDGE	POOR								GOOD	EXCELLENT
0	1	2	3	4	5	6	7	8	9	

COMMENTS_____

_____

_____

10. Rate your instructor's demonstrated knowledge of the course material.

CAN'T JUDGE	POOR								GOOD	EXCELLENT
0	1	2	3	4	5	6	7	8	9	

COMMENTS_____

_____

_____

11. Rate your instructor's ability to convey the technical aspects of the various topics.

CAN'T JUDGE	POOR								GOOD	EXCELLENT
0	1	2	3	4	5	6	7	8	9	

COMMENTS_____

_____

_____

12. Rate the classroom and workshop assistance given you by your instructor.

CAN'T JUDGE	POOR								GOOD	EXCELLENT
0	1	2	3	4	5	6	7	8	9	

COMMENTS_____

_____

_____

13. Rate the instructor's ability to create an environment in which you felt free to ask questions.

CAN'T JUDGE	POOR				GOOD				EXCELLENT
0	1	2	3	4	5	6	7	8	9

COMMENTS _____

14. Rate the relevance of the skills learned in the course with respect to your job or further training.

CAN'T JUDGE	POOR				GOOD				EXCELLENT
0	1	2	3	4	5	6	7	8	9

COMMENTS _____

15. What did you like <u>most</u> about this course?

16. What did you like <u>least</u> about this course?

17. Other comments please:

_____

_____

_____

_____

_____

_____

_____

18. Of the following job categories, check the ones which most nearly represent the bulk of your experience, and to the right of your responses indicate the number of years you have acted in that capacity.

[] Applications Programmer. . . . _____years

[] Field Engineering Analyst. . . _____years

[] Manager. . . . . _____years

[] Marketing Analyst. . . . . _____years

[] Salesperson. . . . . _____years

[] Secretary. . . . . _____years

[] Systems Analyst. . . . . _____years

[] Systems Programmer . . . . . _____years

[] Other. . . . . _____years

Please give "other" title _____

TOPIC II

PL/I Storage Management

OBJECTIVES:

Upon completion of this topic, students should be able to:

1. Allocate and free controlled variables to implement a stack or a variable-extent data item such as a string or array.

2. Use defined variables to change the interpretation of a particular area of storage.

3. Manipulate cross-sections of arrays using "isub"-defined variables.

DECLARING PL/I VARIABLES

- THE DECLARATION OF AN IDENTIFIER IS USUALLY DIVIDED INTO TWO PARTS
  - THE STORAGE TYPE
    - DESCRIBES THE TYPE OF VALUES WHICH CAN BE ACCOMMODATED
    - DESCRIBES THE AMOUNT AND INTERPRETATION OF STORAGE GENERATED
  - THE STORAGE MANAGEMENT CLASS
    - SPECIFIES VARIOUS INFORMATION ABOUT THE HANDLING OF THE STORAGE GENERATED FOR THE IDENTIFIER INCLUDING
      - THE ALLOCATION AND FREEING MECHANISM TO BE USED
      - THE LOCATION OF THE STORAGE TO BE GENERATED
      - INITIALIZATION OF STORAGE
  - AN EXAMPLE
    - dcl x real fixed binary(10,0) automatic variable init(5);
      - 'real fixed binary(10,0)' IS THE STORAGE TYPE
      - 'automatic variable init(5)' IS THE STORAGE MANAGEMENT CLASS

DEFINING THE PL/I STORAGE MANAGEMENT CLASS

- FOUR ATTRIBUTES SPECIFY THE STORAGE MANAGEMENT CLASS
  - THE 'usage category' ATTRIBUTE
    - DESCRIBES HOW THE STORAGE IS USED
    - VALUES ARE 'variable' AND 'constant'
    - MOST OFTEN, THE USAGE CATEGORY ATTRIBUTE IS OMITTED
  - THE 'scope' ATTRIBUTE
    - PARTIALLY DETERMINES THE REGION IN WHICH THE STORAGE IS ALLOCATED
    - AFFECTS THE ACCESSIBILITY OF THE IDENTIFIER
    - VALUES ARE 'internal' AND 'external'
  - THE 'storage class' ATTRIBUTE
    - SELECTS THE MECHANISM TO BE USED FOR THE ALLOCATION AND FREEING OF THE STORAGE GENERATED
    - VALUES ARE 'automatic', 'static', 'controlled', 'based', 'defined' AND 'parameter'
  - THE 'initial value' ATTRIBUTE
    - WHEN PRESENT, SPECIFIES A VALUE TO BE ASSIGNED TO THE IDENTIFIER WHEN IT IS ALLOCATED
    - VALUE IS 'initial (value_list)'

DEFINING THE PL/I STORAGE MANAGEMENT CLASS ABBREVIATIONS AND DEFAULTS

- VALID ABBREVIATIONS FOR STORAGE MANAGEMENT ATTRIBUTES

ATTRIBUTE	ABBREVIATION
internal	int
external	ext
automatic	auto
controlled	ctl
defined	def
parameter	param
initial	init

- STORAGE MANAGEMENT DEFAULT VALUES

OMITTED ATTRIBUTE	DEFAULT VALUE
usage category	'variable' (exception:'constant' if the data type is 'entry' or 'file')
scope	'internal' (exception:'external' if the data type is 'entry' or 'file')
storage class	'automatic' (exception: 'static' if the 'external' attribute is present or implied)

- NOTE: THE DEFAULTS APPLY TO IDENTIFIERS DECLARED IN A FORMAL DECLARATION STATEMENT. FOR EXAMPLE:
  - A LABEL FORMALLY DECLARED IS A variable BY DEFAULT
  - A LABEL DECLARED BY USAGE AS A LABEL PREFIX IS A constant

'controlled' STORAGE CLASS

CHARACTERISTICS

- 'controlled' STORAGE ALLOWS THE PROGRAMMER TO CONTROL THE GENERATION OF STORAGE FOR A VARIABLE

  - IT IS DRIVEN BY EXPLICIT PROGRAM STATEMENTS

  - STORAGE IS ALLOCATED BY THE 'allocate' STATEMENT, AND FREED BY THE 'free' STATEMENT

  - A 'controlled' VARIABLE IS THEREFORE AVAILABLE FOR WHATEVER PORTION OF EXECUTION OF THE PROGRAM THE PROGRAMMER DESIRES

  - A SMALL CONTROL BLOCK ASSOCIATED WITH THE 'controlled' VARIABLE IS USED TO LOCATE ITS CURRENTLY ALLOCATED STORAGE

  - 'controlled' VARIABLES CAN BE "STACKED"

  - THEY CAN HAVE EITHER 'internal' OR 'external' SCOPE (internal IS THE DEFAULT)

'controlled' STORAGE CLASS

ALLOCATION AND FREEING

- A 'controlled' VARIABLE IS ALLOCATED BY EXECUTION OF THE 'allocate' STATEMENT

  - allocate id;

  - alloc id1, id2, ..., idN;

- A 'controlled' VARIABLE IS FREED BY THE EXECUTION OF THE 'free' STATEMENT

  - free id;

  - free id1, id2, ..., idN;

'controlled' STORAGE CLASS

STACKING 'controlled' VARIABLES

- PL/I ALLOWS US TO ALLOCATE A 'controlled' VARIABLE MORE THAN ONCE BEFORE FREEING ITS STORAGE

  - THE HISTORY OF ALLOCATIONS FOR EACH VARIABLE IS MAINTAINED ON A STACK SO THAT:

    - EACH 'allocate' STATEMENT LEAVES EARLIER ALLOCATIONS OF THAT VARIABLE UNDISTURBED

    - A 'free' STATEMENT FREES THE MOST RECENTLY ALLOCATED SPACE FOR THAT VARIABLE

    - EACH TIME THE VARIABLE IS REFERENCED, THE ONE "ON THE TOP OF THE STACK" IS ACCESSED (MOST RECENTLY ALLOCATED BUT NOT FREED)

  - EXAMPLE

```
P1:  proc;

dcl  x float bin controlled;

       . . . (Computation #1)

       allocate x;
       x = 10;
       . . . (Computation #2)

       allocate x;
       x = 20;
       . . . (Computation #3)

       free x;
       . . . (Computation #4)

       free x;
       . . . (Computation #5)
end;
```

'controlled' STORAGE CLASS

VARIABLE EXPRESSIONS IN ATTRIBUTES

- WHEN A 'controlled' VARIABLE IS ALLOCATED, ANY EXTENT EXPRESSIONS AND INITIAL VALUE EXPRESSIONS ARE EVALUATED

  - EXTENTS ARE ARRAY BOUNDS, MAXIMUM STRING LENGTH, OR AREA SIZE

  - EXTENTS MUST BE SET BEFORE THE EXECUTION OF AN 'allocate' STATEMENT

  - EXTENTS ARE SAVED IN A SYSTEM TEMPORARY

  - EXAMPLE

```
P1:  proc;

dcl n fixed bin init(0);
dcl A(n+2) float bin controlled init((n+2)0);
       .
       .
       .
     n = 2;
     allocate A;
     n = 0;            /*HAS NO EFFECT ON EXTENT*/
     put skip list (A);
     free A;
       .
       .
       .
```

'controlled' STORAGE CLASS

GUIDELINES FOR USING 'controlled' STORAGE

- 'controlled' STORAGE IS GENERALLY MORE EXPENSIVE THAN THE BUILT-IN STORAGE MANAGEMENT MECHANISM OF AUTOMATIC OR STATIC STORAGE CLASSES

- POSSIBLE APPLICATIONS:

  - WHEN A STACK OF VARIABLES IS NEEDED (THIS ALLOWS A PROGRAM WHICH USES STATIC VARIABLES TO BECOME REENTRANT BY REPLACING STATIC VARIABLES WITH 'controlled' VARIABLES)

  - WHEN AN EXTERNAL VARIABLE MUST HAVE VARIABLE EXTENTS ('based' VARIABLES, WHICH COULD HAVE VARIABLE EXTENTS, CANNOT HAVE 'external' SCOPE)

  - WHEN CONTROLLING THE AMOUNT OF STORAGE REQUIRED FOR A PROGRAM BECOMES CRITICAL

'controlled' STORAGE CLASS

GUIDELINES FOR USING 'controlled' STORAGE

- NOTE: PROGRAMS USING 'controlled' VARIABLES SHOULD PROVIDE AN 'on unit' FOR THE 'cleanup' CONDITION IN ORDER TO FREE ANY ALLOCATED STORAGE

- THE 'allocation' BUILTIN FUNCTION RETURNS (IN A fixed bin(17)) THE CURRENT ALLOCATION DEPTH OF STORAGE FOR A 'controlled' VARIABLE

- EXAMPLE

```
dcl cleanup condition;
dcl x controlled;

on cleanup begin;
                do i = 1 to allocation (x);
                   free x;
                end;
          end;
```

'defined' STORAGE CLASS

CHARACTERISTICS

- A 'defined' VARIABLE IS USED TO ASSOCIATE A NEW NAME WITH AN EXISTING VARIABLE OR PART OF AN EXISTING VARIABLE

- IT SUPPLIES A POTENTIALLY DIFFERENT INTERPRETATION (REDEFINITION) OF AN EXISTING GENERATION OF STORAGE

  - IT MUST HAVE THE SAME DATA TYPE AS THE PART OF THE BASE VARIABLE BEING REDEFINED (EXAMPLE: A BIT STRING CANNOT BE 'defined' ON A CHARACTER STRING)

  - IT ALWAYS HAS 'internal' SCOPE

  - SINCE IT NEVER HAS STORAGE ALLOCATED FOR IT, A 'defined' VARIABLE CANNOT HAVE AN 'initial' ATTRIBUTE

- NOTE: USE OF 'defined' VARIABLES IS NOT THE SOLE MEANS OF "REDEFINITION" OF VARIABLES ('based' VARIABLES WILL BE DISCUSSED LATER)

'defined' STORAGE CLASS

CHARACTERISTICS

- THE 'defined' ATTRIBUTE CONSISTS OF THE KEYWORD 'defined' FOLLOWED BY A REFERENCE TO A BASE VARIABLE

- THERE ARE THREE WAYS TO USE 'defined' VARIABLES:

  - SIMPLE DEFINING

  - STRING OVERLAY DEFINING

  - 'isub' DEFINING

'defined' STORAGE CLASS

SIMPLE DEFINING

- EACH SCALER IN THE 'defined' VARIABLE AND THE CORRESPONDING SCALER IN THE BASE VARIABLE HAVE IDENTICAL STORAGE TYPES

  - EXAMPLE 1

```
dcl array(5,5) char(4);
dcl same_array(5,5) char(4) defined array;
dcl vector_1(5) char(4) defined array;
dcl vector_2(5) char(4) defined array(2,1);
```

  - EXAMPLE 2

```
dcl  1 a,
       2 b(n),
         3 c float bin,
         3 d float bin,
       2 e char(6);

dcl  x float defined(a.b(i-2).d);

dcl  Y(n) float defined(a.b(*).d);

dcl  1 z defined(a.b(j)),
       2 z1 float bin,
       2 z2 float bin;
```

- NOTE: THE BASE VARIABLE MAY NOT BE A 'defined' VARIABLE OR A NAMED CONSTANT

'defined' STORAGE CLASS

STRING OVERLAY DEFINING

- A STRING 'defined' VARIABLE IS MAPPED ONTO ALL OR PART OF THE STORAGE OF A STRING BASE VARIABLE

  - VALID FOR ALL STRING TYPES AS LONG AS THEY ARE 'nonvarying unaligned'

  - MUST MATCH BITS ONTO BITS OR CHARACTERS ONTO CHARACTERS

  - PICTURED STRINGS CAN BE USED AS THE BASE VARIABLE, A FACT THAT PROVIDES 'defined' STORAGE ONE OF ITS MOST POWERFUL FACILITIES

    - EXAMPLE

```
dcl a pic "999v.999es99";
dcl exponent char (3) defined (a) position (9);
```

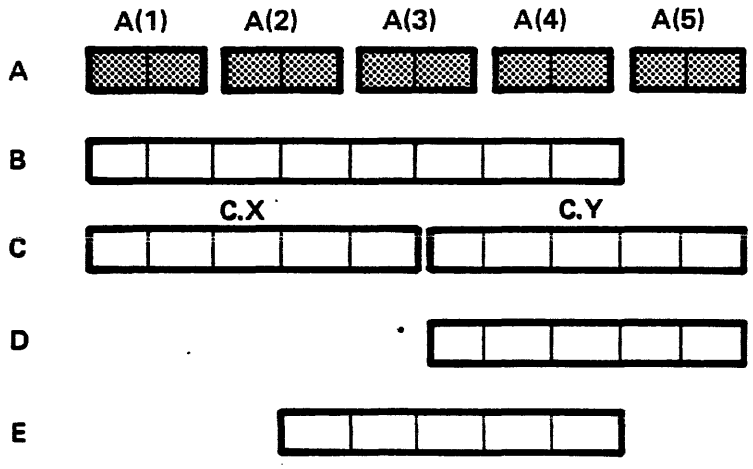
  - THE 'position' OR 'pos' ATTRIBUTE CAN BE USED TO START THE 'defined' VARIABLE AT SOME BIT OR CHARACTER POSITION OTHER THAN THE FIRST

'defined' STORAGE CLASS

STRING OVERLAY DEFINING

▯ EXAMPLES

```
dcl  A(5) char(2) unal;
dcl  B char(8) def(A);
dcl  1 C def(A),
       2 X char(5) unal,
       2 Y char(5) unal;
dcl  D char(5) def(A) pos(6);
dcl  E char(5) def(A(2)) pos(2);
```



'defined' STORAGE CLASS

'isub' DEFINING

- A FACILITY OF PL/I WHICH ALLOWS A 'defined' ARRAY TO MAP ONTO A BASE ARRAY IN A SPECIALIZED MANNER

  - THE VALUE OF THE 'isub' REFERS TO THE SUBSCRIPT OF THE DEFINED ARRAY, NOT THE BASE ARRAY

  - EXAMPLE

```
dcl  A(3,4) float bin;
dcl  Q(3) float bin defined A(1sub,4);
dcl  TRANS(4,3) float bin defined(A(2sub,1sub));

     Q(1) --> A(1,4)

     Q(2) --> A(2,4)

     Q(3) --> A(3,4)
```

  - THE ARRAY 'Q' DEFINES THE FOURTH COLUMN OF 'A'

  - THE ARRAY 'TRANS' REPRESENTS THE TRANSPOSE OF ARRAY 'A'

    - IT REPRESENTS AN INTERPRETATION OF 'A' STORED IN COLUMN-MAJOR ORDER INSTEAD OF ROW-MAJOR ORDER

    - THIS CAN BE USEFUL FOR PASSING ARRAY ARGUMENTS FROM FORTRAN TO PL/I PROGRAMS AND VICE VERSA

'defined' STORAGE CLASS

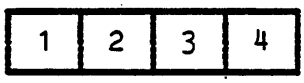
'isub' DEFINING

- CONSIDER A PL/1 2 X 2 ARRAY:

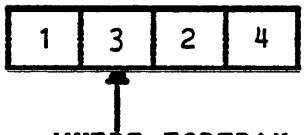
  A(1,1) = 1 A(1,2) = 2

  A(2,1) = 3 A(2,2) = 4

- PL/1 WOULD STORE IT IN MEMORY IN ROW MAJOR ORDER



- FORTRAN WOULD, HOWEVER, STORE IT IN COLUMN MAJOR ORDER



WHERE FORTRAN EXPECTS TO FIND A(2,1)

- PL/I MUST THEREFORE PASS FORTRAN A TRANSPOSE!

```
dcl A(2,2) fixed bin;
dcl transpose_A (2,2) fixed bin
          defined A(2sub,1sub);
      .
      .
      .
call fortran_prog (transpose_A);
```

'defined' STORAGE CLASS

GUIDELINES FOR USING 'defined' STORAGE

- 'defined' STORAGE MANAGEMENT IS "IN COMPETITION" WITH 'based' STORAGE MANAGEMENT
  - 'based' STORAGE MANAGEMENT IS MUCH MORE GENERAL
  - FOR MULTICS, 'based' IS GENERALLY PREFERRED OVER 'defined' STORAGE MANAGEMENT
- USUALLY USED ONLY FOR THE ONE UNIQUE FEATURE PROVIDED -- 'isub' DEFINING

YOU ARE NOW READY FOR WORKSHOP #1

TOPIC III

'based' Storage

OBJECTIVES:

Upon completion of this topic, students should be able to:

1. Allocate and free based variables in the same manner as controlled variables.

2. Differentiate between packed and unpacked pointers.

3. Use builtin functions to manipulate locator variables (pointers and offsets).

4. Use based variables to redefine the interpretation of a particular area of storage.

5. Use the "refer" option to implement self-defining data.

6. Manipulate areas.

CHARACTERISTICS OF 'based' STORAGE

- ADVANCED AND POWERFUL STORAGE MANAGEMENT TECHNIQUE HAVING THREE MAJOR APPLICATIONS
  - EXPLICITLY ALLOCATING AND FREEING SPACE MUCH LIKE CONTROLLED STORAGE
  - EQUIVALENCING TO OR OVERLAYING A TEMPLATE UPON THE STORAGE GENERATED FOR SOME OTHER VARIABLE, MUCH LIKE DEFINED STORAGE
  - ACCESSING A SEGMENT IN THE VIRTUAL MEMORY DIRECTLY, THUS ENABLING I/O TO A SEGMENT WITHOUT USING I/O STATEMENTS
- THE SCOPE OF A 'based' VARIABLE IS ALWAYS 'internal'
- THE DECLARATION OF A 'based' VARIABLE DESIGNATES ONLY THE DATA TYPE AND STORAGE TYPE ATTRIBUTE VALUES FOR THAT VARIABLE
  - IT DOES NOT DESIGNATE THE LOCATION OF THE VARIABLE
  - HENCE, EVERY REFERENCE TO A 'based' VARIABLE MUST BE QUALIFIED WITH A LOCATOR VALUE
  - LOCATOR VALUES CAN BE 'pointer' OR 'offset' VALUES

THE 'based' ATTRIBUTE

- A 'based' VARIABLE IS DECLARED WITH THE KEYWORD 'based' OPTIONALLY FOLLOWED BY A PARENTHESIZED LOCATOR VARIABLE

  - dcl x fixed bin based;

    - EVERY REFERENCE TO 'x' MUST BE QUALIFIED BY A LOCATOR VARIABLE

  - dcl x fixed bin based(p);
    dcl p pointer;

    - THE LOCATOR VARIABLE 'p' IS IMPLICITLY ASSOCIATED WITH 'x'

    - EXPLICIT LOCATOR QUALIFICATION IS NOT NECESSARY (BUT IS RECOMMENDED)

- EVERY 'based' VARIABLE REFERENCE MUST BE QUALIFIED BY A LOCATOR VALUE, EITHER:

  - EXPLICITLY (USING THE -> OPERATOR)

  - OR IMPLICITLY (IF THE VARIABLE WAS DECLARED WITH THE 'based(locref)' ATTRIBUTE)

THE 'based' ATTRIBUTE

- EXAMPLE (EXPLICITLY QUALIFIED)

```
dcl A dec(5,2) based init(0);
dcl p pointer;
dcl sysprint file;
    ...
allocate A set(p);
    ...
p->A = 5;
    ...
put list (p->A);
    ...
free p->A;
```

- EXAMPLE (IMPLICITLY QUALIFIED)

```
dcl n fixed bin;
dcl S char(n+2) based(beta);
dcl beta pointer;
    ...
n = 4;
allocate S;
    ...
S = "abcdef";
    ...
free S;
```

EXPLICITLY ALLOCATED 'based' VARIABLES

- JUST AS IN THE CASE OF 'controlled' VARIABLES, BASED VARIABLES MAY BE EXPLICITLY ALLOCATED AND FREED
  - THE 'allocate' AND 'free' ARE USED

- 'based' VARIABLES MAY BE ALLOCATED IN TWO DIFFERENT WAYS:
  - USING THE 'in (area_name)' OPTION
    - ALLOCATED IN THE 'area' SPECIFIED (ONLY 'based' VARIABLES MAY BE ALLOCATED IN AN 'area')
  - OMITTING THIS OPTION
    - ALLOCATED IN USER FREE AREA WITHIN [pd]>[unique].area.linker

EXPLICITLY ALLOCATED 'based' VARIABLES

THE 'allocate' AND 'free' STATEMENTS

- THE 'allocate' AND 'free' STATEMENTS HAVE THE FOLLOWING FORM WHEN USED FOR 'based' VARIABLES:

  - allocate id [set(locref)] [in(arearef)];

    - WHERE

      - id IS THE NAME OF THE 'based' VARIABLE

      - set(locref) IS USED TO DESIGNATE THE LOCATOR VARIABLE locref AS THE "ADDRESS" OF THE BEGINNING OF STORAGE GENERATED FOR THE 'based' VARIABLE id;

        - MAY BE OMITTED IF THE VARIABLE id WAS DECLARED WITH THE 'based(locref)' ATTRIBUTE

        - locref MUST SPECIFY A pointer OR offset

      - in(arearef) SPECIFIES THE 'area' IN WHICH id IS TO BE ALLOCATED

        - MAY BE OMITTED

  - free id [in(arearef)];

    - WHERE

      - id IS THE 'based' VARIABLE TO BE FREED AND MIGHT HAVE TO BE PTR QUALIFIED

      - in(arearef) IS USED IF THE VARIABLE id WAS ALLOCATED IN THE 'area' arearef (AND IS OTHERWISE OMITTED)

      - NOTE: POINTER IS NULLED AFTER 'based' VARIABLE IS FREED

EXPLICITLY ALLOCATED 'based' VARIABLES

THE 'allocate' AND 'free' STATEMENTS

- EXAMPLE

```
P1:  proc;

dcl  a(5,2) fixed based;
dcl  c char(40) based(p1);
dcl  AREA area;   /* INTERNAL AUTOMATIC, BY DEFAULT */
dcl (p1,p2) pointer;
dcl  sysprint file;

      allocate a set(p2);
      p2 -> a = 0;
      allocate c in(AREA);
      c = "abcdefg";
        .
        .
        .
      put skip(2) data(p2 -> a);
      free p2 -> a, c in(AREA);
   end P1;
```

EXPLICITLY ALLOCATED 'based' VARIABLES

'area' DATA TYPES

- THE PL/I DATA TYPE 'area' PROVIDES A POWERFUL FACILITY FOR STORAGE MANAGEMENT

- BENEFITS OF 'area' MANAGEMENT

  - OPTIONS LIKE ZERO_ON_FREEING, ZERO_ON_ALLOCATING, AND EXTENSIBILITY

  - ENABLES THE USE OF PL/1 OFFSETS

  - EASY FREEING WITH 'empty' BUILTIN

- AN 'area' VARIABLE IS USED BY THE PROGRAMMER AS A MANAGED "POOL" OF FREE STORAGE, TO HOLD 'based' VARIABLES

- THE MAXIMUM SIZE OF A NON-EXTENSIBLE 'area' IS 256K WORDS

  - THE CAPACITY IS ALWAYS SOMEWHAT LESS THAN THIS

    - THE "OCCUPATION RECORD" WHICH RESIDES AT THE BEGINNING OF AN 'area' CATALOGS THE USAGE OF SPACE IN THE 'area'

    - "ALLOCATION RECORDS" PRECEDE EACH BLOCK OF ALLOCATED STORAGE

EXPLICITLY ALLOCATED 'based' VARIABLES

CREATING PL/I AREAS

- AN 'area' MAY BE CREATED IN THREE WAYS:

  - BY THE 'declare' STATEMENT (dcl A area(area_size);)

    - area_size SPECIFIES THE NUMBER OF WORDS TO BE ALLOCATED FOR THE 'area' VARIABLE 'A' (THE DEFAULT IS 1024 WORDS)

    - THE LOCATION OF THE 'area' IS DETERMINED IN THE NORMAL FASHION, BY THE EVALUATION OF THE STORAGE CLASS ATTRIBUTE

      - POSSIBLE ATTRIBUTES ARE static, automatic, internal, external, controlled AND based

      - dcl A area;

        /* automatic - 'A' WOULD BE ALLOCATED ON THE STACK */

      - dcl B area based (get_system_free_area_());
        dcl get_system_free_area_ entry returns (ptr);

        /* 'B' WOULD BE ALLOCATED IN "SYSTEM FREE STORAGE" */

  - BY THE 'define_area_' SUBROUTINE

    - THE CALLER SPECIFIES THE LOCATION OF THE 'area' BY SUPPLYING A POINTER TO A SEGMENT IN WHICH THE 'area' IS TO BE ALLOCATED

      - call define_area_ (info_ptr, code);

    - IF A NULL POINTER IS SUPPLIED, THE SYSTEM ACQUIRES A SEGMENT FOR THE 'area' FROM THE PROCESS DIRECTORY TEMP SEG POOL

    - MUST BE USED IF A BASED AREA IS OVERLAYED UPON ARBITRARY STORAGE

EXPLICITLY ALLOCATED 'based' VARIABLES

CREATING PL/I AREAS

- BY THE 'create_area' COMMAND (AG92)
  - THE COMMAND-LEVEL INTERFACE TO 'define_area_'
    - AT COMMAND-LEVEL: create_area area_seg -extensible

      IN PROGRAM: dcl area_seg$ external area;

EXPLICITLY ALLOCATED 'based' VARIABLES

LOCATOR DATA TYPES

- LOCATORS SPECIFY THE "ADDRESS" OF AN OBJECT, AND ARE USED TO QUALIFY 'based' VARIABLE REFERENCES

- TWO TYPES OF 'locator' VARIABLES:

  - 'pointer'

    - CONTAINS THE ABSOLUTE ADDRESS OF A BIT IN THE VIRTUAL MEMORY

    - MAY BE ALIGNED OR UNALIGNED

      - AN ALIGNED POINTER (DEFAULT)

        - IS DOUBLE WORD ALIGNED

        - IS A PAIR OF WORDS CONTAINING:

          15-BIT SEGMENT NUMBER

          3-BIT RING NUMBER

          6-BIT TAG FIELD CONTAINING OCTAL 43

          18-BIT WORD OFFSET

          6-BIT BIT OFFSET

        - IS DECLARED

          ```
          dcl my_pointer pointer;
          ```

        - IS SOMETIMES REFERRED TO AS AN ITS (INDIRECT TO SEGMENT) PAIR

EXPLICITLY ALLOCATED 'based' VARIABLES

LOCATOR DATA TYPES

- AN UNALIGNED POINTER
  - IS BIT ALIGNED
  - IS A SINGLE WORD CONTAINING

    6-BIT BIT OFFSET

    12-BIT SEGMENT NUMBER

    18-BIT WORD OFFSET
  - IS DECLARED

    dcl my_pointer unal ptr;
  - IS SOMETIMES REFERRED TO AS A PACKED POINTER
  - IS HANDLED BY SPECIAL HARDWARE INSTRUCTIONS
- SINCE ONE OF THE COMPONENTS OF A 'pointer' IS THE SEGMENT NUMBER, THE 'pointer' VALUE IS INVALID ACROSS PROCESS BOUNDARIES

EXPLICITLY ALLOCATED 'based' VARIABLES

LOCATOR DATA TYPES

- 'offset'
  - AN ADDRESS TO A BIT IN AN 'area', RELATIVE TO THE BASE OF THAT 'area'
  - COMPOSED OF A 18 BIT WORD OFFSET AND A 6-BIT BIT OFFSET
  - AN 'offset' DECLARATION MUST BE QUALIFIED BY THE NAME OF THE 'area' INTO WHICH THE 'offset' REFERS IF IT IS TO BE USED IN A 'based' VARIABLE REFERENCE
  - AN 'offset' IS VALID ACROSS PROCESS BOUNDARIES, SINCE IT DOES NOT REFER TO A SEGMENT NUMBER
  - THE PL/I 'offset' ATTRIBUTE IS USED TO DECLARE AN 'offset' VARIABLE
    - dcl off1 offset;
    - dcl off2 offset(A); WHERE 'A' HAS BEEN DECLARED AN 'area'

EXPLICITLY ALLOCATED 'based' VARIABLES

LOCATOR DATA TYPES

- EXAMPLE USING POINTERS AND OFFSETS

```
based_prog: proc;

dcl   sysprint file;
dcl   A area;  /* DEFAULT SIZE IS 1024 WORDS */
dcl   x fixed bin based;
dcl   c char (8) based;
dcl   p ptr;
dcl   o offset(A);

      allocate x set (o) in (A);
      o -> x = 15;
      allocate c set (p);
      p -> c = "abcdefgh";
      put skip data (o -> x, p -> c);
      free o -> x in (A);
      free p -> c;
end based_prog;
```

- RESULT OF RUNNING ABOVE EXAMPLE

```
! based_prog

   x=        15     c="abcdefgh";
```

EXPLICITLY ALLOCATED 'based' VARIABLES

LOCATOR 'builtin' FUNCTIONS

- PL/I BUILTIN FUNCTIONS (AM83) ARE PROVIDED TO CONVERT BETWEEN 'pointer' AND 'offset' LOCATOR DATA TYPES:

  - THE 'pointer' BUILTIN FUNCTION

    - CONVERTS AN 'offset' IN AN 'area' INTO A 'pointer'

    - pointer(X,A)
      ptr(X,A)

      - RETURNS A POINTER POINTING TO 'offset' 'X' IN 'area' 'A'

  - THE 'offset' BUILTIN FUNCTION

    - CONVERTS A 'pointer' WHICH POINTS TO A LOCATION IN AN 'area' INTO THE 'offset' OF THAT LOCATION IN THE 'area'

    - offset(P,A)

      - RETURNS AN 'offset' TO THE 'based' VARIABLE LOCATED BY 'pointer' 'P' IN 'area' 'A'

EXPLICITLY ALLOCATED 'based' VARIABLES

LOCATOR 'builtin' FUNCTIONS

- ADDITIONAL BUILTIN FUNCTIONS FOR THE MANIPULATION OF 'locator' AND 'area' VARIABLES:

  - THE 'null' BUILTIN FUNCTION

    - RETURNS THE VALUE OF THE NULL POINTER, THAT IS, A POINTER TO SEGMENT NUMBER -1 WITH WORD OFFSET 1

    - IS USED TO TEST THE VALIDITY OF 'pointer' VALUES OR TO INITIALIZE THEM

    - NOTE THAT A 'pointer' VARIABLE CAN BE IN ONE OF THREE STATES:

      - UNDEFINED - NO VALUE HAS BEEN ASSIGNED, AND IF USED, 'fault_tag_1' CONDITION IS USUALLY SIGNALLED

      - NULL - THE 'null' BUILTIN HAS BEEN USED TO INITIALIZE THE 'pointer' - AN ATTEMPT TO USE SUCH A 'pointer' USUALLY RESULTS IN THE SIGNALLING OF THE 'null_pointer' CONDITION

      - NON-NULL - A LEGITIMATE ADDRESS HAS BEEN ASSIGNED

  - THE 'nullo' BUILTIN FUNCTION

    - IS USED TO TEST THE VALIDITY OF 'offset' VALUES AND TO INITIALIZE THEM

    - A NULL OFFSET IS ALL "ONES"

EXPLICITLY ALLOCATED 'based' VARIABLES

LOCATOR 'builtin' FUNCTIONS

- THE 'addr' BUILTIN FUNCTION
  - RETURNS THE ADDRESS OF ITS ARGUMENT AS A 'pointer' VALUE
  - addr(x) RETURNS A 'pointer' WHICH LOCATES THE GENERATION OF STORAGE FOR 'x'

- THE 'empty' BUILTIN FUNCTION
  - RETURNS THE "EMPTY" OR "NULL" VALUE OF DATA TYPE 'area'
  - IS USED TO DETERMINE IF AN 'area' IS EMPTY AND IS ALSO USED TO INITIALIZE AN 'area'
  - A "QUICK AND DIRTY" FREEING MECHANISM

- THE NONSTANDARD 'pointer' BUILTIN FUNCTION
  - RETURNS A 'pointer' VALUE GIVEN A 'pointer' POINTING ANYWHERE IN A SEGMENT AND A WORD OFFSET EXPRESSED AS AN ARITHMETIC OR BIT STRING VALUE
  - pointer(P,N) OR ptr(P,N) RETURNS A 'pointer' TO THE Nth WORD OF THE SEGMENT
  - IS DISTINGUISHED FROM THE STANDARD 'pointer' BUILTIN FUNCTION BY THE DATA TYPE OF THE ARGUMENTS

EXPLICITLY ALLOCATED 'based' VARIABLES

LOCATOR 'builtin' FUNCTIONS

- THE NONSTANDARD 'addrel' BUILTIN FUNCTION
  - RETURNS A 'pointer' TO A WORD RELATIVE TO ANOTHER POINTER
  - addrel (P,N) POINTS TO A WORD N WORDS AWAY FROM P
    - THE RESULTING POINTER HAS A 0 BIT OFFSET, REGARDLESS OF P'S BIT OFFSET
    - N IS AS IN THE ABOVE NONSTANDARD pointer BUILTIN

EXPLICITLY ALLOCATED 'based' VARIABLES

USING EXPLICITLY ALLOCATED 'based' STORAGE

- EXPLICITLY ALLOCATED 'based' STORAGE IS GENERALLY USED FOR ONE OF THREE PURPOSES:
  - TO DIRECTLY CONTROL THE ALLOCATION AND FREEING OF STORAGE
  - TO PROVIDE STORAGE FOR DATA ITEMS WHOSE EXTENTS ARE NOT KNOWN AT COMPILE TIME
  - TO TAKE ADVANTAGE OF CERTAIN FEATURES MADE AVAILABLE THROUGH THE USE OF 'area' VARIABLES
    - ZERO ON ALLOCATION
    - ZERO ON FREEING
    - MASS FREEING OF ALLOCATED VARIABLES
    - EXTENSIBILITY OF AREAS

EXPLICITLY ALLOCATED 'based' VARIABLES

USING EXPLICITLY ALLOCATED 'based' STORAGE

- EXPLICITLY ALLOCATED 'based' VARIABLES CAN BE USED TO PROVIDE STORAGE FOR DATA ITEMS WHOSE EXTENTS ARE NOT KNOWN AT COMPILE TIME
  - ADJUSTABLE EXTENTS ARE ARRAY BOUNDS, MAXIMUM STRING LENGTHS, AND 'area' SIZES
  - UNLIKE 'controlled' VARIABLES, FOR 'based' VARIABLES, THE VALUES OF VARIABLE EXTENTS ARE COMPUTED FOR EACH REFERENCE
    - THAT IS, THE ADJUSTED EXTENTS ARE NOT SAVED WHEN THE VARIABLE IS FIRST ALLOCATED
    - IT IS THE RESPONSIBILITY OF THE PROGRAM TO PRESERVE SUCH EXTENTS TO AVOID VIOLATING THE PL/I CONSISTENCY RULES

EXPLICITLY ALLOCATED 'based' VARIABLES

USING EXPLICITLY ALLOCATED 'based' STORAGE

- EXAMPLE OF AN INVALID PROGRAM

```
P1: proc;

dcl   n fixed bin;
dcl   S char(n+2) based(beta);
dcl   beta pointer;
        . . .
      n = 4;
      allocate S;
        . . .
      n = 100;
      S = "abcdef";
        . . .
      free S;
    end;
```

- THIS PROGRAM IS INVALID
  - WHEN THE 'based' VARIABLE 'S' IS ALLOCATED, IT IS GIVEN 6 BYTES OF STORAGE
  - WHEN IT IS REFERENCED IN THE ASSIGNMENT STATEMENT, THE EXTENTS ARE RECOMPUTED TO 102, AND THE STRING "abcdef" WILL BE PADDED TO A LENGTH OF 102 BEFORE BEING ASSIGNED

EXPLICITLY ALLOCATED 'based' VARIABLES

THE 'refer' OPTION

- SINCE THE VARIABLE EXTENTS OF 'based' VARIABLES ARE NOT SAVED BY PL/I, A SPECIAL FEATURE, THE 'refer' OPTION IS PROVIDED

  - IT IS USED TO SAVE THE VALUE CALCULATED FOR VARIABLE EXTENTS OF A 'based' VARIABLE WHEN IT IS ALLOCATED

  - IT IS USED WITHIN A STRUCTURE VARIABLE TO CREATE A "SELF-DEFINING STRUCTURE", WHICH CARRIES ITS OWN EXTENTS

EXPLICITLY ALLOCATED 'based' VARIABLES

THE 'refer' OPTION

- A VALID EXAMPLE

```
P3: proc;

dcl   n fixed bin;
dcl   1 Spair based(beta),
        2 n2 fixed bin,
        2 S char(n+2 refer(n2));
dcl   beta ptr;
       . . .
       n = 4;
       allocate Spair;
       . . .
       n = 100;
       Spair.S = "abcdef";
       . . .
       free Spair;
end P3;
```

- NOTE: A PARENTHESIZED REFERENCE FOLLOWING THE KEYWORD 'refer' MUST DESIGNATE A SCALAR MEMBER DEFINED EARLIER IN THE SAME STRUCTURE

- AT ALLOCATION TIME, ANY INITIAL EXTENT EXPRESSION IS EVALUATED, AND IS SAVED IN THE MEMBER REFERENCED BY THE 'refer' OPTION CLAUSE

- ON SUBSEQUENT REFERENCES TO THE 'based' ADJUSTABLE VARIABLE, THE EXTENT IS DETERMINED BY REFERRING TO THE MEMBER

EXPLICITLY ALLOCATED 'based' VARIABLES USING 'area' VARIABLES

- EXPLICITLY ALLOCATED 'based' VARIABLES MAY BE USED TO TAKE ADVANTAGE OF THE STORAGE MANAGEMENT FACILITIES OFFERED BY THE PL/I 'area' VARIABLES

- NOTE THAT THE ONLY TYPE OF VARIABLE WHICH MAY BE ALLOCATED IN AN 'area' IS AN EXPLICITLY ALLOCATED 'based' VARIABLE

- NOTE ALSO THAT PL/1 'offset' VALUES CAN ONLY LOCATE STORAGE WITHIN AREAS

EQUIVALENCED 'based' STORAGE

- THE USE OF EQUIVALENCED 'based' VARIABLES IS ONE OF THE MOST POWERFUL STORAGE MANAGEMENT CAPABILITIES OFFERED BY PL/I

- UNLIKE EXPLICITLY ALLOCATED 'based' VARIABLES, AN EQUIVALENCED 'based' VARIABLE:

  - IS SUPERIMPOSED ON OR EQUIVALENCED TO A PREVIOUSLY ALLOCATED "BASE" VARIABLE

  - NEVER HAS STORAGE OF ITS OWN, AND THUS IS NEVER ALLOCATED OR FREED

- THE LOCATOR VALUE USED TO REFERENCE THE BASE VARIABLE IS OBTAINED BY THE 'addr' BUILTIN FUNCTION

- EXAMPLE

```
dcl a fixed bin (35);
dcl b fixed bin (35) based (addr(a));

     a = 5;
     b = 2;
     put skip list (a,b);
```

EQUIVALENCED 'based' STORAGE

- ADDITIONAL EXAMPLES (NOTE: FOR THESE EXAMPLES, THE DATA TYPE OF THE 'based' VARIABLE IS THE SAME AS THAT OF THE BASE VARIABLE)

  - EXAMPLE 1

```
P1: proc;

dcl  x fixed dec(5,2);
dcl  y fixed dec(5,2) based;
dcl  p ptr;
dcl (sysin,sysprint) file;

       p = addr(x);
       get list(x);
       put skip list(2 * p->y);
end P1;
```

  - EXAMPLE 2

```
      . . .
dcl  1 A(5),
       2 x fixed bin,
       2 y char(6);
dcl  1 B based,
       2 r fixed bin,
       2 s char(6);
dcl  p ptr;
      . . .
    p = addr(A(3));
      . . .
    p -> B.s = "third";

 /* SETS A(3).y TO "third" */
```

EQUIVALENCED 'based' STORAGE

- IT IS ALSO POSSIBLE FOR THE DATA TYPES OF THE 'based' AND BASE VARIABLE TO DIFFER

  - EXAMPLE 1

```
dcl x fixed bin(35);
dcl y bit(36) based (addr(x));

     x = 5;
     put skip list (x,y);
```

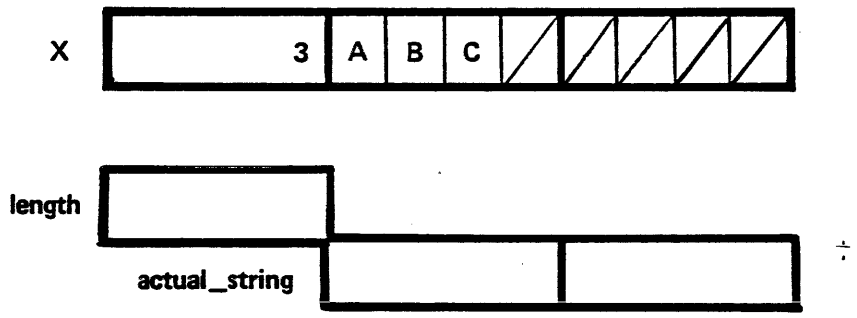
  - EXAMPLE 2

```
dcl  number(1024) float bin;
dcl  1 float_num based,
       2 sign bit(1) unal,
       2 exponent bit(7) unal,
       2 m_sign bit(1) unal,
       2 mantissa bit(27) unal;
         . . .
     p = addr(number(43));
```

    - p -> float_num MEANS number(43)
    - p -> sign MEANS bit 0 of number(43)
    - p -> mantissa MEANS bits 9-35 of number(43)

EQUIVALENCED 'based' STORAGE

```
dcl   x    char(8) varying init("ABC");

dcl 1 y  based  (addr(x)),
        2  length fixed bin (35),
        2  actual_string char (8);
```




```
x = "BONJOUR";
if  y.length = 7
then put list (y.actual_string);
```

AN APPLICATION FOR 'based' VARIABLES

LINKED INFORMATION STRUCTURES

- EQUIVALENCED 'based' STRUCTURES CAN BE USED TO PROVIDE STORAGE FOR DATA ITEMS WHICH HAVE BEEN ORGANIZED INTO AN ARBITRARILY LINKED INFORMATION NETWORK

  - SINGLY AND DOUBLY LINKED LISTS

    - TERMINATING LISTS

    - CIRCULAR LISTS

  - TREES AND OTHER DIRECTED GRAPHS

  - OTHER INFORMATION NETWORKS

- IT SHOULD BE NOTED THAT SUCH STRUCTURES ARE HEAVILY USED IN THE SUPERVISOR, AND THAT MOST OF THE SUPERVISOR DATABASES ARE 'based' STRUCTURES DEFINED IN "INCLUDE FILES" SUBORDINATE TO >ldd>include

AN APPLICATION FOR 'based' VARIABLES

LINKED INFORMATION STRUCTURES

- AN EXAMPLE (from stack_frame.incl.pl1)

```
dcl 1 stack_frame based(sp) aligned,
      2 pointer_registers(0 : 7) ptr,
      2 prev_sp pointer, /* points to previous stack frame */
      2 next_sp pointer, /* points to next stack frame */
      2 return_ptr pointer,
      2 entry_ptr pointer,
      2 operator_and_lp_ptr ptr,
      2 arg_ptr pointer,
      2 static_ptr ptr unaligned,
      2 support_ptr ptr unaligned,
      2 on_unit_relp1 bit(18) unaligned,
      2 on_unit_relp2 bit(18) unaligned,
      2 translator_id bit(18) unaligned,
      2 operator_return_offset bit(18) unaligned,
      2 x(0: 7) bit(18) unaligned,
      2 a bit(36),
      2 q bit(36),
      2 e bit(36),
      2 timer bit(27) unaligned,
      2 pad bit(6) unaligned,
      2 ring_alarm_reg bit(3) unaligned;
```

- THERE ARE OVER 2000 SUCH INCLUDE FILES IN >ldd>include (TOPIC 5 DEMONSTRATES THEIR USAGE)

YOU ARE NOW READY FOR WORKSHOP

#2

TOPIC IV

Introduction to Multics Subroutines

OBJECTIVES:

Upon completion of this topic, students should be able to:

1. Give reasons for having a set of Multics subroutines.

2. Give general guidelines for use of Multics system subroutines.

3. List some of the conventions followed when using Multics system subroutines.

WHAT ARE SYSTEM SUBROUTINES?

- SYSTEM SUBROUTINES ARE CALLABLE PROCEDURES USED BY THE MULTICS OPERATING SYSTEM

  - THEY ARE THE SUBROUTINES THAT THE PROGRAMMER USES TO PERFORM COMMAND LEVEL LIKE FUNCTIONS

  - THEY ARE THE PROCEDURES ACTUALLY CALLED BY COMMAND PROCEDURES (EXAMPLE: THE delete COMMAND PROCEDURE CALLS THE delete_ SUBROUTINE)

    - SOME SUBROUTINES HAVE A ONE-TO-ONE RELATION WITH MULTICS COMMANDS (EXAMPLE: send_message_ SUBROUTINE PERFORMS THE send_message COMMAND FUNCTION FROM WITHIN A PROGRAM)

    - OTHER SUBROUTINES PERFORM ONLY A SMALL PART OF WHAT AN ENTIRE COMMAND DOES. EXAMPLES:

      - iox_ SUBROUTINES ARE USED BY SEVERAL COMMANDS

      - convert_date_to_binary_ IS JUST ONE OF MANY SUBROUTINES CALLED BY THE enter_abs_request AND memo COMMANDS

SYSTEM SUBROUTINE CONVENTIONS

- SYSTEM SUBROUTINE ENTRY NAMES END IN AN UNDERSCORE (_)

- MANY SUBROUTINES HAVE SEVERAL ENTRY POINTS

  hcs_$list_acl
  hcs_$make_seg
  hcs_$status_

- THEY ARE DOCUMENTED IN MULTICS SUBROUTINES & I/O MODULES (AG93)

- THEY ARE LOCATED PRIMARILY IN >system_library_standard AND >system_library_1

- THEY ARE WRITTEN IN PL/I OR ALM

USING SYSTEM SUBROUTINES

- SINCE THEY ARE EXTERNAL SUBROUTINES, EACH MUST BE DECLARED IN THE USER'S PROGRAM AS 'external entry'
  - THE DATA TYPES FOR THE PARAMETER LIST CAN BE FOUND IN THE MANUAL DESCRIPTION OF THE SUBROUTINE
  - IF THEY ACCEPT A VARIABLE NUMBER OF ARGUMENTS, THEY ARE DECLARED 'entry options (variable)'

- SEVERAL MAKE USE OF STRUCTURES TO PASS DATA TO AND FROM THE CALLING PROCEDURE
  - IN THIS CASE, ONE OF THE ARGUMENTS PASSED TO THE PROCEDURE IS A POINTER TO THAT STRUCTURE
  - THE DECLARATIONS REQUIRED FOR THESE STRUCTURES ARE FOUND IN THE DOCUMENTATION FOR THE SUBROUTINE
  - THE DECLARATIONS OF SOME OF THESE STRUCTURES ARE FOUND IN INCLUDE FILES IN >ldd>include
  - EXAMPLE: hcs_$status_
    - THIS SUBROUTINE IS PASSED A POINTER TO A STRUCTURE INTO WHICH IT IS TO PUT ITS INFORMATION
    - A DECLARATION FOR THAT STRUCTURE IS FOUND IN >ldd>include>status_structures.incl.pl1 (FURTHER DISCUSSED IN TOPIC 10)

STATUS CODES

- ONE OF THE OUTPUT ARGUMENTS OF A SUBROUTINE IS USUALLY A 'status code'

  - THE 'status code' IS THE MEANS BY WHICH THE CALLED PROCEDURE MAY REPORT ANY UNUSUAL OCCURRENCE TO ITS IMMEDIATE CALLER

  - THE VARIABLE THAT RECEIVES THE 'status code' MUST BE DECLARED 'fixed bin(35)'

  - IF THE SUBROUTINE RUNS TO COMPLETION WITH ABSOLUTELY NO ABNORMAL CONDITIONS TO REPORT, THE STATUS CODE IS 0 (ZERO)

- com_err_

  - USED TO REPORT ERRORS FROM WITHIN A PROGRAM

  - TYPICAL USAGE

```
dcl com_err entry options (variable);
dcl code fixed bin(35);
...
call hcs_$status_ (.............,code);
if code ^= 0
then do;
          call com_err_ (code, "gamma");
          return;
     end;
```

  - IF AN ERROR OCCURRED, IT MIGHT PRINT SOMETHING LIKE:

    gamma: Incorrect access to directory containing...

  - SOME NON-ZERO STATUS CODES DO NOT INDICATE AN ERROR

STATUS CODES

- STATUS CODES AND THEIR MEANINGS ARE LISTED IN CHAPTER 7 OF THE MULTICS PROGRAMMER'S REFERENCE GUIDE (AG91)

- THE STANDARD STATUS CODES AND THEIR CORRESPONDING MESSAGES ARE IN A SEGMENT CALLED error_table_, WHICH IS IN >sl1

- IT IS POSSIBLE TO TEST FOR A PARTICULAR STATUS CODE VALUE USING THE SYMBOLIC REPRESENTATION

```
dcl error_table_$segknown external fixed bin(35);
...
if code = error_table_$segknown
then do;
          call com_err_ (code, "beta");
          goto try_again;
     end;
```

▯ THE probe 'display' REQUEST CAN BE USED TO DISPLAY THE ERROR MESSAGE ASSOCIATED WITH A STATUS CODE

```
segknown: proc;

dcl  initiate_file_     entry (char(*), char(*), bit(*), ptr,
                              fixed bin(24), fixed bin(35));
dcl  seg_ptr            pointer;
dcl  bit_count          fixed bin (24);
dcl  code               fixed bin (35);
dcl  null               builtin;

call initiate_file_ (">udd>MED>jcj>15c", "foo", "101"b, seg_ptr,
                     bit_count, code);

end /* segknown */;


r 11:41 0.100 3

! segknown
  Stopped after line 10 of segknown. (level 5)
! sc
  call initiate_file_ (">udd>MED>jcj>15c", "foo", "101"b, seg_ptr
                       bit_count, code);
! v seg_ptr
  seg_ptr = null
! v code
  code = 8589679427
! display code code
  error_table_$noentry  "Entry not found."
! q
  r 11:42 0.733 86

! ls foo
  list: foo not found
  r 11:42 0.212 11
```

TOPIC V

Advanced Based Variable Usage

OBJECTIVES:

Upon completion of this topic, students should be able to:

1. Use Multics subroutines to manipulate segments directly instead of using PL/1 I/O statements.

2. Manipulate archive components using Multics subroutines.

3. Examine some system databases using based structures and Multics subroutines.

GAINING DIRECT ACCESS TO SEGMENTS

MOTIVATION

- EQUIVALENCED BASED VARIABLES CAN BE USED TO GAIN DIRECT ACCESS TO SEGMENTS IN THE VIRTUAL MEMORY
  - IN THIS WAY, AN ENTIRE DATA SEGMENT CAN BE ACCESSED WITHOUT RESORTING TO LANGUAGE I/O
  - ONE MUST OBTAIN A 'pointer' TO THE SEGMENT IN ORDER TO GAIN DIRECT ACCESS TO IT
  - THE FOLLOWING PAGES SHOW SUBROUTINES THAT RETURN A POINTER TO A SEGMENT

GAINING DIRECT ACCESS TO SEGMENTS

OBTAINING A POINTER TO A SEGMENT

- MULTICS SUBROUTINES WHICH OBTAIN A 'pointer' TO A SEGMENT:

  - hcs_$make_seg

    - BASIC FUNCTIONS

      - SEGMENT CREATION IF IT DOES NOT EXIST
      - SEGMENT INITIATION

    - USAGE

```
dcl hcs_$make_seg entry
    (char(*),                  /* INPUT */
     char(*),                  /* INPUT */
     char(*),                  /* INPUT */
     fixed bin(5),             /* INPUT */
     ptr,                      /* OUTPUT */
     fixed bin(35));           /* OUTPUT */

call hcs_$make_seg
     (dir_name,      /* PATH OF CONTAINING DIR */
      entryname,     /* SEGMENT NAME */
      ref_name,      /* DESIRED REFERENCE NAME */
      mode,          /* ACCESS FOR THIS USER */
      seg_ptr,       /* POINTS TO CREATED/FOUND SEG */
      code);         /* STATUS CODE */
```

GAINING DIRECT ACCESS TO SEGMENTS

OBTAINING A POINTER TO A SEGMENT

- NOTES
  - IF SEGMENT DOESN'T EXIST, APPEND PERMISSION REQUIRED ON CONTAINING DIRECTORY
  - MAKING-KNOWN REQUIRES NONNULL ACCESS ON SEGMENT
  - IF entryname IS NULL, UNIQUE SEGNAME IS GENERATED
  - IF dir_name IS NULL, SEGMENT IS CREATED IN PROCESS DIRECTORY
  - ref_name USUALLY NULL
  - mode ENCODES THUSLY

```
READ ->    01000b
EXECUTE -> 00100b
WRITE ->   00010b
```

  - seg_ptr IS RETURNED NULL IF REAL TROUBLE WAS ENCOUNTERED
  - code MIGHT BE NON-ZERO UNDER 'NORMAL' CIRCUMSTANCES:

```
error_table_$namedup
error_table_$segknown
```

GAINING DIRECT ACCESS TO SEGMENTS

OBTAINING A POINTER TO A SEGMENT

- IF THE PROGRAMMER DOESN'T CARE IF THE SEGMENT ALREADY EXISTS OR IS ALREADY INITIATED HE RELIES ONLY ON THE NON-NULL seg_ptr

```
dcl hcs_$make_seg entry (char (*), char (*), char (*),
                  fixed bin (5), ptr, fixed bin (35));
dcl com_err_ entry options (variable);
    .
    .
    .
call hcs_$make_seg(.......seg_ptr, code);
if seg_ptr = null()
then do;
       call com_err_ (code, "alpha");
          ...
     end;
```

- IF THE PROGRAMMER EXPECTS TO BE CREATING A NEW SEGMENT AND DOES NOT WANT TO REFERENCE AN ALREADY EXISTING SEGMENT, HE MUST CHECK THE CODE

```
dcl hcs_$make_seg entry (char (*), char (*), char (*),
                  fixed bin (5), ptr, fixed bin (35));
dcl com_err_ entry options (variable);
dcl error_table_$namedup fixed bin(35) ext static;
dcl error_table_$segknown fixed bin(35) ext static;
    .
    .
    .
call hcs_$make_seg (.........seg_ptr, code);
if seg_ptr = null() | code = error_table_$segknown
                    | code = error_table_$namedup
then do;
       call com_err_ (code, "alpha");
          ...
     end;
```

GAINING DIRECT ACCESS TO SEGMENTS

OBTAINING A POINTER TO A SEGMENT

- initiate_file_
  - BASIC FUNCTIONS
    - MAKES A SEGMENT KNOWN WITH A NULL REFERENCE NAME
    - CHECKS THAT THE USER'S PROCESS HAS AT LEAST THE DESIRED ACCESS ON THE SEGMENT
    - RETURNS A POINTER TO THE SEGMENT
    - RETURNS A BIT COUNT
  - USAGE

```
dcl initiate_file_ entry
    (char(*),                /* INPUT */
     char(*),                /* INPUT */
     bit(*),                 /* INPUT */
     pointer,                /* OUTPUT */
     fixed binary (24),      /* OUTPUT */
     fixed binary (35));     /* OUTPUT */

call initiate_file_
     (dirname,          /* PATH OF CONTAINING DIR */
      entryname,        /* SEGMENT NAME */
      mode,             /* REQUIRED ACCESS MODE */
      seg_ptr,          /* POINTS TO INITIATED SEG */
      bit_count,        /* BIT COUNT OF SEGMENT */
      code);            /* STANDARD SYSTEM CODE */
```

OBTAINING A POINTER TO A SEGMENT

- NOTES
  - THE SEGMENT MUST EXIST
  - MAKING-KNOWN REQUIRES NONNULL ACCESS ON THE SEGMENT, AS WELL AS THE REQUIRED MODES SPECIFIED IN THE CALL
  - mode ENCODES THUSLY

```
READ ->    "100"b
EXECUTE -> "010"b
WRITE ->   "001"b
```

    (>ldd>include>access_mode_values.incl.pl1 CONTAINS NAMED CONSTANTS FOR THESE ACCESS MODES)

  - seg_ptr IS NULL IF THE SEGMENT IS NOT MADE KNOWN
  - code IS A STANDARD STATUS CODE AND COULD BE:

```
error_table_$no_r_permission
error_table_$no_e_permission
error_table_$no_w_permission
```

GAINING DIRECT ACCESS TO SEGMENTS

OBTAINING A POINTER TO A SEGMENT

- initiate_file_$component
  - BASIC FUNCTIONS
    - MAKES EITHER A SEGMENT OR AN ARCHIVE COMPONENT KNOWN WITH A NULL REFERENCE NAME
    - IF NO COMPONENT NAME IS SPECIFIED, THIS ENTRY POINT IS IDENTICAL TO initiate_file_
  - USAGE

```
dcl  initiate_file_$component entry
     (char (*),              /* INPUT */
      char (*),              /* INPUT */
      char (*),              /* INPUT */
      bit (*),               /* INPUT */
      pointer,               /* OUTPUT */
      fixed binary (24),     /* OUTPUT */
      fixed binary (35));    /* OUTPUT */

call initiate_file_$component
     (dirname,          /* PATH OF CONTAINING DIR */
      entryname,        /* NAME OF SEGMENT OR ARCHIVE */
      component_name,   /* NULL OR NAME OF COMPONENT */
      mode,             /* REQUIRED ACCESS MODE */
      component_ptr,    /* PTR TO SEGMENT OR COMPONENT */
      bit_count,        /* BIT COUNT OF SEGMENT OR COMPONENT */
      code);            /* STANDARD SYSTEM CODE */
```

  - NOTES
    - THE ARCHIVE COMPONENT MAY NOT BE MODIFIED (ONLY READ ACCESS IS PERMITTED)
    - ONLY THE DATA STARTING AT THE POINTER AND EXTENDING AS FAR AS THE BIT COUNT MAY BE REFERENCED (NO DATA BEFORE OR AFTER THE COMPONENT MAY BE REFERENCED)

GAINING DIRECT ACCESS TO SEGMENTS

OBTAINING A POINTER TO A SEGMENT

- TO OBTAIN A POINTER TO A COMPONENT WITHIN AN ARCHIVE SEGMENT SEE
  - archive_$get_component
  - archive_$next_component

- NOTE THAT THE SUBROUTINES DISCUSSED REQUIRE AN ABSOLUTE DIRECTORY PATHNAME

- THE expand_pathname_ SUBROUTINE CAN BE USED TO CONVERT A PATHNAME (WHETHER RELATIVE OR ABSOLUTE) INTO THE REQUIRED DIRECTORY PATHNAME AND ENTRYNAME STRINGS

- USAGE

```
dcl expand_pathname_ entry
    (char(*), char(*), char(*), fixed bin(35));

call expand_pathname_
     (rel_path,     /* RELATIVE OR ABSOLUTE PATHNAME
                       TO BE EXPANDED */
      dir_name,     /* RETURNED DIRECTORY PORTION OF
                       PATHNAME */
      entryname,    /* RETURNED ENTRYNAME PORTION OF
                       PATHNAME */
      code);
```

GAINING DIRECT ACCESS TO SEGMENTS

AN EXAMPLE

```
stack_tracer: proc;

%include stack_header;
%include stack_frame;

dcl  com_err_          entry options (variable);
dcl  get_pdir_         entry () returns (char (168));
dcl  initiate_file_    entry (char (*), char (*), bit (*), pointer,
                              fixed binary (24), fixed binary (35));
dcl  interpret_ptr_    entry (ptr, ptr, ptr);

dcl  bit_count         fixed binary (24);
dcl  code              fixed bin (35);
dcl  ME                char (12) static
                       init ("stack_tracer") options (constant);
dcl  no_frames         fixed bin;
dcl 1 owner,
      2 message        char (64),
      2 segname        char (32),
      2 entryname      char (33);
dcl (save_ptr, sp,
     shp)              ptr;
dcl  sysprint          file;
dcl (addr,
     ltrim,
     null)             builtin;


/* GET POINTER TO BASE OF STACK SEGMENT */

call initiate_file_ (get_pdir_ (), "stack_4", "100"b,
                     shp, bit_count, code);
if shp = null ()
then do;
        call com_err_ (code, ME);
        return;
     end /* then do */;

/* WALK FRAMES TO FIND LAST ONE */

no_frames = 0;
do sp = shp -> stack_header.stack_begin_ptr
        repeat sp -> stack_frame.next_sp
        while (sp ^= shp -> stack_header.stack_end_ptr);
   save_ptr = sp;
   no_frames = no_frames + 1;
end /* do sp */;

/* NOW TRACE BACKWARDS AND DUMP */
```

GAINING DIRECT ACCESS TO SEGMENTS

AN EXAMPLE

```
   do sp = save_ptr
         repeat sp -> stack_frame.prev_sp
         while (sp ^= null ());
      call interpret_ptr_ (sp -> stack_frame.entry_ptr, sp,
                           addr (owner));
      put skip (2) edit ("FRAME", no_frames, " IS OWNED BY ",
                         rtrim(owner.segname), rtrim(owner.entryname))
                        (a,f(3),a,a,a);
      put skip list ("   FRAME STARTS AT", sp);
      put skip list ("   ARG POINTER IS", sp -> stack_frame.arg_ptr);
      no_frames = no_frames -1;
   end /* do sp */;

   /* ALL DONE */

   put skip (2) list ("End stack_tracer");
   put skip;
   close file (sysprint);

   end /* stack_tracer */;


   r 14:08 0.237 6

 ! stack_tracer


   FRAME  5 IS OWNED BY stack_tracer$stack_tracer
      FRAME STARTS AT        pointer(234|5640)
      ARG POINTER IS         pointer(234|5202)

   FRAME  4 IS OWNED BY command_processor_$command_processor_
      FRAME STARTS AT        pointer(234|5000)
      ARG POINTER IS         pointer(234|4274)

   FRAME  3 IS OWNED BY abbrev$abbrev_cp
      FRAME STARTS AT        pointer(234|2700)
      ARG POINTER IS         pointer(234|2564)

   FRAME  2 IS OWNED BY listen_$listen_
      FRAME STARTS AT        pointer(234|2400)
      ARG POINTER IS         pointer(234|2236)

   FRAME  1 IS OWNED BY initialize_process_$initialize_process_
      FRAME STARTS AT        pointer(234|2000)
      ARG POINTER IS         pointer(234|0)

   End stack_tracer
   r 14:09 0.658 46
```

GAINING DIRECT ACCESS TO SEGMENTS

AN EXAMPLE

YOU ARE NOW READY FOR WORKSHOP #3

TOPIC VI

Multics Condition Mechanism

OBJECTIVES:

Upon completion of this topic, students should be able to:

1. Describe the actions taken by Multics when a condition is signalled.

2. Write handlers for the following conditions:

   cleanup

   program_interrupt

   finish

   User-defined and PL/1-defined conditions

3. Write an "any_other" handler.

4. Discuss the circumstances under which the system-defined conditions occur.

INTRODUCTION

- THE MULTICS CONDITION MECHANISM IS A FACILITY THAT NOTIFIES A PROGRAM OF AN EXCEPTIONAL CONDITION
  - A CONDITION IS A STATE OF THE EXECUTING PROCESS
  - A CONDITION MAY OR MAY NOT INDICATE THAT AN ERROR HAS OCCURRED

- IN MULTICS, THERE ARE THREE BROAD CATEGORIES OF CONDITIONS:
  - SYSTEM-DEFINED CONDITIONS (MULTICS LEVEL)
    - ARE DEFINED AS PART OF THE MULTICS SYSTEM
    - ARE DETECTED BY THE MULTICS HARDWARE OR SOFTWARE
    - ARE SIGNALLED BY THE MULTICS SUPERVISOR
    - EXAMPLES
      - cleanup
      - no_read_permission
      - out_of_bounds
      - quit
      - record_quota_overflow
      - AND OTHERS, TO BE DISCUSSED LATER

INTRODUCTION

- LANGUAGE-DEFINED CONDITIONS
  - ARE DEFINED AS PART OF PL/I
  - ARE DETECTED AND SIGNALLED BY THE PL/I RUNTIME PROCESSOR
  - EXAMPLES
    - conversion
    - endfile
    - AND OTHERS...
- PROGRAMMER-DEFINED CONDITIONS
  - ARE DEFINED BY THE PROGRAMMER
  - ARE DETECTED AND SIGNALLED EXPLICITLY BY THE PROGRAMMER
  - EXAMPLES
    - oops
    - OR WHATEVER ONE DESIRES...

INTRODUCTION

- THE MULTICS CONDITION MECHANISM IS INVOKED WHEN A CONDITION IS DETECTED AND SIGNALLED BY:

  - THE SYSTEM

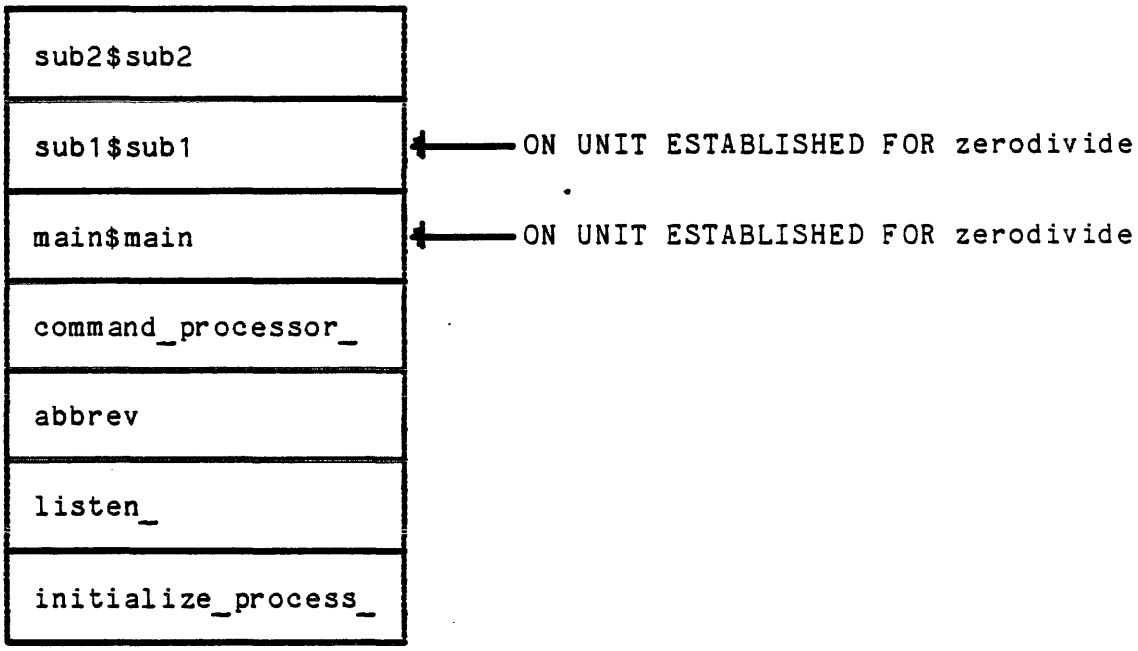
    - EXAMPLE: zerodivide OCCURS

  - THE USER PROGRAM

    - EXAMPLE: "signal zerodivide;"

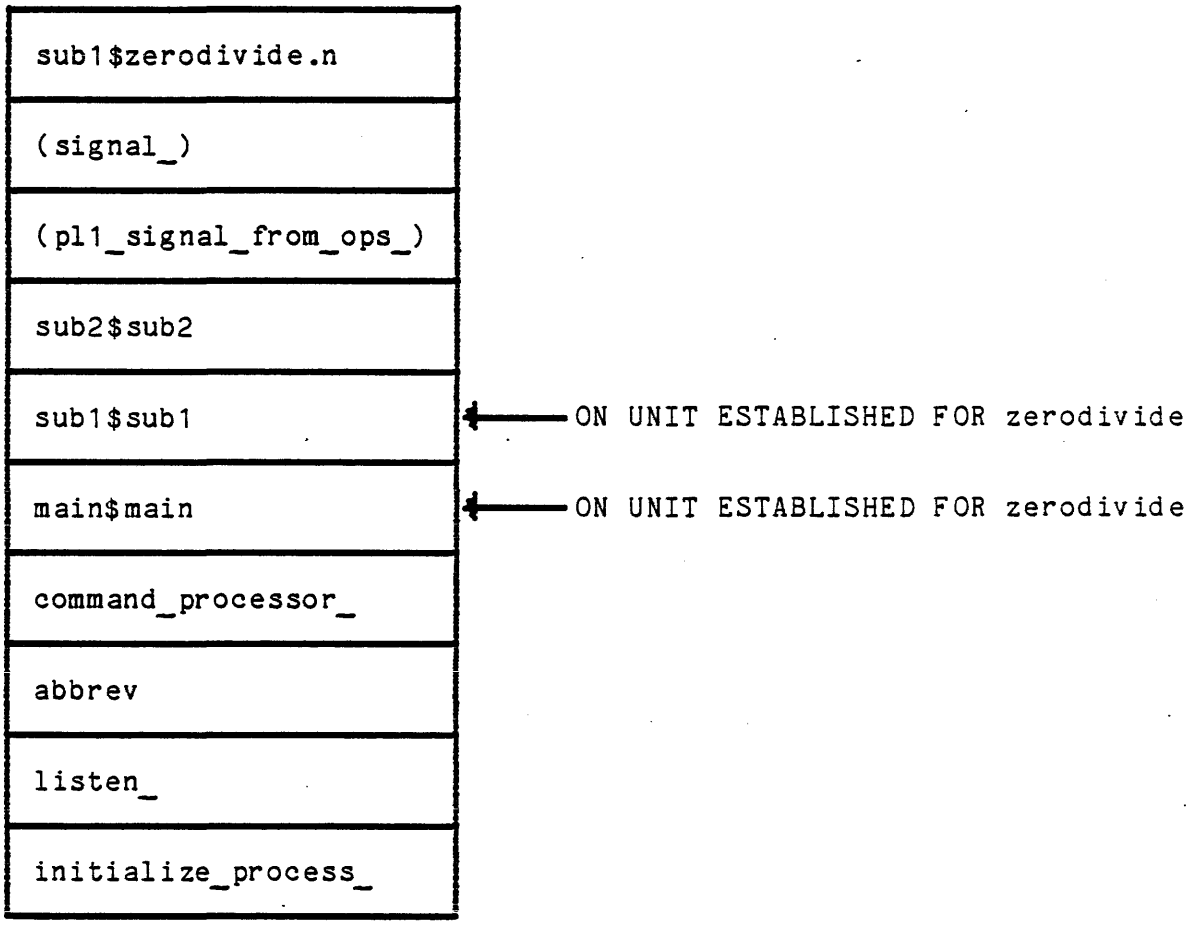
INTRODUCTION

- THE SIGNALLING OF A CONDITION:
  - IMMEDIATELY STOPS THE PROGRAM AT THE CURRENT POINT OF EXECUTION
  - CAUSES A BLOCK ACTIVATION OF THE MOST RECENTLY ESTABLISHED ON UNIT FOR THAT CONDITION
    - THE APPROPRIATE ON UNIT IS FOUND BY MAKING A BACKWARDS TRACE OF THE STACK
    - EACH BLOCK ACTIVATION ON THE STACK CAN HAVE ONLY ONE ON UNIT ESTABLISHED FOR EACH CONDITION AT ANY GIVEN TIME



USER STACK

- IF zerodivide IS SIGNALLED IN sub2, A BLOCK IS ACTIVATED FOR THE ON UNIT ESTABLISHED IN sub1



USER STACK

ESTABLISHING AND REVERTING CONDITION HANDLERS

- EXAMPLES OF ESTABLISHING CONDITION HANDLERS

  - on zerodivide begin;
    ...
    ...
    end;

  - on zerodivide system;

  - on zerodivide snap system;

    - IF THE CONDITION SPECIFIED IS SIGNALLED, THE 'probe' COMMAND IS IMMEDIATELY INVOKED BEFORE THE 'on unit' IS INVOKED (FOR AN ABSENTEE PROCESS, THE 'trace_stack' COMMAND IS EXECUTED)

  - on zerodivide call probe;

- THERE ARE THREE WAYS TO REVERT AN 'on unit'

  - PL/I 'revert' STATEMENT (EXAMPLE: revert zerodivide;)

  - BLOCK DEACTIVATION CAUSED BY REACHING A BLOCK 'end' STATEMENT

  - NON-LOCAL 'go to' WHICH CAUSES DEACTIVATION OF OF ALL BLOCKS FROM THE TOP OF THE STACK TO THE PROCEDURE CONTAINING THE LABEL THAT IS THE TARGET OF THE 'go to'

This Page Intentionally Left Blank

- EXAMPLE OF THE CONDITION MECHANISM

```
example:  proc;

dcl sub1 external entry;
dcl sub2 external entry;
dcl overflow condition;

on overflow <on unit 1>;

   call sub1;

   <statement 1>;

   call sub2;
end /* example */;
```

```
sub1: proc;

dcl overflow condition;

   <statement 2>;

on overflow <on unit 2>;

   <statement 3>;
end /* sub1 */;
```

```
sub2: proc;

dcl overflow condition;

   <statement 4>;

on overflow <on unit 3>;

   <statement 5>;

revert overflow;

   <statement 6>;
end /* sub2 */;
```

ESTABLISHING AND REVERTING CONDITION HANDLERS

- ASSUME THAT EACH OF THE 6 NUMBERED STATEMENTS IN THE 3 PROCEDURES ON THE PREVIOUS PAGE IS A SIMPLE ASSIGNMENT STATEMENT (THERE ARE NO goto's)

  FILL IN THE CHART SHOWING WHICH 'on unit' WOULD BE INVOKED IF 'overflow' OCCURRED IN THE NUMBERED STATEMENT SPECIFIED

STATEMENT CAUSING overflow TO BE SIGNALLED	ON UNIT INVOKED
1	_____
2	_____
3	_____
4	_____
5	_____
6	_____

A SPECIAL CATCH-ALL CONDITION HANDLER

- THE 'any_other' CONDITION REFERS TO CONDITIONS FOR WHICH NO 'on unit' HAS BEEN SPECIFICALLY ESTABLISHED

  - EXAMPLE

```
dcl (zerodivide, overflow, any_other) condition;

on zerodivide begin;
   ...
end;

on any_other begin;
   ...
end;

signal overflow;
```

  - BACKWARD TRACE OF STACK LOOKS FOR CONDITION HANDLER TWICE FOR EACH FRAME:

    - LOOKS FOR SPECIFIC CONDITION HANDLER FIRST

    - LOOKS FOR CONDITION HANDLER FOR 'any_other' SECOND

  - THE 'cleanup' CONDITION IS AN EXCEPTION IN THAT IT DOES NOT INVOKE THE any_other HANDLER

ACTION TAKEN IF NO 'on unit' IS FOUND ON STACK

- THERE IS A DEFAULT HANDLER 'default_error_handler_'

- THE PROGRAM, initialize_process_, HAS ONLY ONE 'on unit' (FOR THE CONDITION any_other)

  - THE any_other CONDITION HANDLER CALLS default_error_handler_$wall

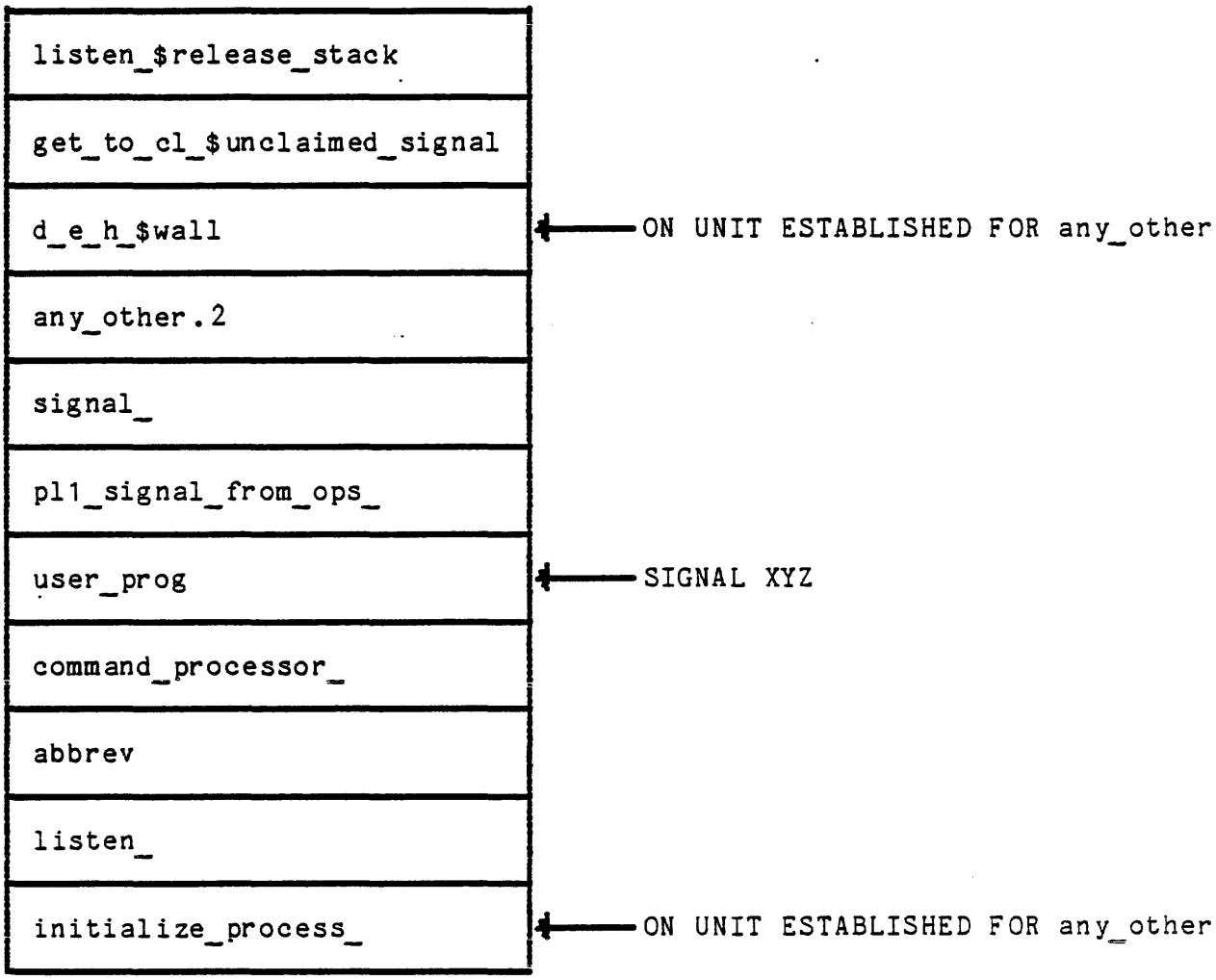
  - default_error_handler_ CHECKS TO SEE WHICH CONDITION WAS SIGNALLED

    - EXECUTES DIFFERENT CODE BASED ON THE CONDITION

    - NOTIFIES USER IF IT WAS NOT SET UP TO HANDLE CONDITION (EXAMPLE: USER DEFINED CONDITIONS AND program_interrupt

    - SEVERAL CONDITIONS RESULT IN CALL TO get_to_cl_$unclaimed_signal

ACTION TAKEN IF NO 'on unit' IS FOUND ON STACK



ACTION TAKEN IF NO 'on unit' IS FOUND ON STACK

- default_error_handler_$wall SETS UP CONDITION HANDLER FOR any_other THAT RESULTS IN A CALL TO default_error_handler_$wall_ignore_pi

- THUS, A "CONDITION WALL" IS SET UP BETWEEN PROGRAMS RAISING CONDITIONS THAT HAVE NO HANDLERS FOR THEM & PROGRAMS RUN AT A NEW COMMAND LEVEL THEREAFTER

- THE WALL IS TRANSPARENT TO THE 'program_interrupt' AND 'finish' CONDITIONS

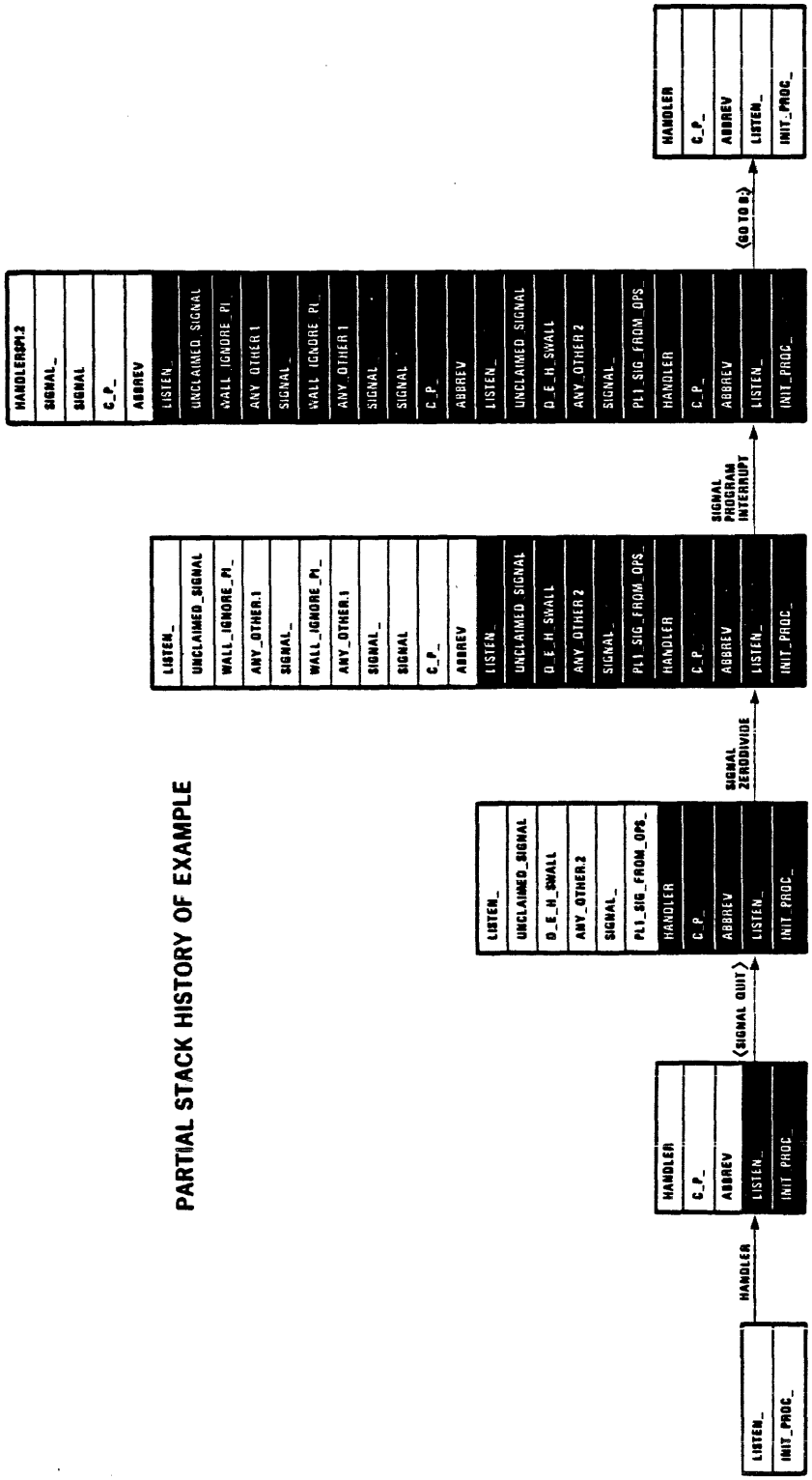
- THE PSEUDO CODE FOR program_interrupt IS AS FOLLOWS:

```
program_interrupt: pi: proc;

dcl program_interrupt condition;
dcl signal_ entry options (variable);
dcl start entry options (variable);

call signal_ ("program_interrupt", ...);
if handler_was_found
then call start;
else call com_err_ (..., "program_interrupt", "There is no suspended
                    invocation of a subsystem that supports the use of
                    this command.");

end /* program_interrupt */;
```

'program_interrupt' CONDITION

- EXAMPLE DEMONSTRATING THAT 'program_interrupt' "PENETRATES THE WALL"

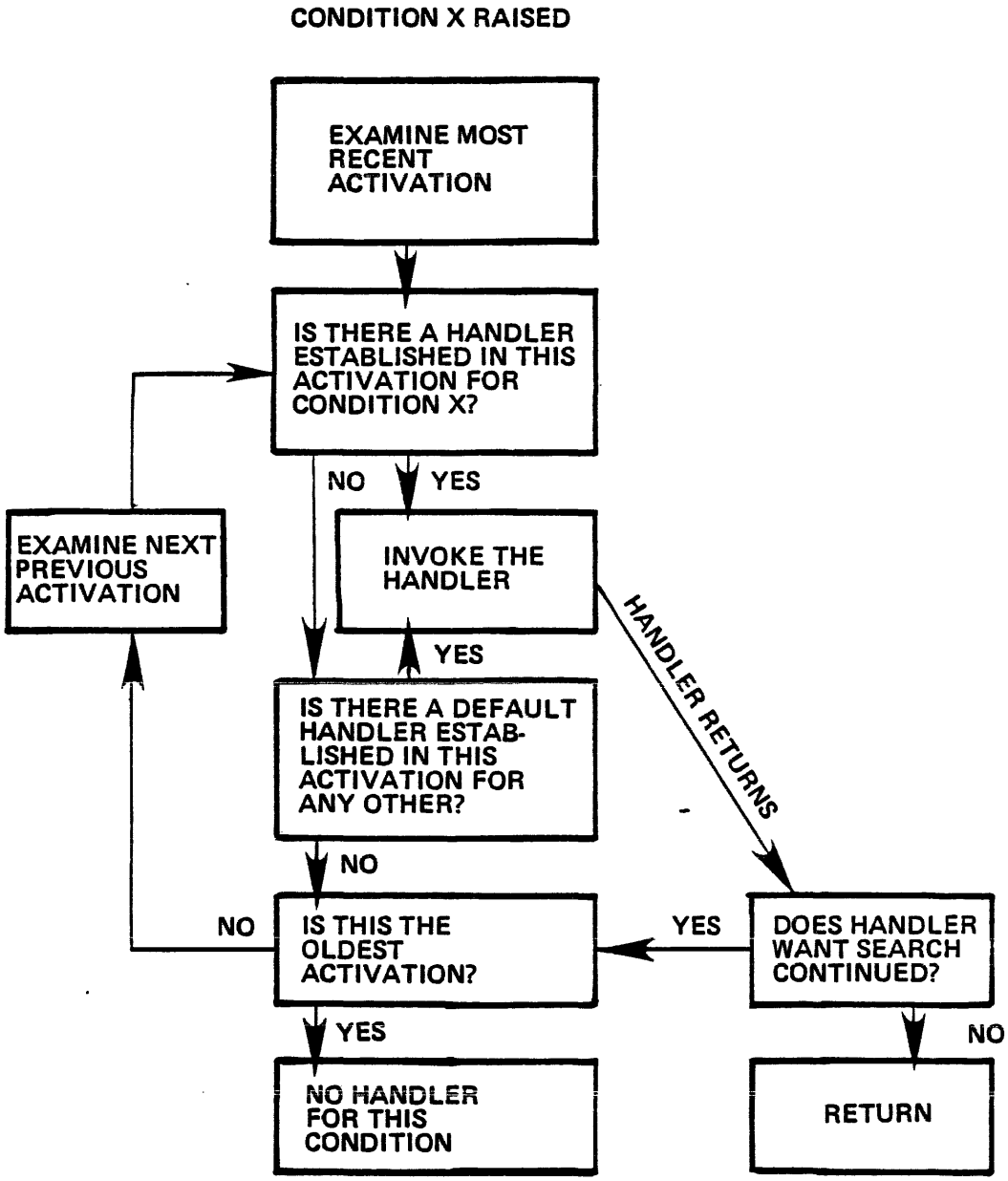
```
handler: proc;

dcl (program_interrupt,
     quit,
     zerodivide)        condition;
dcl  sysprint           file;

    on zerodivide go to A;
    on program_interrupt go to B;

    signal quit;

 A: put skip list ("ZERODIVIDE HAPPENED");
    put skip;
 B: put skip list ("PROGRAM INTERRUPT HAPPENED");
    put skip;

end /* handler */;


r 14:52 0.153 2

! handler
  QUIT
  r 14:52 0.265 3 level 2

! signal zerodivide

  Error:  Attempt to divide by zero at signal$|1101
  (>system_library_standard>bound_command_env_)
  system handler for error returns to command level
  r 14:52 0.524 20 level 3

! signal program_interrupt

  PROGRAM INTERRUPT HAPPENED
  r 14:52 0.221 7
```


PARTIAL STACK HISTORY OF EXAMPLE
LISTEN_
INIT_PROC_
HANDLER
C_P_
ABBREV
LISTEN_
INIT_PROC_
⟨SIGNAL QUIT⟩
LISTEN_
UNCLAIMED_SIGNAL
O_E_H_SWALL
ANY_OTHER.2
SIGNAL_
PL1_SIG_FROM_OPS_
HANDLER
C_P_
ABBREV
LISTEN_
INIT_PROC_
SIGNAL ZERODIVIDE
LISTEN_
UNCLAIMED_SIGNAL
WALL_IGNORE_PI_
ANY_OTHER.1
SIGNAL_
WALL_IGNORE_PI_
ANY_OTHER.1
SIGNAL_
SIGNAL
C_P_
ABBREV
LISTEN_
UNCLAIMED_SIGNAL
O_E_H_SWALL
ANY_OTHER.2
SIGNAL_
PL1_SIG_FROM_OPS_
HANDLER
C_P_
ABBREV
LISTEN_
INIT_PROC_
SIGNAL PROGRAM INTERRUPT
HANDLER/PI.2
SIGNAL_
SIGNAL
C_P_
ABBREV
LISTEN_
UNCLAIMED_SIGNAL
WALL_IGNORE_PI_
ANY_OTHER.1
SIGNAL_
WALL_IGNORE_PI_
ANY_OTHER.1
SIGNAL_
SIGNAL
C_P_
ABBREV
LISTEN_
UNCLAIMED_SIGNAL
O_E_H_SWALL
ANY_OTHER.2
SIGNAL_
PL1_SIG_FROM_OPS_
HANDLER
C_P_
ABBREV
LISTEN_
INIT_PROC_
⟨GO TO P.⟩
HANDLER
C_P_
ABBREV
LISTEN_
INIT_PROC_

'program interrupt' CONDITION

- NOTE: 'any_other' CONDITION HANDLERS SHOULD PASS ON THE 'program_interrupt' CONDITION (SEE continue_to_signal_ AND find_condition_info_)

SUMMARY OF CONDITION HANDLING MECHANISM



REVIEW OF PL/I DEFINED CONDITIONS

	Default Error Handler Signals Error	Undefined if hit End of On Unit	Can be Enabled/ Disabled	Disabled by Default
area	X	X		
error	(X)	X		
storage	X	X		
fixedoverflow	X	X	X	
overflow	X	X	X	
size	X	X	X	X
stringrange	X	X	X	X
subscriptrange	X	X	X	X
zerodivide	X	X	X	
conversion	X		X	
endfile	X			
key	X			
record	X			
transmit	X			
undefinedfile	X			
underflow			X	
stringsize			X	X
name				
endpage				
finish				

NOTE THAT THE 'size' CONDITION IS ENABLED DURING PL/I I/O (pl1_signal_from_ops), AND CONSEQUENTLY, A PL/I PROGRAM WHICH IS EXECUTING 'put' STATEMENTS TO THE 'sysprint' FILE MAY CAUSE 'size' CONDITIONS TO BE SIGNALLED EVEN THOUGH THE CONDITION IS NOT ENABLED IN THE PROGRAM ITSELF

REVIEW OF PL/I DEFINED CONDITIONS

- CONDITIONS IN THE PRECEDING TABLE WERE COVERED IN EARLIER COURSES, HOWEVER, THE 'finish', 'area' AND 'storage' CONDITIONS ARE COVERED BELOW SINCE THEY ARE NOT USUALLY FULLY UNDERSTOOD IN AN INTRODUCTORY COURSE

  - 'finish' CONDITION

    - THE FINISH CONDITION IS SIGNALLED JUST PRIOR TO RUN UNIT OR PROCESS TERMINATION

    - IT IS SIGNALLED BY A STOP STATEMENT OR BY COMMANDS SUCH AS 'stop_run', 'logout' AND 'new_proc'

    - IT BEHAVES JUST LIKE 'program_interrupt' IN THAT IT "PENETRATES THE WALL"

    - ALL CONDITION HANDLERS, WHETHER THEY HANDLE 'finish' OR NOT, SHOULD PASS THIS CONDITION ON (BY CALLING continue_to_signal_) SO THAT ALL PROGRAMS WILL BE NOTIFIED OF THE IMPENDING PROCESS, OR RUN UNIT, DESTRUCTION

REVIEW OF PL/I DEFINED CONDITIONS

- 'area' CONDITION
  - AN ATTEMPT HAS BEEN MADE TO ALLOCATE STORAGE IN A PL/I 'area' VARIABLE WHICH DOES NOT HAVE SUFFICIENT STORAGE FOR THE ATTEMPTED ALLOCATION
  - PRINTS A MESSAGE AND SIGNALS THE ERROR CONDITION
  - EXAMPLE

```
dcl (p,q,r) ptr;
dcl (A,B) (1000) fixed bin based;
dcl  C area(2000) static;
dcl  d float bin based;
      . . .
     allocate A set(p) in(C);
      . . .
     allocate d set(q) in(C);
      . . .
     allocate B set(r) in(C);

  /* causes 'area' condition (unless intervening
     'free' statements were executed)          */
```

- 'storage' CONDITION
  - AN ATTEMPT HAS BEEN MADE TO GROW A STACK SEGMENT PAST ITS MAXIMUM LENGTH
  - GENERALLY OCCURS AS A RESULT OF ATTEMPTING TO GENERATE A LARGE AMOUNT OF 'automatic' STORAGE, OR AS A RESULT OF A RUNAWAY RECURSIVE PROCEDURE
  - IS ALSO SIGNALLED IF A PL/I PROGRAM OVERFLOWS THE SYSTEM FREE STORAGE AREA

SOME SYSTEM-DEFINED CONDITIONS

- THE MULTICS SYSTEM HAS DEFINED SOME CONDITIONS OF ITS OWN

- SOME OF THE USEFUL SYSTEM-DEFINED (NON-PL/I) CONDITIONS ARE LISTED BELOW:

  - active_function_error, command_error

    - ARE SIGNALLED BY THE active_fnc_err_ AND com_err_ SUBROUTINES RESPECTIVELY

    - DEFAULT HANDLER FOR command_error PRINTS A MESSAGE AND RETURNS

    - DEFAULT HANDLER FOR active_function_error PRINTS AN ERROR MESSAGE AND RETURNS TO A NEW COMMAND LEVEL

  - cleanup

    - SIGNALLED TO THOSE PROCEDURES OWNING STACK FRAMES TO BE DISCARDED AS A RESULT OF A NON-LOCAL TRANSFER

    - THIS IS A VERY ATYPICAL USE OF THE CONDITION MECHANISM, SINCE 'cleanup' IS SIGNALLED IN EVERY FRAME BETWEEN THE CURRENT STACK FRAME AND THE FRAME CONTAINING THE TARGET OF THE NON-LOCAL TRANSFER

    - TYPE OF THING USUALLY DONE IN A 'cleanup' HANDLER

      - CLOSE FILES WHICH HAD BEEN OPENED IN THAT ACTIVATION BLOCK

      - FREE ALLOCATED 'controlled' OR 'based' VARIABLES

      - REINITIALIZE STATIC VARIABLES

      - SHOULD NOT DO A NON-LOCAL 'goto'

        - THIS WOULD INTERFERE WITH THE ONE ALREADY IN PROGRESS

SOME SYSTEM-DEFINED CONDITIONS

- fault_tag_1
  - SIGNALLED WHEN AN ATTEMPT IS MADE TO ACCESS THROUGH AN UNINITIALIZED POINTER OR A POINTER CONTAINING INVALID DATA
- illegal_opcode, illegal_procedure
  - SIGNALLED WHEN AN ATTEMPT IS MADE TO EXECUTE AN INVALID OR PRIVILEGED MACHINE INSTRUCTION
- linkage_error
  - SIGNALLED WHEN THE DYNAMIC LINKING MECHANISM OF MULTICS CAN NOT LOCATE AN EXTERNAL OBJECT
- lockup
  - SIGNALLED WHEN A PROGRAM IS EXECUTING A TIGHT LOOP OF CODE FOR TOO LONG A TIME
- null_pointer
  - SIGNALLED WHEN AN ATTEMPT IS MADE TO USE AN INVALID (NULL) POINTER
- out_of_bounds
  - SIGNALLED WHEN AN ATTEMPT IS MADE TO REFER TO A LOCATION BEYOND THE CURRENT LENGTH OF A SEGMENT

SOME SYSTEM-DEFINED CONDITIONS

- program_interrupt
  - SIGNALLED WHEN THE USER HAS ISSUED THE 'program_interrupt' COMMAND
- quit
  - SIGNALLED WHEN THE USER HITS THE 'break' OR 'attention' KEY ON HIS/HER TERMINAL (THE DEFAULT HANDLER PRINTS THE WORD "QUIT" ON THE USER'S TERMINAL, ABORTS THE PROGRAM, AND ESTABLISHES A NEW COMMAND LEVEL)
  - IN GENERAL, USER PROGRAMS SHOULD NOT HANDLE THE 'quit' CONDITION
- record_quota_overflow
  - SIGNALLED WHEN A USER ATTEMPTS TO ALLOCATE A RECORD IN SECONDARY STORAGE WHICH WILL OVERFLOW HIS/HER ALLOTTED LIMIT
- seg_fault_error
  - SIGNALLED WHEN AN ATTEMPT IS MADE TO USE A POINTER WITH AN INVALID SEGMENT NUMBER, AND CAN BE CAUSED BY:
    - THE DELETION OR TERMINATION OF A SEGMENT AFTER THE POINTER IS INITIALIZED
    - THE POINTER IS NOT INITIALIZED IN THE CURRENT PROCESS
    - THE USER HAS NO ACCESS TO THE SEGMENT

SOME SYSTEM-DEFINED CONDITIONS

YOU ARE NOW READY FOR WORKSHOP #4

TOPIC VII

The Multics Input/Output System

OBJECTIVES:

Upon completion of this topic, students should be able to:

1. Define the following terms:

   I/O switch

   I/O module

   stream I/O

   record sequential I/O

   record blocked I/O

   indexed I/O

2. List the more popular I/O modules.

3. List the steps required to perform I/O.

4. Describe an I/O control block (IOCB).

CHARACTERISTICS

- THE MULTICS INPUT/OUTPUT SYSTEM IS A FLEXIBLE, GENERALIZED I/O SYSTEM CAPABLE OF SUPPORTING SEVERAL PROTOCOLS OF DATA TRANSMISSION TO A FULL COMPLEMENT OF FILES AND DEVICES

- I/O SYSTEM BASIC CHARACTERISTICS:

  - LOGICAL INPUT/OUTPUT REQUESTS ARE USED RATHER THAN DEVICE-SPECIFIC PHYSICAL REQUESTS

  - DEVICE INDEPENDENCE IS ACHIEVED VIA THE MULTICS I/O SWITCH MECHANISM

  - UNFAMILIAR OR NEW DEVICES CAN BE ADDRESSED VIA THE IMPLEMENTATION OF SITE-PREPARED INPUT/OUTPUT INTERFACE MODULES

THE MULTICS I/O MECHANISM

- THE I/O MECHANISM USES THE FOLLOWING CONSTRUCTS:

  - SWITCH, SWITCHNAME

    - A SWITCH IS A LOGICAL CONSTRUCT USED TO DESIGNATE THE TARGET OF AN INPUT OR OUTPUT REQUEST

    - ASSOCIATED WITH AN I/O SWITCH IS A "SWITCHNAME"

    - ALL I/O REQUESTS ARE DIRECTED TO A "SWITCH" WHICH IS "ATTACHED" BY A DEVICE-DEPENDENT PROGRAM, CALLED AN I/O MODULE, TO A PARTICULAR DEVICE OR FILE

    - THE SUPPORTING DATA STRUCTURE OF A SWITCH IS AN I/O CONTROL BLOCK (IOCB)

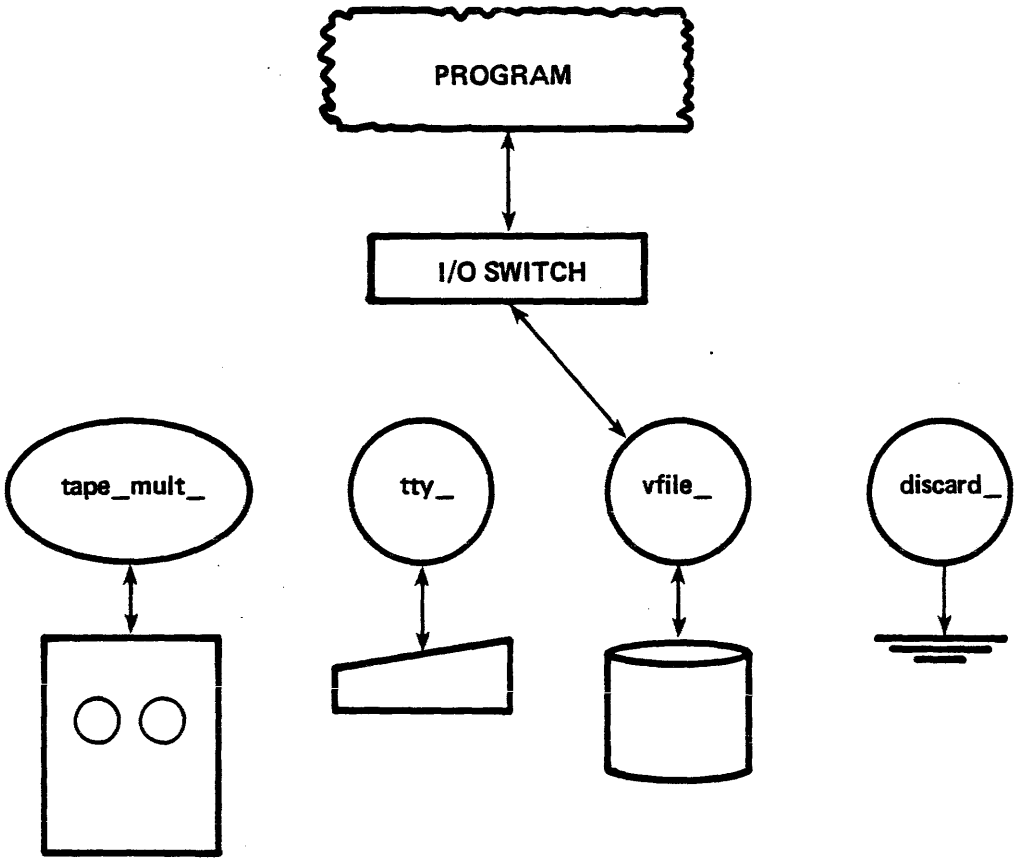
  - INPUT/OUTPUT MODULE

    - A DEVICE-DEPENDENT COMMUNICATION MODULE WHICH ACTS AS THE INTERFACE BETWEEN THE USER'S LOGICAL I/O REQUESTS AND THE HARDWARE-LEVEL I/O SYSTEM

    - TRANSLATES THE USER'S LOGICAL REQUESTS INTO THE PHYSICAL REQUESTS APPROPRIATE TO THE TYPE OF DEVICE OR FILE FOR WHICH IT WAS WRITTEN

    - SYSTEM STANDARD MODULES SUPPORT I/O TO/FROM BASIC DEVICES (TAPE, REMOVABLE DISK, TERMINAL DEVICES, CARD READERS, ETC.) AND FILES (SEGMENTS IN THE VIRTUAL MEMORY)

THE MULTICS I/O MECHANISM



THE MULTICS I/O MECHANISM

PROTOCOLS SUPPORTED

- FOUR BASIC I/O PROTOCOLS (FILE STRUCTURES) SUPPORTED

  - THE TYPE OF PROTOCOL BEING USED LIMITS THE REQUESTS THAT CAN BE SATISFIED

  - CERTAIN I/O MODULES SUPPORT ONLY ONE PROTOCOL, SOME I/O MODULES SUPPORT ALL THE PROTOCOLS

  - THEY ARE:

    - 1) STREAM INPUT/OUTPUT

      - A STREAM FILE IS A SEQUENCE OF ASCII CHARACTERS, SEPARATED BY NEWLINE AND NEWPAGE CHARACTERS

      - OFTEN CALLED AN "UNSTRUCTURED" FILE

      - EXAMPLES: TERMINAL DIALOG, TEXT EDITOR CREATED SEGMENTS, TAPES WRITTEN VIA tape_mult_

    - 2) RECORD SEQUENTIAL INPUT/OUTPUT

      - A "STRUCTURED" FILE OF VARIABLE LENGTH RECORDS, EACH RECORD REPRESENTING ONE STRUCTURE

      - A RECORD FILE MAY BE ACCESSED IN "SEQUENTIAL" PROTOCOL, WHICH MEANS THAT THE CURRENT RECORD AND NEXT RECORD ARE WELL-DEFINED

      - EXAMPLES: TAPES WRITTEN VIA tape_ibm_ OR tape_ansi_, CERTAIN VIRTUAL MEMORY SEGMENTS

PROTOCOLS SUPPORTED

- 3) RECORD BLOCKED INPUT/OUTPUT
  - A RECORD FILE MAY BE CREATED IN LOGICAL BLOCKS, THUS ALLOWING I/O TO BE DONE A BLOCK AT A TIME
  - BLOCK SIZE IS FIXED
  - A BLOCK CONTAINS
    - ONE RECORD (WITH POTENTIAL WASTED SPACE) IF IN A VIRTUAL MEMORY FILE
    - ONE OR MORE RECORDS IF ON ANSI OR IBM TAPE
  - SPECIFY BLOCKED MODE AT ATTACH TIME

- 4) INDEXED INPUT/OUTPUT
  - AN INDEXED FILE IS A "KEYED" FILE, IMPLEMENTED AS A MULTI-SEGMENT FILE WITH ONE (OR MORE) COMPONENTS HOLDING THE "KEY VALUES", AND ONE (OR MORE) COMPONENTS HOLDING THE "DATA RECORDS"
  - AN INDEXED FILE MAY BE ACCESSED IN EITHER "KEYED SEQUENTIAL" MODE, OR "KEYED DIRECT" MODE
  - MUST BE IN THE VIRTUAL MEMORY
  - EXAMPLE: "RELATIONS" IN A MRDS DATABASE

- PL/I DEDUCES THE PROTOCOL BY EXAMINING LANGUAGE I/O STATEMENTS AND/OR THE ATTACH DESCRIPTION

THE MORE POPULAR I/O MODULES

- SOME OF THE SYSTEM STANDARD I/O MODULES, THEIR FUNCTIONS, AND THE PROTOCOLS SUPPORTED ARE:

NAME	FUNCTION	PROTOCOLS SUPPORTED
1) vfile_	I/O TO/FROM SEGMENTS IN THE VIRTUAL MEMORY	ALL
2) tty_	I/O TO/FROM TERMINAL DEVICES	STREAM
3) discard_	OUTPUT SINK	ALL
4) syn_	ALLOWS ONE SWITCH TO SERVE AS A SYNONYM FOR ANOTHER SWITCH	ALL
5) rdisk_	I/O TO/FROM REMOVABLE, NON-MULTICS DISK PACKS	SEQUENTIAL, KEYED, OR BLOCKED
6) record_stream_	ALLOWS RECORD I/O OPERATIONS TO BE DIRECTED TO A STREAM FILE AND VICE VERSA	STREAM <-> SEQUENTIAL
7) tape_mult_	I/O TO/FROM A MULTICS FORMAT TAPE	STREAM
8) tape_ibm_ tape_ansi_	I/O TO/FROM A TAPE FILE IN IBM OR ANSI FORMAT	SEQUENTIAL, BLOCKED
9) tape_nstd_	I/O TO/FROM TAPES IN NON-STANDARD OR UNKNOWN FORMATS	SEQUENTIAL
10) bisync_	I/O ACROSS A BINARY SYNCHRONOUS COMMUNICATIONS CHANNEL	STREAM
11) audit_	INTERCEPTS I/O ACTIVITY ON A GIVEN SWITCH, ALLOWING LOGGING AND EDITING OF DATA	STREAM

THE MULTICS I/O MECHANISM

PERFORMING MULTICS I/O

- STEPS REQUIRED TO PERFORM I/O

  1) THE SPECIFIED SWITCH MUST BE "ATTACHED" (INITIALIZED) BY A SPECIFIED I/O MODULE TO SOME TARGET DEVICE OR FILE (SUBSEQUENT REQUESTS DIRECTED TO THE SWITCHNAME OPERATE VIA THE I/O MODULE ON THE TARGET DEVICE OR FILE)

  2) THE SWITCH MUST BE "OPENED" IN A MODE COMPATIBLE WITH THE TYPE OF DEVICE OR FILE BEING MANIPULATED

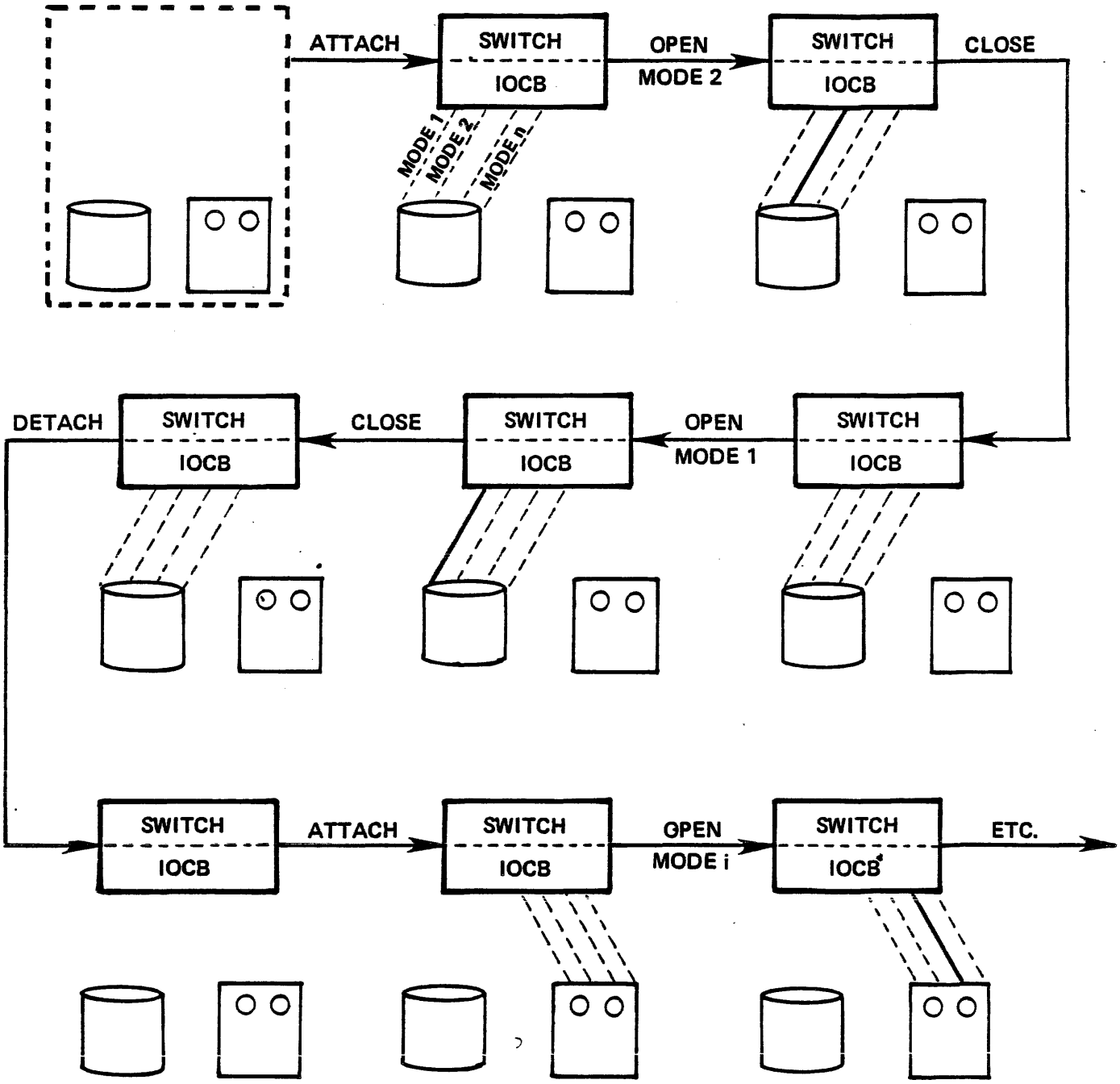
  3) INPUT/OUTPUT OPERATIONS CAN NOW BE DIRECTED TO THE SWITCH (OPERATIONS MUST BE CONSISTENT WITH THE ATTACHMENT AND OPENING MODE OF THE SWITCH)

  4) THE SWITCH MUST BE "CLOSED" LEAVING THE SWITCH IN THE STATE IT WAS PRIOR TO THE "OPENING" (THAT IS, IT MAY NOW BE OPENED WITH A DIFFERENT MODE)

  5) THE SPECIFIED SWITCH MUST BE "DETACHED" BREAKING THE ASSOCIATION BETWEEN THE SWITCHNAME AND THE I/O MODULE AND TARGET (HENCE, THE SWITCH MAY BE ATTACHED IN A NEW WAY)

THE MULTICS I/O MECHANISM

PERFORMING MULTICS I/O



THE MULTICS I/O MECHANISM

PERFORMING MULTICS I/O

- ALL I/O OPERATIONS CAN BE PERFORMED AT THREE BASIC LEVELS:
  - LANGUAGE LEVEL - 'open', 'close', 'get', 'read', 'put', 'write'
  - COMMAND LEVEL - THE 'io_call' COMMAND
  - SUBROUTINE LEVEL - THE 'iox_' SUBROUTINE
  - EXAMPLES (THE FOLLOWING ARE EQUIVALENT):
    - PL/I

```
open file (x) title ("vfile_ user_file") stream output;
```

    - COMMAND LEVEL

```
io_call attach x vfile_ user_file
io_call open x stream_output
```

    - SUBROUTINE LEVEL

```
call iox_$attach_name ("x", iocb_ptr, "vfile_ user_file",
                       ref_ptr, code);
call iox_$open (iocb_ptr, 2, "0"b, code);
```

  - LANGUAGE VS. I/O SYSTEM

PL/I STATEMENT	EQUIVALENT I/O CALLS
open	attach open
close	close detach

THE MULTICS I/O MECHANISM

PERFORMING MULTICS I/O

- THE ATTACHMENT AND DETACHMENT OF A SWITCH CAN BE PERFORMED EITHER EXTERNALLY TO A PROGRAM OR INTERNALLY BY THE PROGRAM ITSELF

  - IF THE SWITCH IS ATTACHED EXTERNALLY, THE PROGRAM RECOGNIZES THIS ATTACHMENT, HONORS THIS PRIOR ATTACHMENT, AND IGNORES THE SPECIFIED INTERNAL ATTACH DESCRIPTION (THUS YIELDING DEVICE INDEPENDENCE)

  - IF THE SWITCH HAS NOT BEEN ATTACHED EXTERNALLY, THE ATTACH DESCRIPTION SUPPLIED BY THE PROGRAM (EITHER EXPLICITLY OR IMPLICITLY) WILL BE USED TO ATTACH THE SWITCH

  - IF THE SWITCH IS ATTACHED EXTERNALLY, IT MUST BE DETACHED EXTERNALLY

    - IF THE SWITCH IS ATTACHED INTERNALLY BY EXECUTION OF THE 'open' STATEMENT, IT WILL BE DETACHED BY EXECUTION OF THE 'close' STATEMENT

- THE ABOVE STATEMENTS SIMILARLY APPLY TO THE OPEN AND CLOSE OPERATIONS

- EXAMPLE

```
x: proc;

dcl line char(80);
dcl (abc, xyz) file;
dcl i;

open file (abc) input;
open file (xyz) output;

do i = 1 to 50;
   get file (abc) list (line);
   put file (xyz) list (line);
end;

close file (abc), file (xyz);

end /* x */;
```

- TO HAVE OUTPUT SENT TO TERMINAL INSTEAD OF FILE xyz USER COULD TYPE THE FOLLOWING:

```
!   io_call attach xyz syn_ user_output

!   x
        .....
        .....
        .....

!   io_call detach xyz
```

THE 'iox_' SUBROUTINE

- iox_ IS THE USER-RING INTERFACE TO THE MULTICS INPUT/OUTPUT SYSTEM

  - ALL I/O OPERATIONS ISSUED AT THE USER-RING LEVEL (WHETHER FROM COMMAND LEVEL, LANGUAGE LEVEL, OR DIRECT iox_ CALL) RESULT IN A CALL TO iox_

  - iox_ PROVIDES ENTRY POINTS FOR ALL INPUT/OUTPUT OPERATIONS

  - EVERY iox_ ENTRY POINT REQUIRES AN ARGUMENT DENOTING THE PARTICULAR I/O SWITCH (ACTUALLY THE IOCB) INVOLVED IN THE OPERATION

  - IF AN ENTRY POINT REQUIRES THE I/O SWITCH TO BE OPEN, AND IF IT IS NOT, THE CODE 'error_table_$not_open' IS RETURNED

  - IF THE I/O SWITCH IS OPEN, BUT THE OPERATION IS NOT ALLOWED FOR THAT OPENING MODE, THE CODE 'error_table_$no_operation' IS RETURNED

THE 'iox ' SUBROUTINE

- THE MAJOR ENTRY POINTS OF iox_ CAN BE CLASSIFIED AS FOLLOWS:

  - ATTACHING/DETACHING
    - iox_$attach_name
    - iox_$attach_ptr
    - iox_$detach_iocb
    - iox_$destroy_iocb
    - iox_$find_iocb
    - iox_$look_iocb
    - iox_$move_attach
  - OPENING/CLOSING
    - iox_$open
    - iox_$close
  - STREAM I/O REQUESTS
    - iox_$get_chars
    - iox_$get_line
    - iox_$put_chars

THE 'iox_' SUBROUTINE

- RECORD I/O REQUESTS
  - iox_$delete_record
  - iox_$read_key
  - iox_$read_length
  - iox_$read_record
  - iox_$rewrite_record
  - iox_$seek_key
  - iox_$write_record
- CONTROL REQUESTS
  - iox_$control
  - iox_$modes
  - iox_$position

I/O CONTROL BLOCKS

- WHAT IS AN I/O CONTROL BLOCK (IOCB)?
  - EVERY SWITCHNAME HAS ASSOCIATED WITH IT AN 'IOCB'
  - AN 'IOCB' IS A STANDARD DATA STRUCTURE
  - IT IS THE PHYSICAL REALIZATION OF A SWITCH
  - THEY ARE FOUND IN THE USER'S PROCESS DIRECTORY
  - AN 'IOCB' IS CREATED BY iox_ WHEN A SWITCHNAME IS USED IN AN "ATTACH STATEMENT" OR "ATTACH COMMAND" FOR THE FIRST TIME IN A PROCESS
    - IF THE SAME SWITCHNAME IS USED LATER IN THE PROCESS, THE SAME 'IOCB' IS REUSED
    - THUS THERE IS A ONE TO ONE MAPPING BETWEEN SWITCHNAMES AND IOCB'S
  - ONCE AN 'IOCB' IS CREATED, IT LIVES THROUGHOUT THE PROCESS (UNLESS EXPLICITLY DELETED)

```
/* BEGIN INCLUDE FILE ..... iocb.incl.pl1 .....
                              13 Feb 1975, M. Asherman */
/* Modified 11/29/82 by S. Krupp to add new entries and
                     to change version number to IOX2. */
/* format: style2 */

dcl   1 iocb                    aligned based,
          /* I/O control block. */
        2 version               character (4) aligned,
          /* IOX2 */
        2 name                  char (32),
          /* I/O name of this block. */
        2 actual_iocb_ptr       ptr,
          /* IOCB ultimately SYNed to. */
        2 attach_descrip_ptr    ptr,
          /* Ptr to printable attach description. */
        2 attach_data_ptr       ptr,
          /* Ptr to attach data structure. */
        2 open_descrip_ptr      ptr,
          /* Ptr to printable open description. */
        2 open_data_ptr         ptr,
          /* Ptr to open data structure (old SDB). */
        2 reserved              bit (72),
          /* Reserved for future use. */
        2 detach_iocb           entry (ptr, fixed (35)),
          /* detach_iocb(p,s) */
        2 open                  entry (ptr, fixed, bit (1) aligned,
                                       fixed (35)),
          /* open(p,mode,not_used,s) */
        2 close                 entry (ptr, fixed (35)),
          /* close(p,s) */
        2 get_line              entry (ptr, ptr, fixed (21),
                                       fixed (21), fixed (35)),
          /* get_line(p,bufptr,buflen,actlen,s) */
        2 get_chars             entry (ptr, ptr, fixed (21),
                                       fixed (21), fixed (35)),
          /* get_chars(p,bufptr,buflen,actlen,s) */
        2 put_chars             entry (ptr, ptr, fixed (21),
                                       fixed (35)),
          /* put_chars(p,bufptr,buflen,s) */
        2 modes                 entry (ptr, char (*), char (*),
                                       fixed (35)),
          /* modes(p,newmode,oldmode,s) */
        2 position              entry (ptr, fixed, fixed (21),
                                       fixed (35)),
          /* position(p,u1,u2,s) */
        2 control               entry (ptr, char (*), ptr,
                                       fixed (35)),
          /* control(p,order,infptr,s) */
        2 read_record           entry (ptr, ptr, fixed (21),
                                       fixed (21), fixed (35)),
          /* read_record(p,bufptr,buflen,actlen,s) */
        2 write_record          entry (ptr, ptr, fixed (21),
```

```
                                        fixed (35)),
          /* write_record(p,bufptr,buflen,s) */
        2 rewrite_record    entry (ptr, ptr, fixed (21),
                                        fixed (35)),
          /* rewrite_record(p,bufptr,buflen,s) */
        2 delete_record     entry (ptr, fixed (35)),
          /* delete_record(p,s) */
        2 seek_key          entry (ptr, char (256) varying,
                                        fixed (21), fixed (35)),
          /* seek_key(p,key,len,s) */
        2 read_key          entry (ptr, char (256) varying,
                                        fixed (21), fixed (35)),
          /* read_key(p,key,len,s) */
        2 read_length       entry (ptr, fixed (21), fixed (35)),
          /* read_length(p,len,s) */
        2 open_file         entry (ptr, fixed bin, char (*),
                                        bit (1) aligned, fixed bin (35)),
          /* open_file(p,mode,desc,not_used,s) */
        2 close_file        entry (ptr, char (*), fixed bin (35)),
          /* close_file(p,desc,s) */
        2 detach            entry (ptr, char (*), fixed bin (35));
          /* detach(p,desc,s) */

declare iox_$iocb_version_sentinel
                    character (4) aligned external static;

/* END INCLUDE FILE ..... iocb.incl.pl1 ..... */

dcl 1 attach_descrip based aligned,
      2 length      fixed bin (17),
      2 string      char (0 refer (attach_descrip.length));
```

I/O CONTROL BLOCKS

- AN ATTACH DESCRIPTION IS A CHARACTER STRING CONVEYING THE FOLLOWING INFORMATION:

  - MODULE NAME

  - MODULE-SPECIFIC ARGUMENTS, SUCH AS:

    - PATHNAME (vfile_)

    - CHANNEL NAME (tty_, bisync_)

    - VOLUME ID (tape_ibm_, tape_ansi_, tape_mult_, tape_nstd_)

    - DISK_DRIVE_ID AND PACK_ID (rdisk_)

    - SWITCHNAME (syn_, record_stream_)

  - MODULE-SPECIFIC CONTROL ARGUMENTS, SUCH AS:

    - -extend (vfile_, tape_ibm_, tape_ansi_)

    - -density (tape_ibm_, tape_ansi_, tape_mult_)

    - -block (tape_ibm_, tape_ansi_)

    - -blocked (vfile_)

  - COMPLETE DESCRIPTIONS OF THE I/O MODULES AND THE ARGUMENTS SPECIFIED AT ATTACH TIME ARE IN Multics Subroutines & I/O Modules (AG93)

I/O CONTROL BLOCKS

- THE PRINCIPAL COMPONENTS OF AN 'IOCB' ARE 'pointer' VARIABLES AND 'entry' VARIABLES

- THERE IS ONE 'entry' VARIABLE FOR EACH I/O OPERATION, WITH THE EXCEPTION OF THE ATTACH OPERATION

- TO PERFORM AN I/O OPERATION THROUGH THE SWITCH, THE APPROPRIATE ENTRY VALUE IN THE CORRESPONDING 'IOCB' IS CALLED

  FOR EXAMPLE:

```
call iox_$put_chars(iocb_ptr,.....);
```

  CAN BE THOUGHT OF AS:

```
call iocb_ptr->iocb.put_chars(.....);
```

I/O CONTROL BLOCKS

- WHEN iox_$attach_name IS CALLED IT:
  - CREATES/LOCATES THE 'IOCB' ASSOCIATED WITH THAT SWITCHNAME
  - INITIALIZES SOME OF THE ELEMENTS IN THE 'IOCB' STRUCTURE
  - CALLS <module_name>$<module_name>attach
    - THUS THERE NEED BE NO ENTRY FOR THE ATTACH OPERATION IN THE 'IOCB'
    - THIS ENTRY POINT IN THE I/O MODULE FINISHES THE INITIALIZATION OF THE 'IOCB'
  - FOR EXAMPLE, IF THE I/O MODULE INVOLVED IN THE ATTACHMENT WAS vfile_:
    - vfile_$vfile_attach IS CALLED
    - AFTER THE ATTACHMENT (INITIALIZATION) IS COMPLETE:
      - iocb.open CONTAINS THE ENTRY TO vfile_$open
      - iocb.close CONTAINS THE ENTRY iox_$err_not_open

I/O CONTROL BLOCKS

- AFTER THE ATTACHMENT OF THE SWITCH, EVERY I/O OPERATION ON THAT SWITCH REFERENCES THE CORRESPONDING 'IOCB' TO FIND THE ENTRY POINT AT WHICH TO START EXECUTION

  - ONE OF TWO ACTIONS MAY RESULT:

    - iox_ GENERATES AN ERROR MESSAGE (IF IT IS AN ILLEGAL OPERATION)

    - EXECUTION STARTS AT THE APPROPRIATE ENTRY POINT OF THE APPROPRIATE MODULE

      - THIS EXECUTION UPDATES THE 'IOCB', USUALLY REPLACING SOME ENTRY VALUES CAUSING ERROR MESSAGES WITH ENTRY VALUES INDICATING ENTRY POINTS IN THE MODULE (AND VISA VERSA)

      - EXAMPLE (IN THE ABOVE CASE):

IOCB MEMBER	BEFORE OPENING	AFTER OPENING
iocb.open	vfile_$open	iox_$err_not_closed
iocb.close	iox_$err_not_open	vfile_$close

- IT IS THE RESPONSIBILITY OF THE I/O MODULE TO MAINTAIN THE ACCURACY OF THE 'IOCB'

- ONLY THE iox_ ENTRY POINTS RESULTING IN ATTACHMENT OF A SWITCH REQUIRE THE MODULE AS AN INPUT ARGUMENT

  - AFTER THAT TIME, THE 'IOCB' "POINTS TO" THE APPROPRIATE ENTRY POINTS IN THE APPROPRIATE MODULE (THE USER NEED ONLY PROVIDE A POINTER TO THE 'IOCB')

I/O CONTROL BLOCKS

- IN VIEW OF THE ABOVE DISCUSSION OF IOCB'S AND SWITCHES, THE TERM "SWITCH" SHOULD MAKE MORE SENSE

  - A SWITCH/IOCB CAN BE THOUGHT OF AS A STRUCTURE CONTAINING TRANSFER VECTORS

YOU ARE NOW READY FOR WORKSHOP #5

TOPIC VIII

The iox_ Multics Subroutine

OBJECTIVES:

Upon completion of this topic, students should be able to:

1. Open and close I/O switches using iox_.

2. Read data from the user's terminal.

3. Display information on the user's terminal.

4. Read and write stream files.

5. Read and write sequential and keyed files.

INTRODUCTION TO USING iox_

- WHY USE iox_ RATHER THAN PL/I I/O STATEMENTS?
  - iox_ IS MORE EFFICIENT
    - WRITTEN IN alm
    - NUMBER OF MEMORY ACCESSES
      - iox_ ACCESSES 'IOCB' ONLY
      - PL/I STATEMENTS ACCESS 'FSB' (FILE STATE BLOCK) AND 'IOCB'
    - MORE POWERFUL
    - BETTER ERROR DETECTION
  - ACCEPTED CONVENTION FOR SYSTEM CODE

- WARNING: SHOULD NOT MIX iox_ AND PL/I I/O DUE TO INCONSISTENCIES (DIRECT CALLS TO iox_ DO NOT MAINTAIN 'FSB')

iox_ OPENING MODES

- iox_ OPENING MODES SUPPORTED AND THE iox_ OPERATIONS PERMITTED FOR EACH OPENING:

NO	NAME	I/O OPERATIONS PERMITTED
1	stream_input	get_line, get_chars, position
2	stream_output	put_chars
3	stream_input_output	1 + 2
4	sequential_input	read_record, read_length, position
5	sequential_output	write_record
6	sequential_input_output	4 + 5
7	sequential_update	4, rewrite_record, delete_record
8	keyed_sequential_input	read_record, read_length, position, seek_key, read_key
9	keyed_sequential_output	seek_key, write_record
10	keyed_sequential_update	8 + 9,rewrite_record,delete_record
11	direct_input	read_record, read_length, seek_key
12	direct_output	seek_key, write_record
13	direct_update	11 + 12,rewrite_record,delete_record

SEE >ldd>include>iox_modes.incl.pl1

- NOTE:

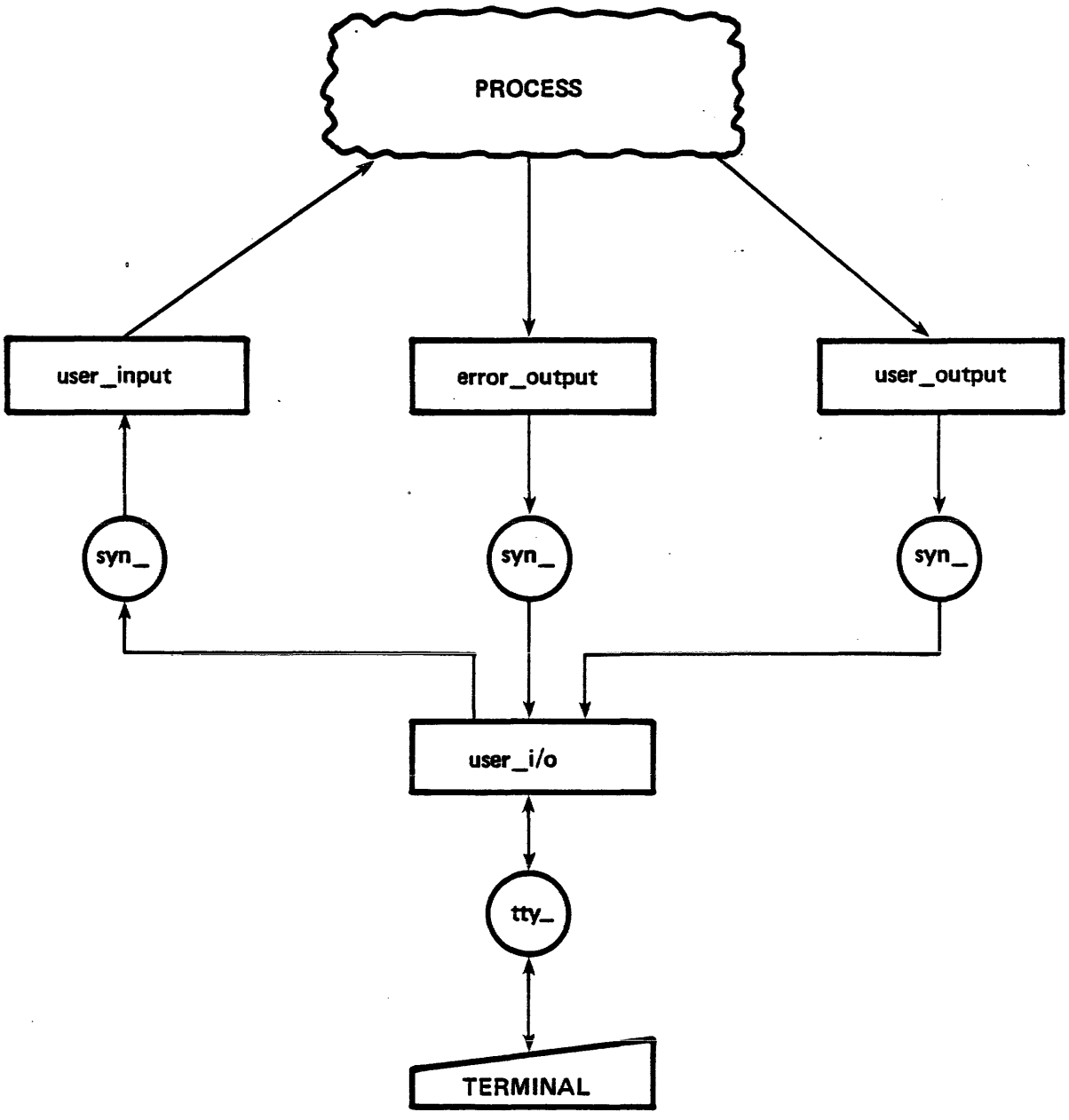
  - THE 'open', 'close', 'control', AND 'modes' OPERATIONS ARE PERMITTED WITH ANY OPENING MODE

  - THE ABOVE NUMBERS ARE USED IN CALLS TO iox_ TO SPECIFY OPENING MODES

  - THE LONG NAME (AS GIVEN ABOVE) IS USED WITH 'io_call'

  - PL/I SPECIFIES THE OPENING MODE IN THE FILE DESCRIPTION

STANDARD SWITCH ATTACHMENTS



STANDARD SWITCH ATTACHMENTS

- THE MULTICS STANDARD PROGRAMMING ENVIRONMENT MAKES USE OF FOUR SWITCHES WHICH ARE ATTACHED AND OPENED AS PART OF THE PROCESS CREATION CYCLE

- THE STANDARD ATTACHMENTS ARE:

```
user_i/o            tty_ -login_channel
          stream_input_output
user_input          syn_ user_i/o
user_output  syn_ user_i/o
error_output        syn_ user_i/o
```

- IN TERMS OF iox_, THESE SWITCHES ARE IDENTIFIED BY THE FOLLOWING DECLARATIONS:

```
dcl iox_$user_io external pointer;

dcl iox_$user_input external pointer;

dcl iox_$user_output external pointer;

dcl iox_$error_output external pointer;
```

EXAMPLE

```
call iox_$put_chars (iox_$user_output, buffer_ptr,
                     buffer_length, code);
```

iox ENTRY POINTS

- THERE ARE OVER 25 ENTRY POINTS FOR THE iox_ SUBROUTINE (SEVERAL ARE PRESENTED IN THE REMAINDER OF THIS TOPIC)

- THE FIRST 7 ENTRY POINTS:

  - ARE SUMMARIZED ON THE NEXT 2 PAGES

  - WILL BE STUDIED IN DETAIL BY REFERRING TO THE SUBROUTINES MANUAL

  - WILL BE USED IN WORKSHOP 6

  - REPRESENT SOME COMMONLY USED ENTRY POINTS THAT WOULD BE USED TO PROMPT A USER FOR A KEY AND THEN FIND THE CORRESPONDING RECORD IN A KEYED FILE

- THE OTHER ENTRY POINTS (STARTING ON PAGE 8-7) WILL BE COVERED IN MUCH LESS DETAIL

- SEVERAL OPERATIONS INVOLVE THE USE OF A BUFFER

  - A BUFFER IS A BLOCK OF STORAGE PROVIDED BY THE CALLER OF THE OPERATION AS THE TARGET FOR INPUT OR THE SOURCE FOR OUTPUT

  - A PTR TO THE BUFFER IS PASSED TO iox_ SUBROUTINES

iox_ ENTRY POINTS

- iox_$attach_name
  - ACCEPTS A SWITCHNAME
  - RETURNS A POINTER TO THE 'IOCB' FOR THE CORRESPONDING SWITCH
  - ATTACHES THE SWITCH IN ACCORDANCE WITH THE SUPPLIED ATTACH DESCRIPTION

- iox_$open
  - OPENING MODE IS SPECIFIED BY A NUMBER (SEE PAGE 8-2)

- iox_$get_line
  - THE NEWLINE CHARACTER SIGNIFIES THE END OF THE LINE
  - A CODE OF ZERO IS RETURNED ONLY IF A NEWLINE CHARACTER IS READ
  - THE NEWLINE ITSELF IS READ INTO THE BUFFER

iox_ ENTRY POINTS

- iox_$seek_key
  - THE NEXT RECORD POSITION AND CURRENT RECORD POSITION ARE SET TO THE RECORD WITH THE GIVEN KEY
  - USED BEFORE DOING A read, delete, rewrite, ETC.

- iox_$read_record
  - READS THE NEXT RECORD IN A STRUCTURED FILE
  - KEYED READS FIRST REQUIRE A CALL TO iox_$seek_key

- iox_$close

- iox_$detach_iocb
  - DOES NOT FREE THE IOCB'S STORAGE

iox_ ENTRY POINTS

- THE REST OF THIS TOPIC WILL SERVE AS AN OVERVIEW OF OTHER iox_ ENTRY POINTS

- iox_$attach_ptr

  - call iOx_$attach_ptr (iocb_ptr, atd, ref_ptr, code);

  - BEHAVES LIKE iox_$attach_name, EXCEPT iocb_ptr IS AN INPUT NOT AN OUTPUT VARIABLE

- iox_$find_iocb

  - call iox_$find_iocb (switchname, iocb_ptr, code);

  - GIVEN A SWITCHNAME, RETURNS A POINTER TO THE IOCB, BUT DOES NO ATTACHMENT (IF THE BLOCK DOES NOT ALREADY EXIST, IT IS CREATED)

  - iox_$find_iocb + iox_$attach_ptr = iox_$attach_name

- iox_$look_iocb

  - call iox_$look_iocb (switchname, iocb_ptr, code);

  - BEHAVES LIKE iox_$find_iocb, HOWEVER DOES NOT CREATE A BLOCK IF ONE DOES NOT ALREADY EXIST

iop_ ENTRY POINTS

- iox_$move_attach

  - call iox_$move_attach (iocb_ptr1, iocb_ptr2, code);

  - INCLUDED FOR COMPLETENESS (NOT FOR NOVICE USERS)

  - MOVES AN ATTACHMENT FROM ONE ATTACHED SWITCH TO ANOTHER DETACHED SWITCH

  - THE PERFECT EXAMPLE (FOR WHICH move_attach WAS WRITTEN) IS THE CASE OF file_output, IN WHICH A TEMPORARY SWITCH IS CREATED, THE CURRENT ATTACHMENT OF user_output IS MOVED TO THAT TEMPORARY SWITCH, AND THEN user_output IS ATTACHED TO THE OUTPUT FILE.

- iox_$destroy_iocb

  - call iox_$destroy_iocb (iocb_ptr, code);

  - FREES THE STORAGE USED BY A DETACHED CONTROL BLOCK

- iox_$get_chars

  - call iox_$get_chars (iocb_ptr, buff_ptr, n, n_read, code);

  - USER REQUESTS n BYTES (CHARACTERS) FROM A STREAM FILE OR DEVICE (ACTUALLY NUMBER READ IS n_read BYTES)

  - IF n = n_read THEN code = 0

  - IF n_read < n THEN code = error_table_$short_record

  - IF NEXT BYTE IS "END OF FILE" THEN code = error_table_$end_of_info (NOTE THAT THE 'endfile' CONDITION IS NOT SIGNALLED WHEN USING iox_)

  - READS NEWLINE CHARACTERS INTO BUFFER JUST LIKE ANY OTHER CHARACTER

  - IF n IS GREATER THAN THE SIZE OF THE RECEIVING BUFFER, OVERFLOW CHARACTERS WILL BE WRITTEN PAST THE END OF THE BUFFER, YIELDING POTENTIALLY DISASTROUS RESULTS

  - BUFFER OUGHT TO BE EXPLICITLY FLUSHED PRIOR TO CALL, BECAUSE JUST n_read CHARACTERS WILL BE OVERWRITTEN

    - ALTERNATIVE:

      ```
      dcl max_buff char(80) based (buff_ptr);
      dcl buff char (n_read) based (buff_ptr);
      ```

iox_ ENTRY POINTS

- iox_$put_chars

  - call iox_$put_chars (iocb_ptr, buff_ptr, n, code);

  - WRITES n BYTES (CHARACTERS) TO THE UNSTRUCTURED FILE OR DEVICE

  - BUFFER SHOULD CONTAIN A NEWLINE, IF ONE IS INTENDED (THERE IS NO 'put_line' ENTRY POINT)

  - IF OPEN FOR stream_output THE CHARACTERS ARE APPENDED TO THE END OF THE FILE. IF OPEN FOR stream_input_output FILE TRUNCATION OCCURS JUST BEFORE THE NEXT BYTE

- iox_$write_record

  - call iox_$write_record (iocb_ptr, buff_ptr, rec_len, code);

  - ADDS A RECORD TO A STRUCTURED FILE

  - IF OPEN FOR sequential_output, THE RECORD IS APPENDED TO THE FILE. IF OPEN FOR sequential_input_output, FILE TRUNCATION OCCURS JUST BEFORE THE NEXT RECORD

  - iox_$seek_key MUST BE CALLED BEFORE DOING A KEYED WRITE IN ORDER TO "SET THE KEY" FOR INSERTION

iox_ ENTRY POINTS

- iox_$rewrite_record

  - call iox_$rewrite_record (iocb_ptr, buff_ptr, rec_len, code);

  - REPLACES THE CURRENT RECORD IN A STRUCTURED FILE THAT HAS BEEN OPENED FOR "UPDATE"

  - IF THE CURRENT RECORD POSITION IS NULL, error_table_$no_record IS RETURNED

  - THUS IT IS FIRST NECESSARY TO "LOCATE" THE RECORD TO BE REPLACED (USING read_record, seek_key OR position ENTRY POINTS)

- iox_$read_length

  - call iox_$read_length (iocb_ptr, rec_len, code);

  - RETURNS THE LENGTH OF THE NEXT RECORD IN A STRUCTURED FILE

  - IF THE NEXT RECORD POSITION IS AT THE END OF FILE, code = error_table_$end_of_info

  - APPLICATION: TO DETERMINE HOW LONG THE BUFFER MUST BE IN ORDER TO HOLD THE NEXT RECORD TO BE READ (EXAMPLE: VARIABLE LENGTH RECORDS)

iox_ ENTRY POINTS

- iox_$delete_record

  - call iox_$delete_record (iocb_ptr, code);

  - DELETES THE CURRENT RECORD FROM THE STRUCTURED FILE, WHOSE SWITCH MUST BE OPENED FOR "UPDATE"

  - IF THE CURRENT RECORD IS NULL, code = error_table_$no_record

  - AGAIN, IT IS FIRST NECESSARY TO "LOCATE" THE RECORD TO BE DELETED (USING read_record, seek_key OR position ENTRY POINTS)

- iox_$read_key

  - call iox_$read_key (iocb_ptr, key, rec_len, code);

  - RETURNS BOTH THE KEY AND THE LENGTH OF THE NEXT RECORD IN AN INDEXED FILE

  - code = error_table_$end_of_info IF THE NEXT RECORD POSITION IS AT THE END OF FILE

  - code = error_table_$no_record IF THE NEXT RECORD POSITION IS NULL

- iox_$position

  - call iox_$position (iocb_ptr, type, n, code);

  - POSITIONS TO THE BEGINNING OR END OF A FILE, OR SKIPS FORWARD OR BACKWARD OVER A SPECIFIED NUMBER OF LINES OR CHARACTERS (UNSTRUCTURED FILES) OR RECORDS (STRUCTURED FILES)

    - type IDENTIFIES THE TYPE OF POSITIONING (INPUT)

      - -1 GO TO THE BEGINNING OF FILE (n = 0)
      - +1 GO TO THE END OF FILE (n = 0)
      - 0 SKIP NEWLINE CHARACTERS OR RECORDS (n positive or negative)
      - 2 POSITION TO AN ABSOLUTE CHARACTER OR RECORD (n)
      - 3 SKIP CHARACTERS (stream_input) (n positive or negative)

iox_ ENTRY POINTS

- iox_$modes

  - USED TO OBTAIN OR SET MODES THAT AFFECT THE SUBSEQUENT BEHAVIOR OF THE SWITCH (BEST KNOWN MODES ARE THOSE ASSOCIATED WITH tty_ : echoplex,tabs,polite,etc.)

  - call iox_$modes (iocb_ptr, new_modes, old_modes, code);

  - SWITCH MUST BE ATTACHED VIA AN I/O MODULE THAT SUPPORTS MODES (EXAMPLE: tty_ SUPPORTS MODES, vfile_ DOES NOT)

  - FOR A LIST OF THE VALID MODES, SEE THE DESCRIPTION OF THE MODULE INVOLVED

- iox_$control

  - call iox_$control (iocb_ptr, order, info_ptr, code);

    - info_ptr IS NULL OR POINTS TO DATA WHOSE FORM DEPENDS ON THE MODULE

  - PERFORMS A SPECIFIED CONTROL ORDER ON AN I/O SWITCH; THE ALLOWED ORDERS DEPEND ON THE I/O MODULE VIA WHICH THE SWITCH IS ATTACHED (REFER TO THE I/O MODULE WRITE UPS)

    - EXAMPLES OF tty_ CONTROL ORDERS: set_delay, set_editing_chars, quit_enable, hangup

    - EXAMPLE OF vfile_ CONTROL ORDER: read_position (RETURNS THE ORDINAL POSITION (0, 1, 2...) OF THE NEXT RECORD/BYTE AND THE END OF THE FILE)

AN EXAMPLE USING iox

```
print_file: proc;

dcl  iox_$attach_name entry (char (*), ptr, char (*), ptr, fixed bin (35));
dcl  iox_$detach_iocb entry (ptr, fixed bin (35));
dcl  iox_$open entry (ptr, fixed bin, bit (1) unaligned, fixed bin (35));
dcl  iox_$close entry (ptr, fixed bin (35));
dcl  iox_$put_chars entry (ptr, ptr, fixed bin (21), fixed bin (35));
dcl  iox_$read_record entry (ptr, ptr, fixed bin (21), fixed bin (21),
                             fixed bin (35));
dcl  iox_$read_length entry (ptr, fixed bin (21), fixed bin (35));
dcl  iox_$get_line entry (ptr, ptr, fixed bin (21), fixed bin (21),
                          fixed bin (35));
dcl  iox_$control entry (ptr, char (*), ptr, fixed bin (35));
dcl  iox_$user_output ext ptr;
dcl  iocb_ptr ptr init (null ());
dcl  code fixed bin (35) init (0);
dcl  com_err_ entry options (variable);
dcl  ME char (10) static init ("print_file") options (constant);
dcl  LF char (1) static options (constant) init ("
");
dcl 1 info,
      2 next_position fixed bin (34),
      2 last_position fixed bin (34);
dcl  buffer char (buf_len) based (buf_ptr);
dcl  buf_len fixed bin (21);
dcl  buf_ptr ptr init ( null() );
dcl  rec_len fixed bin (21);
dcl  i fixed bin;
dcl  (null, addr) builtin;
dcl  cleanup condition;

on cleanup call WRAPUP;

call iox_$attach_name ("sw", iocb_ptr, "vfile_ sample_file", null (), code);
if code ^= 0
then call WRAPUP;

call iox_$open (iocb_ptr, 4, "0"b, code);
if code ^= 0
then call WRAPUP;

call iox_$control (iocb_ptr, "read_position", addr(info), code);
if code ^= 0
then call WRAPUP;

call iox_$read_length (iocb_ptr, rec_len, code);
if code ^= 0
then call WRAPUP;

buf_len = rec_len + 40;
allocate buffer set (buf_ptr);
```

AN EXAMPLE USING iox

```
   do i = 1 to last_position;
      call iox_$read_record (iocb_ptr, buf_ptr, buf_len, rec_len, code);
      if code ^= 0
      then call WRAPUP;
      substr (buffer, rec_len+1, 1) = LF;
      call iox_$put_chars (iox_$user_output, buf_ptr, rec_len + 1, code);
      if code ^= 0
      then call WRAPUP;
   end /* do i */;

   call WRAPUP;

   WRAPUP: proc;

   if code ^= 0
   then call com_err_ (code, ME);

   if iocb_ptr ^= null ()
   then do;
           call iox_$close (iocb_ptr, code);
           call iox_$detach_iocb (iocb_ptr, code);
        end /* then do */;

   if buf_ptr ^= null ()
   then free buf_ptr -> buffer;

   goto FINIS;

   end /* WRAPUP */;

   FINIS:

   end /* print_file */;


   r 14:40 0.259 32

 ! vfs sample_file
   type: sequential
   records: 5
   r 14:41 0.261 19

 ! print_file
   This is record number 1
   THIS IS RECORD TWO
   Hi, I'm the third record
   Would you believe four?
   I am the last record
   r 14:41 0.288 7
```

TOPIC IX

The 'ioa_' Multics Subroutine

OBJECTIVES:

Upon completion of this topic, students should be able to:

1. Write simple character strings to the user's terminal.

2. Use iteration and conditional evaluation to form complex output strings for display on the terminal.

3. Write to a file via an I/O switch.

4. Write to a file using the Multics Virtual Memory.

CHARACTERISTICS

- USED FOR FORMATTING A CHARACTER STRING FROM FIXED-POINT NUMBERS, FLOATING-POINT NUMBERS, CHARACTER STRINGS, BIT STRINGS, AND POINTERS
  - THE CHARACTER STRING IS FORMATTED ACCORDING TO THE CONTROL CHARACTERS EMBEDDED IN AN 'ioa_' CONTROL STRING
  - THE ENTIRE PROCEDURE IS SIMILAR TO FORMATTING OUTPUT IN PL/I OR FORTRAN
- SEVERAL ENTRY POINTS ARE PROVIDED IN 'ioa_' TO PROVIDE VARIOUS OPTIONS
  - SINCE ALL OF THE ENTRY POINTS CAN BE CALLED WITH A VARIABLE NUMBER OF ARGUMENTS, THEY ALL MUST BE DECLARED 'entry options(variable)'
  - 'ioa_' NORMALLY APPENDS A NEWLINE CHARACTER TO THE END OF THE STRING CREATED
  - A CORRESPONDING ENTRY POINT IS PROVIDED FOR EVERY STANDARD ENTRY POINT WHICH SPECIFIES THAT "NO NEWLINE" IS TO BE APPENDED

ENTRY POINTS

- ENTRY POINTS IN ioa_ ARE:

  - ioa_, ioa_$nnl

    - call ioa_ (control_string, arg1, ..., argN);

    - FORMAT THE INPUT DATA ACCORDING TO THE CONTROL STRING, AND WRITE THE RESULTING STRING ON 'user_output'

  - ioa_$ioa_stream, ioa_$ioa_stream_nnl

    - call ioa_$ioa_stream (switchname, control_string, arg1, ..., argN);

    - FORMAT THE RESULTING STRING AS ABOVE, BUT THE STRING IS THEN WRITTEN TO AN I/O SWITCH SPECIFIED BY THE SWITCHNAME ARGUMENT

  - ioa_$ioa_switch, ioa_$ioa_switch_nnl

    - call ioa_$ioa_switch (iocb_ptr, control_string, arg1, ..., argN);

    - IDENTICAL TO THE ioa_$ioa_stream AND ioa_$ioa_$stream_nnl ENTRY POINTS EXCEPT THAT THE I/O SWITCH IS DESIGNATED BY A POINTER TO ITS IOCB, RATHER THAN BY SWITCHNAME (HENCE, THESE ENTRY POINTS ARE A BIT MORE EFFICIENT)

ENTRY POINTS

- ioa_$rs, ioa_$rsnnl

  - call ioa_$rs (control_string, ret_string, ret_length, arg1, ..., argN);

  - EDITING OCCURS AS IN THE ABOVE CALLS, BUT INSTEAD OF BEING WRITTEN TO AN I/O SWITCH, THE STRING IS PASSED BACK TO THE CALLER IN A CHARACTER STRING VARIABLE

  - THE CHARACTER STRING VARIABLE PROVIDED BY THE CALLER MAY BE VARYING OR NONVARYING, ALIGNED OR UNALIGNED AND OF ANY LENGTH

  - THE LENGTH OF THE CREATED STRING IS ALSO RETURNED

- ioa_$rsnp, ioa_$rsnpnnl

  - THESE ARE IDENTICAL TO THE ioa_$rs AND ioa_$rsnnl ENTRY POINTS EXCEPT THAT THEY DO "NO PADDING" OF A STRING RETURNED INTO A NONVARYING CHARACTER STRING

CONTROL STRING

- A NON-VARYING CHARACTER STRING CONSISTING OF TEXT TO BE COPIED AND/OR ioa_ CONTROL CODES

- ioa_ CONTROL CODES ARE ALWAYS IDENTIFIED BY A LEADING CIRCUMFLEX (^) CHARACTER, AND SPECIFY THE TYPE OF EDITING TO BE DONE FOR THEIR CORRESPONDING arg*i*

- PROCESSING BY ioa_ BEGINS BY SCANNING THE CONTROL STRING UNTIL A CIRCUMFLEX IS FOUND, OR THE END OF THE STRING IS REACHED

  - ANY TEXT (INCLUDING BLANKS) PASSED OVER IS COPIED TO THE OUTPUT STRING

  - CONTROL CODES ARE INTERPRETED, GENERALLY BY EDITING THE NEXT arg*i* INTO THE OUTPUT STRING IN A FASHION DICTATED BY THE CONTROL CODE

CONTROL STRING

CONTROL CODE		ACTION
^d	^nd	Edit a fixed-point decimal integer
^i	^ni	same as ^d (FOR COMPATIBILITY WITH FORTRAN)
^f	^nf ^n.df ^.df	Edit a floating-point number
^e	^ne	Edit a floating-point number in exponential form
^o	^no	Edit a fixed-point number in octal
^w	^nw	Edit a full machine word in octal
^a	^na	Edit a character string in ASCII
^b	^nb ^n.db ^.db	Edit a bit string
^p		Edit a pointer
^|	^n|	Insert formfeed character(s)
^/	^n/	Insert newline character(s)
^-	^n-	Insert horizontal tab character(s)
^x	^nx	Insert space character(s)
^^	^n^	Insert circumflex character(s)
^s	^ns	Skip argument(s)
^(	^n(	Start an iteration loop
^)		End an iteration loop
^[		Start an if/then/else or case selection group
^]		Limit the scope of a ^[
^;		Use as a clause delimiter between ^[^]
^nt	^n.mt	Insert enough space to reach column n

CONTROL STRING

- WHEN n AND/OR d APPEAR IN A CONTROL CODE, THEY GENERALLY REFER TO A FIELD WIDTH OR A REPETITION FACTOR (THE EXACT MEANING DEPENDS ON THE CONTROL CODE WITH WHICH THEY APPEAR)

  - THE n OR d MUST BE SPECIFIED AS UNSIGNED DECIMAL INTEGERS, OR AS THE LETTER "v", IN WHICH CASE, THE NEXT argi ARGUMENT (WHICH MUST BE FIXED BINARY) IS USED TO OBTAIN THE ACTUAL VALUE

- IF NO FIELD WIDTH IS SPECIFIED, ioa_ USES A FIELD LARGE ENOUGH TO CONTAIN THE DATA TO BE EDITED

- IF TOO SMALL A FIELD WIDTH IS SPECIFIED, ioa_ IGNORES THE WIDTH AND SELECTS AN APPROPRIATE WIDTH

- NUMERIC CONTROL CODES TAKE ANY PL/I NUMERIC DATA TYPE, INCLUDING A NUMERIC CHARACTER STRING, AND USE STANDARD PL/I CONVERSION ROUTINES IF NECESSARY

- ARGUMENTS THAT ARE EDITED INTO THE CONTROL STRING MAY BE ARRAYS

  - THE ELEMENTS ARE TREATED SEPARATELY IN ROW MAJOR ORDER

CONTROL STRING

- THE FOLLOWING EXAMPLES ILLUSTRATE MANY, BUT NOT ALL, OF THE FEATURES OF THE ioa_ SUBROUTINE. THE SYMBOL ß IS USED TO REPRESENT A SPACE IN THE PLACES WHERE THE SPACE IS SIGNIFICANT

```
Source:   call ioa_("This is ^a the third of ^a","Mon","July");

Result:   This is Mon the third of July

Source:   call ioa_("date ^d/^d/^d, time ^d:^d",6,20,74,2014,36);

Result:   date 6/20/74, time 2014:36

Source:   call ioa_("overflow at ^p",ptr);

Result:   overflow at 271|4671

Source:   call ioa_("^2(^2(^w ^)^/^)",w1,w2,w3,w4);

Result:   112233445566 000033004400
          000000000001 777777777777

Source:   bit="110111000011"b;
          call ioa_("^vxoct=^.3b hex=^.4b",6,bit,bit);

Result:   ßßßßßßoct=6703ßhex=DC3

Source:   call ioa_("^f ^e ^f ^5.2f",1.0,1,1e-10,1);

Result:   1. ß1.e0 ß1.e-10 ß1.00

Source:   call ioa_("^(^d ^)",1,2,56,198,456.7,3e6);

Result:   1 2 56 198 456 3000000

Source:   abs_sw=0;
          call ioa_$rsnnl("^v(Absentee user ^)^a ^a logged out.",
                          out_str,out_cnt,abs_sw,"LeValley","Shop");

Result:   out_cnt=25;
          out_str="LeValley Shop logged out."
```

Source:
```
abs sw=1; /* Using same call to ioa_$rsnnl */
call ioa_$rsnnl("^v(Absentee user ^)^a ^a logged out.",
                out_str,out_cnt,abs_sw,"LeValley","Shop");
```

Result:
```
out_cnt=39;
out_str="Absentee user LeValley Shop logged out."
```

Source:
```
dcl a(2,2)fixed bin init(1,2,3,4);
call ioa_("^d^s ^d ^w",a);
```

Result:
```
1 3 000000000004
```

Source:
```
dcl b(6:9)fixed bin init(6,7,8,9);
call ioa_("^v(^3d^)",dim(b,1),b);
```

Result:
```
  6  7  8  9
```

Source:
```
sw="0"b;
call ioa_ ("a=^d ^[b=^d^;^s^] c=^d",5,sw,7,9);
```

Result:
```
a=5 c=9
```

Source:
```
sw="1"b;
call ioa_ ("a=^d ^[b=^d^;^s^] c=^d",5,sw,7,9);
```

Result:
```
a=5 b=7 c=9
```

Source:
```
dir=">"; ename="foo";
call ioa_ ("Error in segment ^a^[>^]^a", dir,
          (dir ^=">"), ename);
```

Result:
```
Error in segment >foo
```

Source:
```
dir=">foo"; ename="bar";
call ioa_ ("Error in segment ^a^[>^]^a", dir,
          (dir ^= ">"), ename);
```

Result:
```
Error in segment >foo>bar
```

Source:
```
option=2;  /* Assume following call is on one line */
call ioa_ ("Insurance option selected:
^[no fault^;bodily injury^;propertydamage^]", option);
```

Result:
```
Insurance option selected: bodily injury
```

YOU ARE NOW READY FOR WORKSHOP #6

TOPIC X

Multics Storage System Subroutines

OBJECTIVES:

Upon completion of this topic, students should be able to:

1. Add and remove entries to and from the Multics Storage System.

2. Manipulate pathnames using Multics subroutines.

3. Obtain status information on entries in the storage system.

4. Change the access control lists (ACLs) of various entries in the storage system.

5. Use Multics subroutines to obtain information about a user's home, working, and process directories.

6. Discuss the access required to perform any of the above operations.

THE MULTICS STORAGE SYSTEM

- THE STORAGE HIERARCHY IS ORGANIZED INTO AN INVERTED TREE STRUCTURE

  - THIS TREE IS MADE UP OF DIRECTORY SEGMENTS, SEGMENTS, MULTI-SEGMENT FILES AND LINKS

- FOR NON-DIRECTORY SEGMENTS:

  - SUBJECT TO THE THREE ACCESS CONTROL MECHANISMS, THE USER IS FREE TO CREATE, DESTROY, AND MODIFY THE CONTENTS OF SEGMENTS

  - USER-CREATED SEGMENTS NORMALLY "RESIDE" IN THE RING OF THE CREATOR. THE USER IS FREE TO ACCESS SUCH SEGMENTS WITHOUT HAVING TO "CROSS" ANY RING BOUNDARIES

- FOR DIRECTORY SEGMENTS:

  - THE USER MAY CREATE, DESTROY, AND MODIFY DIRECTORY SEGMENTS, BUT NOT DIRECTLY (THEY ARE PROTECTED AGAINST DIRECT ACCESS VIA THE RING MECHANISM)

  - ALLOWING USERS TO MANIPULATE DIRECTORY SEGMENTS DIRECTLY WOULD BE INVITING CHAOS, SINCE DIRECTORY SEGMENTS DETERMINE THE INTEGRITY, SECURITY AND CONSISTENCY OF THE HIERARCHY

  - DIRECTORY SEGMENTS ARE PLACED IN RING 0 AND USERS ULTIMATELY ACCESS SUCH SEGMENTS BY USING A SYSTEM-PROVIDED GATE PROCEDURE CALLED hcs_

THE MULTICS STORAGE SYSTEM

- THE hcs_ SUBROUTINE
  - PROVIDES VARIOUS ENTRY POINTS FOR MANIPULATION OF THE STORAGE SYSTEM AND VIRTUAL ADDRESS SPACE
  - ALL ACCESS TO THE STORAGE SYSTEM IS ACCOMPLISHED VIA THIS GATE PROCEDURE

- THE STORAGE MANIPULATION SUBROUTINES COVERED IN THIS COURSE ARE SUMMARIZED BELOW:

SUMMARY OF DISCUSSED SUBROUTINES

CREATING STORAGE SYSTEM ENTITIES

hcs_$append_branch
hcs_$append_branchx
hcs_$append_link
hcs_$create_branch_
hcs_$make_seg

DELETING STORAGE SYSTEM ENTITIES

delete_$path
delete_$ptr

OBTAINING STATUS INFORMATION

hcs_$status_
hcs_$status_long
hcs_$status_minf
hcs_$status_mins

SECURITY

get_group_id_
get_group_id_$tag_star
hcs_$add_acl_entries
hcs_$add_dir_acl_entries
hcs_$delete_acl_entries
hcs_$delete_dir_acl_entries
hcs_$fs_get_mode
hcs_$list_acl
hcs_$list_dir_acl
hcs_$replace_acl
hcs_$replace_dir_acl

WORKING, DEFAULT, AND PROCESS DIRECTORIES

change_default_wdir_
change_wdir_
get_default_wdir_
get_pdir_
get_wdir_

SUMMARY OF DISCUSSED SUBROUTINES

MANIPULATING PATHNAMES

absolute_pathname_
absolute_pathname_$add_suffix
expand_pathname_
expand_pathname_$add_suffix
expand_pathname_$component
expand_pathname_$component_add_suffix
get_shortest_path_
pathname_
pathname_$component
pathname_$component_check

NAMING AND MOVING DIRECTORY ENTRIES

hcs_$chname_file
hcs_$chname_seg
hcs_$fs_move_file
hcs_$fs_move_seg

AFFECTING LENGTH OF ENTRIES

adjust_bit_count_
hcs_$set_bc
hcs_$truncate_file
terminate_file_

MANIPULATING THE ADDRESS AND NAME SPACES

hcs_$fs_get_path_name
hcs_$fs_get_ref_name
hcs_$fs_get_seg_ptr
hcs_$make_seg
initiate_file_
term_$refname
term_$seg_ptr
term_$single_refname
term_$term_
term_$unsnap
terminate_file_

This Page Intentionally Left Blank

CREATING STORAGE SYSTEM ENTITIES

	hcs_$make_seg	hcs_$append_branch	hcs_$append_branchx	hcs_$create_branch_
Requires append permission	X	X	X	X
Can use to create segments	X	X	X	X
Gives full access to *.SysDaemon.*	X	X	X	X
Obeys initial acl	X	X	X	X
Can set access for one user_id	X	X	X	X
Can specify the user_id			X	X
Can use to create directories			X	X
Can set ring brackets			X	X
Can set copy switch			X	X
Can set bit count			X	X
Can be told to chase links				X
Can move quota to directory				X
Can manipulate aim				X
Requires info structure				X
Initiates created segment	X			

CREATING STORAGE SYSTEM ENTITIES

- call hcs_$make_seg (dir_name, entryname, ref_name, mode, seg_ptr, code);

  null()

- call hcs_$append_branch (dir_name, entryname, mode, code);

- call hcs_$append_branchx (dir_name, entryname, mode, ~~rings~~, user_id, ~~user_id~~, dir_sw, copy_sw, bit_count, code);

- call hcs_$create_branch_ (dir_name, entryname, info_ptr, code);

info_ptr POINTS TO THE FOLLOWING STRUCTURE:

```
/* BEGIN INCLUDE FILE - - - create_branch_info.incl.pl1
   - - - created January 1975 */

/* this include files gives the argument structure for
   create_branch_ */

dcl 1 create_branch_info aligned based,
      2 version fixed bin,
        /* set this to the largest value given below */
      2 switches unaligned,
        3 dir_sw bit (1) unaligned,
          /* if on, a directory branch is wanted */
        3 copy_sw bit (1) unaligned,
          /* if on, initiating segment will be done by copying */
        3 chase_sw bit (1) unaligned,
          /* if on, if pathname is a link, it will be chased */
        3 priv_upgrade_sw bit (1) unaligned,
          /* privileged creation (ring 1) of upgraded object */
        3 parent_ac_sw bit (1) unaligned,
          /* if on, use parent's access class for seg or
             dir created */
        3 mbz1 bit (31) unaligned,
          /* pad to full word */
      2 mode bit (3) unaligned,
        /* segment or directory for acl for userid */
      2 mbz2 bit (33) unaligned,
        /* pad to full word */
      2 rings (3) fixed bin (3),
        /* branch's ring brackets */
      2 userid char (32),
        /* user's access control name */
      2 bitcnt fixed bin (24),
        /* bit count of the segment */
      2 quota fixed bin (18),
        /* for directories, this am't of quota will be moved
           to it */
      2 access_class bit (72);
        /* is the access class of the body of the branch */

/* The following versions are implemented . . . */
/* (Changes to structure require defining new static
   initialized variable) */

dcl  create_branch_version_1 static fixed bin init (1);
       /* branch info valid through access class field */

/* END INCLUDE FILE - - - create_branch_info.incl.pl1 - - - */
```

- NOTES:

  - FOR BOTH hcs_$make_seg AND hcs_$append_branch:

    - THE BIT COUNT AND COPY SWITCH ARE SET TO 0

    - THE SPECIFIED MODE IS SET FOR Person_id.Project_id.*

  - FOR hcs_$make_seg, hcs_$append_branch AND hcs_$append_branchx THE MODE IS SPECIFIED AS FOLLOWS:

    - FOR SEGMENTS:

read	the 8-bit is 1 (01000b)
execute	the 4-bit is 1 (00100b)
write	the 2-bit is 1 (00010b)

    - FOR DIRECTORIES:

status	the 8-bit is 1 (01000b)
modify	the 2-bit is 1 (00010b)
append	the 1-bit is 1 (00001b)

  - THE MODE FOR hcs_$create_branch_ IS SPECIFIED IN SIMILAR MANNER, USING ONLY 3 BITS

```
/* BEGIN INCLUDE FILE ... access_mode_values.incl.pl1

   Values for the "access mode" argument so often used in hardcore
   James R. Davis 26 Jan 81  MCR 4844
   Added constants for SM access 4/28/82 Jay Pattin
*/


dcl (N_ACCESS                  init ("000"b),
     R_ACCESS                  init ("100"b),
     E_ACCESS                  init ("010"b),
     W_ACCESS                  init ("001"b),
     RE_ACCESS                 init ("110"b),
     REW_ACCESS                init ("111"b),
     RW_ACCESS                 init ("101"b),

     S_ACCESS                  init ("100"b),
     M_ACCESS                  init ("010"b),
     A_ACCESS                  init ("001"b),
     SA_ACCESS                 init ("101"b),
     SM_ACCESS                 init ("110"b),
     SMA_ACCESS                init ("111"b))
  bit (3) internal static options (constant);

dcl (N_ACCESS_BIN              init (00000b),
     R_ACCESS_BIN              init (01000b),
     E_ACCESS_BIN              init (00100b),
     W_ACCESS_BIN              init (00010b),
     RW_ACCESS_BIN             init (01010b),
     RE_ACCESS_BIN             init (01100b),
     REW_ACCESS_BIN            init (01110b),

     S_ACCESS_BIN              init (01000b),
     M_ACCESS_BIN              init (00010b),
     A_ACCESS_BIN              init (00001b),
     SA_ACCESS_BIN             init (01001b),
     SM_ACCESS_BIN             init (01010b),
     SMA_ACCESS_BIN            init (01011b))

     fixed bin (5) internal static options (constant);

/* END INCLUDE FILE ... access_mode_values.incl.pl1 */
```

CREATING STORAGE SYSTEM ENTITIES

- hcs_$append_link

  - call hcs_$append_link (dir_name, entryname, path, code);

  - CREATES A LINK IN SPECIFIED DIRECTORY

  - LINK'S TARGET NEEDN'T EXIST AT CREATION TIME (CODE OF ZERO STILL RETURNED)

  - APPEND PERMISSION REQUIRED ON CONTAINING DIRECTORY

DELETING SEGMENTS, DIRECTORIES, AND LINKS

- delete_
  - HAS TWO ENTRY POINTS
    - delete_$path
      - GIVEN AN ENTRYNAME, DELETES SEGMENTS, MSFs, DIRECTORIES, AND LINKS
    - delete_$ptr
      - GIVEN A POINTER, DELETES SEGMENTS ONLY
  - call delete_$path (dir_name, entryname, switches, caller, code);
  - call delete_$ptr (seg_ptr, switches, caller, code);
  - DIRECTORY TO BE DELETED NEED NOT BE EMPTY
  - UNSNAPS ANY LINKS THIS PROCESS HAS SNAPPED TO THE OBJECTS DELETED
    - NOTE: delete_ CAN'T PREVENT DISASTER WHEN ONE PROCESS DELETES ANOTHER'S SHARED SEGMENT
  - THE 6 BIT INPUT VARIABLE 'switches' MAKES THIS SUBROUTINE EXTREMELY FLEXIBLE
    - SEE THE SUBROUTINES MANUAL FOR DETAILS OF THE 6 SWITCHES (force_sw, question_sw, directory_sw, segment_sw, link_sw, chase_sw)

OBTAINING STATUS INFORMATION

- THE FOLLOWING 4 ENTRY POINTS RETURN STATUS INFORMATION FOR A DIRECTORY ENTRY (LISTED IN ORDER OF INCREASING COMPLEXITY)

  → hcs_$status_mins

  → hcs_$status_minf

  hcs_$status_

  hcs_$status_long

  - ALL THE ABOVE ENTRY POINTS HAVE A CURIOUS ACCESS REQUIREMENT

    - INFORMATION IS RETURNED IF CALLER HAS STATUS ON THE CONTAINING DIRECTORY, OR NON-NULL ACCESS ON THE ENTRY

    - ENTRYNAMES ARE NOT RETURNED UNLESS THE CALLER HAS STATUS ACCESS ON THE CONTAINING DIRECTORY

  - TO THE STATUS ENTRY POINTS, DIRECTORIES AND MULTI-SEGMENT FILES LOOK IDENTICAL

    - THE ONLY DISTINGUISHING ATTRIBUTE IS THE BIT COUNT

      - BIT COUNT = 0 FOR A DIRECTORY

      - BIT COUNT = NUMBER OF COMPONENTS FOR A MSF

OBTAINING STATUS INFORMATION

- hcs_$status_minf

  - call hcs_$status_minf (dir_name, entryname, chase_sw, type, bit_count, code);

  - RETURNS BIT COUNT AND ENTRY TYPE OF ENTRY, GIVEN A PATH

    - TYPE OF ENTRY:

      0 MEANS link
      1 MEANS segment
      2 MEANS msf OR directory

    - OFTEN USED WHEN TRYING TO DISTINGUISH BETWEEN DIR AND MSF

- hcs_$status_mins

  - call hcs_$status_mins (seg_ptr, type, bit_count, code);

  - RETURNS BIT COUNT AND ENTRY TYPE OF A SEGMENT GIVEN A POINTER TO THE SEGMENT

OBTAINING STATUS INFORMATION

- hcs_$status_

  - call hcs_$status_ (dir_name, entryname, chase_sw, status_ptr, status_area_ptr, code);

  - RETURNS INFORMATION ABOUT A SEGMENT, DIR, MSF, OR LINK:

    - INFORMATION INCLUDES ENTRY TYPE, DATE TIME CONTENTS LAST MODIFIED, DATE TIME LAST USED, NUMBER OF RECORDS USED, USER'S RAW MODE, USER'S EFFECTIVE MODE AND ENTRYNAMES (NO BIT COUNT)

  - CALLER MUST PROVIDE

    - POINTER TO CALLER-ALLOCATED INFO STRUCTURE

    - POINTER TO CALLER-DESIGNATED AREA TO CONTAIN "names" (IF NULL, NO NAMES RETURNED)

```
/* --- BEGIN include file status_structures.incl.pl1 --- */

/* Revised from existing include files 09/26/78
   by C. D. Tavares */

/* This include file contains branch and link structures
   returned by hcs_$status_ and hcs_$status_long. */

dcl 1 status_branch aligned based (status_ptr),
    2 short aligned,
      3 type fixed bin (2) unaligned unsigned,
          /* seg, dir, or link */
      3 nnames fixed bin (16) unaligned unsigned,
          /* number of names */
      3 names_relp bit (18) unaligned,
          /* see entry_names dcl */
      3 dtcm bit (36) unaligned,
          /* date/time contents last modified */
      3 dtu bit (36) unaligned,
          /* date/time last used */
      3 mode bit (5) unaligned,
          /* caller's effective access */
      3 raw_mode bit (5) unaligned,
          /* caller's raw "rew" modes */
      3 pad1 bit (8) unaligned,
      3 records_used fixed bin (18) unaligned unsigned,
          /* number of NONZERO pages used */

/* Limit of information returned by hcs_$status_ */

    2 long aligned,
      3 dtd bit (36) unaligned,
          /* date/time last dumped */
      3 dtem bit (36) unaligned,
          /* date/time branch last modified */
      3 lvid bit (36) unaligned,
          /* logical volume ID */
      3 current_length fixed bin (12) unaligned unsigned,
          /* number of last page used */
      3 bit_count fixed bin (24) unaligned unsigned,
          /* reported length in bits */
      3 pad2 bit (8) unaligned,
      3 copy_switch bit (1) unaligned,
          /* copy switch */
      3 tpd_switch bit (1) unaligned,
          /* transparent to paging device switch */
      3 mdir_switch bit (1) unaligned,
          /* is a master dir */
      3 damaged_switch bit (1) unaligned,
          /* salvager warned of possible damage */
      3 synchronized_switch bit (1) unaligned,
          /* DM synchronized file */
      3 pad3 bit (5) unaligned,
```

```
      3 ring_brackets (0:2) fixed bin (6) unaligned unsigned,
      3 uid bit (36) unaligned;              /* unique ID */

dcl 1 status_link aligned based (status_ptr),
    2 type fixed bin (2) unaligned unsigned, /* as above */
    2 nnames fixed bin (16) unaligned unsigned,
    2 names_relp bit (18) unaligned,
    2 dtem bit (36) unaligned,
    2 dtd bit (36) unaligned,
    2 pathname_length fixed bin (17) unaligned,
      /* see pathname */
    2 pathname_relp bit (18) unaligned;  /* see pathname */

dcl  status_entry_names (status_branch.nnames)
       character (32) aligned based
       (pointer (status_area_ptr, status_branch.names_relp)),
        /* array of names returned */
     status_pathname character (status_link.pathname_length)
       aligned based
       (pointer (status_area_ptr, status_link.pathname_relp)),
        /* link target path */
     status_area_ptr pointer,
     status_ptr pointer;

dcl (Link initial (0),
     Segment initial (1),
     Directory initial (2)) fixed bin internal static
                            options (constant);
       /* values for type fields declared above */

/* --- END include file status_structures.incl.pl1 --- */
```

OBTAINING STATUS INFORMATION

- hcs_$status_long

  - call hcs_$status_long (dir_name, entryname, chase_sw, status_ptr, status_area_ptr, code);

  - RETURNS EVERYTHING hcs_$status_ RETURNS PLUS:

    - DATE-TIME-LAST-DUMPED (SEGS ONLY)

    - CURRENT LENGTH IN 1024-WORD UNITS (SEGS, MSFS)

    - BIT COUNT (SEGS, MSFS)

    - PHYSICAL VOLUME ID OF STORAGE DEVICE ON WHICH ENTRY CURRENTLY RESIDES

    - COPY AND DAMAGED SWITCH VALUES

      SEE THE switch_on and switch_off COMMANDS (AG92)

    - RING BRACKETS

    - SEGMENT UNIQUE ID

OBTAINING STATUS INFORMATION

- OTHER ENTRY POINTS THAT RETURN STATUS TYPE INFORMATION[1]
  - hcs_$get_author, hcs_$get_bc_author
  - hcs_$get_max_length, hcs_$get_max_length_seg
  - hcs_$get_safety_sw, hcs_$get_safety_sw_seg
  - hcs_$get_link_target

- TO OBTAIN STATUS INFORMATION FOR ARCHIVE COMPONENTS SEE
  - archive_$get_component_info
  - archive_$list_components
  - archive_$next_component_info

---

1 COVERED IN MULTICS COURSE F15D

OBTAINING STATUS INFORMATION

AN EXAMPLE

```
Status: proc;

dcl 1 status_branch aligned based (status_ptr),
      2 type fixed bin (2) unaligned unsigned,
      2 nnames fixed bin (16) unaligned unsigned,
      2 names_relp bit (18) unaligned,
      2 dtcm bit (36) unaligned,
      2 dtu bit (36) unaligned,
      2 mode bit (5) unaligned,
      2 raw_mode bit (5) unaligned,
      2 pad1 bit (8) unaligned,
      2 records_used fixed bin (18) unaligned unsigned;
dcl   status_entry_names (status_branch.nnames) character (32) aligned
        based (pointer (get_system_free_area_(), status_branch.names_relp));
dcl   pointer builtin;
dcl   get_system_free_area_ entry() returns(ptr);
dcl   status_ptr ptr;
dcl (ioa_,
     com_err_) entry options (variable);
dcl   hcs_$status_ entry (char (*), char (*), fixed bin (1), ptr,
                          ptr, fixed bin (35));
dcl   code fixed bin (35);
dcl   i;

allocate status_branch;
call hcs_$status_ (">udd>MEDclass>F15C", "s1", 0, status_ptr,
                   get_system_free_area_(), code);
if code ^= 0
then do;
        call com_err_ (code, "Status");
        return;
     end /* then do */;

call ioa_ ("^/s1 is a ^[link^;segment^;directory^] with ^d names:",
           status_branch.type + 1, status_branch.nnames);
do i = 1 to status_branch.nnames;
   call ioa_("     ^a", status_entry_names(i));
end /* do i */;

end /* Status */;
```

OBTAINING STATUS INFORMATION

AN EXAMPLE

```
r 15:00 0.148 19

! Status

s1 is a directory with 2 names:
     Student_01
     s1
r 15:00 0.124 6
```

SECURITY

- MULTICS HAS THREE ACCESS CONTROL MECHANISMS
  - THE ACCESS CONTROL LIST MECHANISM (ACLS)
  - THE ACCESS ISOLATION MECHANISM (AIM)
  - THE RING MECHANISM
- hcs_ AND OTHER SUBROUTINES ENABLE US TO MANIPULATE THESE MECHANISMS

ACCESS CONTROL LISTS

- hcs_$add_acl_entries

  - call hcs_$add_acl_entries (dir_name, entryname, acl_ptr, acl_count, code);

  - ADDS OR CHANGES ("SETS") ACL ON A SEGMENT (rewn)

    - CALLER MUST ALLOCATE AND FILL IN AN ARRAY OF STRUCTURES

    - "MATCHING" ACCESS NAMES ACCEPTABLE TO THE set_acl COMMAND ARE NOT ACCEPTABLE

    - SEE msf_manager_$acl_add FOR MULTI-SEGMENT FILES[1]

- hcs_$add_dir_acl_entries

  - call hcs_$add_dir_acl_entries (dir_name, entryname, acl_ptr, acl_count, code);

  - ADDS OR CHANGES ("SETS") ACL ON DIRECTORIES (sman)

    - SIMILAR TO hcs_$add_acl_entries EXCEPT STRUCTURE MISSING extended_mode

---

1 COVERED IN MULTICS COURSE F15D

```
/* Begin include file -- acl_structures.incl.pl1 BIM 3/82 */
/* format: style3 */

declare   acl_ptr              pointer;
declare   acl_count            fixed bin;

declare   1 segment_acl        aligned based (acl_ptr),
            2 version          fixed bin,
            2 count            fixed bin,
            2 entries          (acl_count refer (segment_acl.count))
                               aligned like segment_acl_entry;

declare   1 segment_acl_entry  aligned based,
            2 access_name      character (32) unaligned,
            2 mode             bit (36) aligned,
            2 extended_mode    bit (36) aligned,
            2 status_code      fixed bin (35);

declare   1 segment_acl_array (acl_count) aligned like
                               segment_acl_entry based (acl_ptr);

declare   1 directory_acl      aligned based (acl_ptr),
            2 version          fixed bin,
            2 count            fixed bin,
            2 entries          (acl_count refer (directory_acl.cour
                               aligned like directory_acl_entry;

declare   1 directory_acl_entry based,
            2 access_name      character (32) unaligned,
            2 mode             bit (36) aligned,
            2 status_code      fixed bin (35);

declare   1 directory_acl_array (acl_count) aligned like
                               directory_acl_entry based (acl_ptr);

declare   1 delete_acl_entry   aligned based,
            2 access_name      character (32) unaligned,
            2 status_code      fixed bin (35);

declare   1 delete_acl         based (acl_ptr) aligned,
            2 version          fixed bin,
            2 count            fixed bin,
            2 entries          (acl_count refer (delete_acl.count))
                               aligned like delete_acl_entry;

declare   1 delete_acl_array   (acl_count) aligned like
                               delete_acl_entry based (acl_ptr);

declare ACL_VERSION_1 internal static fixed bin init (1)
                               options (constant);

/* End include file acl_structures.incl.pl1 */
```

ACCESS CONTROL LISTS

- hcs_$delete_acl_entries

  - call hcs_$delete_acl_entries (dir_name, entryname, acl_ptr, acl_count, code);

  - DELETES ONE OR MORE ENTRIES FROM A SPECIFIED SEGMENT'S ACL

    - USES A STRUCTURE ALLOCATED BY CALLER

    - "MATCHING" ACCESS NAMES ACCEPTABLE TO THE delete_acl COMMAND ARE NOT ACCEPTABLE TO hcs_$delete_acl_entries

    - SEE msf_manager_$acl_delete FOR MULTI-SEGMENT FILES[1]

- hcs_$delete_dir_acl_entries

  - call hcs_$delete_dir_acl_entries (dir_name, entryname, acl_ptr, acl_count, code);

  - DELETES ONE OR MORE ENTRIES FROM A SPECIFIED DIRECTORY'S ACL

    - OTHERWISE SIMILAR TO hcs_$delete_acl_entries

---

1 COVERED IN MULTICS COURSE F15D

ACCESS CONTROL LISTS

- hcs_$list_acl

  - call hcs_$list_acl (dir_name, entryname, area_ptr, area_ret_ptr, acl_ptr, acl_count, code);

  - RETURNS ALL OR PART OF A SEGMENT'S ACL IN A 'segment_acl' STRUCTURE (SAME STRUCTURE AS USED BY hcs_$add_acl_entries)

  - THERE ARE TWO DIFFERENT WAYS TO USE THIS ENTRY POINT:

    - IF ENTIRE ACL REQUIRED:

      - SET "area_ptr" NON-NULL AND EXPECT BACK "acl_count" AND "area_ret_ptr"

      - SUBROUTINE ALLOCATES AN ARRAY OF STRUCTURES

    - IF JUST SOME MODE ENTRIES REQUIRED:

      - SET "area_ptr" NULL

      - USER ALLOCATES AN ARRAY OF PARTIALLY FILLED IN STRUCTURES

      - PASS A PTR TO THIS ARRAY (acl_ptr)

      - MODES AND CODES WILL HAVE BEEN FILLED IN UPON RETURN

- hcs_$list_dir_acl

  - call hcs_$list_dir_acl (dir_name, entryname, area_ptr, area_ret_ptr, acl_ptr, acl_count, code);

  - RETURNS ALL OR PART OF A DIRECTORY'S ACL

    - SIMILAR TO hcs_$list_acl EXCEPT USES dir_acl STRUCTURE

ACCESS CONTROL LISTS

- hcs_$replace_acl

  - call hcs_$replace_acl (dir_name, entryname, acl_ptr, acl_count, no_sysdaemon_sw, code);

  - REPLACES ENTIRE ACL FOR A SEGMENT WITH A USER-SUPPLIED ONE

    - USES SAME STRUCTURE AS hcs_$add_acl_entries AND hcs_$list_acl

    - CAN (OPTIONALLY) ADD "rw" FOR *.SysDaemon.*

    - CAN BE MADE TO DELETE ENTIRE ACL (IF acl_count=0)

- hcs_$replace_dir_acl

  - call hcs_$replace_dir_acl (dir_name, entryname, acl_ptr, acl_count, no_sysdaemon_sw, code);

  - REPLACES ENTIRE ACL FOR A DIRECTORY

    - USES SAME STRUCTURE AS hcs_$`dd_dir_acl_entries AND hcs_$list_dir_acl

    - CAN (OPTIONALLY) ADD "sma" FOR *.SysDaemon.*

    - CAN BE MADE TO DELETE ENTIRE ACL

ACCESS CONTROL LISTS

- hcs_$fs_get_mode

  - call hcs_$fs_get_mode (seg_ptr, mode, code);

  - RETURNS THE EFFECTIVE ACCESS MODE (rew) OF THE CALLER ON A SPECIFIED SEGMENT

    - TAKES INTO ACCOUNT ACL, RING BRACKETS AND CURRENT VALIDATION LEVEL

    - NOTE: SINCE A POINTER IS PASSED, SEGMENT MUST HAVE BEEN MADE KNOWN, WHICH IMPLIES USER HAS NON-NULL ACCESS

- get_group_id_

  - user_id = get_group_id_ ();

  - RETURNS IN A char(32) nonvarying Personid.Projectid.tag

- get_group_id_$tag_star

  - user_id = get_group_id_$tag_star ();

  - RETURNS Personid.Projectid.*

WORKING, DEFAULT, AND PROCESS DIRECTORIES

- change_wdir_

  call change_wdir_ (path, code);

  CHANGES THE WORKING DIRECTORY TO THE SPECIFIED DIRECTORY

  REQUIRES ABSOLUTE PATHNAME

  COMMAND INTERFACE: cwd

- get_wdir_

  working_dir = get_wdir_ ();

  RETURNS THE ABSOLUTE PATHNAME OF THE USER'S CURRENT WORKING DIRECTORY IN A char(168) nonvarying

  COMMAND INTERFACE: pwd

WORKING, DEFAULT, AND PROCESS DIRECTORIES

- get_pdir_

  - process_dir = get_pdir_ ();

  - THIS FUNCTION RETURNS THE ABSOLUTE PATHNAME OF THE USER'S PROCESS DIRECTORY IN A char(168)nonvarying

  - COMMAND INTERFACE: pd

- get_default_wdir_

  - default_wdir = get_default_wdir_ ();

  - RETURNS THE ABSOLUTE PATHNAME OF THE CALLER'S DEFAULT WORKING DIRECTORY IN A char(168) nonvarying

  - COMMAND INTERFACE: pdwd

- change_default_wdir_

  call change_default_wdir_ (path, code);

  CHANGES THE USER'S CURRENT DEFAULT WORKING DIRECTORY TO THE DIRECTORY SPECIFIED

  COMMAND INTERFACE: cdwd

MANIPULATING PATHNAMES

- expand_pathname_

  - call expand_pathname_ (pathname, dirname, entryname, code);

  - CONVERTS A RELATIVE OR ABSOLUTE PATHNAME INTO A DIRECTORY PATHNAME AND AN ENTRYNAME

    - COVERED IN TOPIC 5

- expand_pathname_$add_suffix

  - call expand_pathname_$add_suffix (pathname, suffix, dirname, entryname, code);

  - SAME AS expand_pathname_, BUT ALSO ADDS A SPECIFIED SUFFIX ONTO THE ENTRYNAME, IF THAT SUFFIX IS NOT ALREADY PRESENT

- expand_pathname_$component

  - call expand_pathname_$component (pathname, dirname, entryname, componentname, code);

  - EXPANDS A RELATIVE OR ABSOLUTE PATHNAME INTO A DIRECTORY NAME, AN ARCHIVE NAME, AND AN ARCHIVE COMPONENT PORTION (OR INTO A DIRECTORY NAME AND ENTRYNAME PORTION IF NO COMPONENT NAME IS PRESENT)

MANIPULATING PATHNAMES

- expand_pathname_$component_add_suffix

  ▯ call expand_pathname_$component_add_suffix (pathname, suffix, dirname, entryname, componentname, code);

  ▮ SAME AS expand_pathname_$component, BUT ALSO ADDS A SPECIFIED SUFFIX TO EITHER THE ENTRYNAME OR THE COMPONENT NAME, IF NOT ALREADY PRESENT

- absolute_pathname_

  ▯ call absolute_pathname_ (pathname, full_pathname, code);

  ▮ CONVERTS A RELATIVE OR ABSOLUTE PATHNAME INTO AN ABSOLUTE PATHNAME

- absolute_pathname_$add_suffix

  ▯ call absolute_pathname_$add_suffix (pathname, suffix, full_pathname, code);

  ▮ SAME AS absolute_pathname_, BUT ALSO ADDS A SPECIFIED SUFFIX IF THAT SUFFIX IS NOT ALREADY PRESENT

- get_shortest_path_

  - shortest_path = get_shortest_path_ (original_path);

- pathname_

  - path = pathname_ (dirname, entryname);

  - GIVEN A DIRECTORY NAME AND AN ENTRY NAME, RETURNS THE PATHNAME OF THE ENTRY IN A char (168)

    - IF THE RESULTING PATHNAME IS >168 CHARACTERS, THE LAST 20 CHARACTERS OF THE RESULT ARE SET TO "<PATHNAME TOO LONG>"

- pathname_$component

  - path = pathname_$component (dirname, entryname, component_name);

  - GIVEN A DIRECTORY NAME, AN ENTRY NAME, AND OPTIONALLY, AN ARCHIVE COMPONENT NAME, CONSTRUCTS A PATHNAME OR AN ARCHIVE COMPONENT PATHNAME

    - IF COMPONENT NAME IS NULL AND THE RESULTING PATHNAME IS >168 CHARACTERS, THE LAST 20 CHARACTERS OF THE PATHNAME ARE SET TO "<PATHNAME TOO LONG>"

    - IF COMPONENT NAME IS NOT NULL AND THE RESULTING PATHNAME IS >194 CHARACTERS, THEN THE LAST 20 CHARACTERS OF THE dirname>entryname PORTION OF THE ARCHIVE PATHNAME ARE CHANGED TO "<PATHNAME TOO LONG>" AND THE component_name REMAINS IN THE PATHNAME

MANIPULATING PATHNAMES

- pathname_$component_check

  - call pathname_$component_check (dirname, entryname, component_name, path, code);

  - SAME AS pathname_$component EXCEPT A STATUS CODE INDICATES TRUNCATION INSTEAD OF AN INVALID PATHNAME CONTAINING "<PATHNAME TOO LONG>"

- NOTE: NONE OF THE PREVIOUS SUBROUTINES CHECK TO SEE IF THE ENTRY EXISTS

YOU ARE NOW READY FOR WORKSHOP #7

TOPIC XI

Multics Storage System Subroutines--Continued

OBJECTIVES:

Upon completion of this topic, students should be able to:

1. Move entries from one place in the storage system to another.

2. Change the lengths and names of entries in the storage system.

3. Add and remove entries to and from the user's name space.

NAMING AND MOVING DIRECTORY ENTRIES

- hcs_$chname_file

  - call hcs_$chname_file (dir_name, entryname, oldname, newname, code);

  - ADDS, DELETES, OR CHANGES NAMES OF SEGMENTS, DIRECTORIES, MSFS, OR LINKS (SPECIFIED BY NAME)

    - EITHER oldname OR newname (BUT NOT BOTH) MAY BE null ("")

    - MODIFY PERMISSION ON CONTAINING DIRECTORY REQUIRED

- hcs_$chname_seg

  - call hcs_$chname_seg (seg_ptr, oldname, newname, code);

  - ADDS, DELETES, OR CHANGES NAMES OF A SEGMENT, GIVEN A POINTER TO IT

    - OTHERWISE SIMILAR TO hcs_$chname_file

NAMING AND MOVING DIRECTORY ENTRIES

- hcs_$fs_move_file

  call hcs_$fs_move_file (from_dir, from_entry, at_sw,
  to_dir, to_entry, code);

  MOVES CONTENTS OF ONE SEGMENT TO ANOTHER SEGMENT

  - at_sw HAS 2 BITS (fixed bin(2))
    - THE APPEND BIT ON FORCES CREATION OF NEW SEGMENT IF IT DOESN'T EXIST
    - THE TRUNCATE BIT ON FORCES TRUNCATION OF NEW SEGMENT IF IT EXISTS
  - OLD (ZEROED OUT) SEGMENT REMAINS
    - RECORD LENGTH = 0
    - BIT COUNT NOT CHANGED
  - NEW SEGMENT'S BIT COUNT NOT ADJUSTED
  - ACCESS REQUIRED
    - READ AND WRITE ON OLD SEGMENT
    - READ, WRITE ON NEW SEGMENT (IF IT EXISTS)
    - APPEND ON NEW SEGMENT'S CONTAINING DIRECTORY (IF SEG MUST BE CREATED)
  - FOR A SHORT TIME, 2 IMAGES EXIST (POSSIBLE QUOTA PROBLEM)

NAMING AND MOVING DIRECTORY ENTRIES

- hcs_$fs_move_seg

  call hcs_$fs_move_seg (from_ptr, to_ptr, trun_sw, code);

  MOVES CONTENTS OF ONE SEGMENT TO ANOTHER, GIVEN POINTERS TO EACH

  trun_sw HAS ONLY ONE BIT

  OTHERWISE SIMILAR TO hcs_$fs_move_file

AFFECTING THE LENGTH OF A FILE

- hcs_$truncate_file

  - call hcs_$truncate_file (dir_name, entryname, length, code);

  - TRUNCATES A SEGMENT TO A SPECIFIED LENGTH (IN WORDS), GIVEN ITS NAME AND CONTAINING DIRECTORY NAME

    - TRAILING FULL PAGES ARE DISCARDED

    - ZEROES ARE STORED (IN LAST PAGE) BEYOND SPECIFIED LENGTH

    - WRITE PERMISSION ON TARGET REQUIRED

    - THE BIT COUNT IS NOT SET (USE EITHER hcs_$set_bc OR adjust_bit_count_)

    - truncate COMMAND PERFORMS BOTH hcs_$truncate_file AND hcs_$set_bc

AFFECTING THE LENGTH OF A FILE

- hcs_$set_bc

  - call hcs_$set_bc (dir_name, entryname, bit_count, code);

  - SETS THE BIT COUNT OF A SEGMENT TO A SPECIFIED NUMBER, GIVEN ITS NAME AND CONTAINING DIRECTORY

    - ALSO SETS BIT COUNT AUTHOR TO USER ID OF CALLER

    - WRITE PERMISSION ON SEGMENT REQUIRED

    - MODIFY PERMISSION ON DIRECTORY NOT REQUIRED

    - COMMAND INTERFACE: set_bit_count (sbc)

- adjust_bit_count_

  - call adjust_bit_count_ (dir_name, entryname, char_sw, bit_count, code);

  - SETS THE BIT COUNT TO THE LAST NON-ZERO WORD OR BYTE

    - WORKS ON SEGMENTS AND MULTISEGMENT FILES

    - char_sw DETERMINES WHETHER THE BIT COUNT IS ADJUSTED TO THE LAST WORD OR CHARACTER

    - COMMAND INTERFACE: adjust_bit_count (abc)

AFFECTING THE LENGTH OF A FILE

- terminate_file_

  - call terminate_file_ (seg_ptr, bit_count, switches, code);

  - PERFORMS COMMON OPERATIONS OFTEN NECESSARY AFTER A PROGRAM HAS FINISHED USING A SEGMENT, SUCH AS

    - SETTING THE BIT COUNT

    - TRUNCATING THE SEGMENT

    - ENSURING THAT BITS IN THE LAST WORD OF THE SEGMENT AFTER THE BIT COUNT ARE ZERO

    - TERMINATING A NULL REFERENCE NAME

    - ENSURING THAT ALL MODIFIED PAGES OF THE SEGMENT ARE NO LONGER IN MAIN MEMORY

  - USES THE terminate_file_switches STRUCTURE

AFFECTING THE LENGTH OF A FILE

```
/* BEGIN INCLUDE FILE ... terminate_file.incl.pl1 */
/* format: style2,^inddcls,idind32 */

declare 1 terminate_file_switches       based,
          2 truncate                    bit (1) unaligned,
          2 set_bc                      bit (1) unaligned,
          2 terminate                   bit (1) unaligned,
          2 force_write                 bit (1) unaligned,
          2 delete                      bit (1) unaligned;

declare TERM_FILE_TRUNC                 bit (1) internal
                  static options (constant) initial ("1"b);
declare TERM_FILE_BC                    bit (2) internal
                  static options (constant) initial ("01"b);
declare TERM_FILE_TRUNC_BC              bit (2) internal
                  static options (constant) initial ("11"b);
declare TERM_FILE_TERM                  bit (3) internal
                  static options (constant) initial ("001"b);
declare TERM_FILE_TRUNC_BC_TERM         bit (3) internal
                  static options (constant) initial ("111"b);
declare TERM_FILE_FORCE_WRITE           bit (4) internal
                  static options (constant) initial ("0001"b);
declare TERM_FILE_DELETE                bit (5) internal
                  static options (constant) initial ("00001"b);

/* END INCLUDE FILE ... terminate_file.incl.pl1 */
```

- terminate_file_ SHOULD NEVER BE CALLED FROM A CLEANUP HANDLER WITH THE truncate OR set_bc SWITCHES ON (seg_ptr MAY CONTAIN AN INVALID SEGMENT NUMBER)

- force_write SHOULD BE USED ONLY WHEN DATA INTEGRITY IS ABSOLUTELY ESSENTIAL AS IT MAY INTRODUCE A SUBSTANTIAL REAL TIME DELAY IN EXECUTION

MANIPULATING THE ADDRESS AND NAME SPACES

- DEFINITION OF TERMS

  - ADDRESS SPACE IS

    - THE PER-PROCESS COLLECTION OF SEGMENTS THAT CAN BE DIRECTLY REFERENCED VIA HARDWARE

    - EXPANDING AND CONTRACTING DURING A PROCESS' LIFE

    - A COLLECTION OF "KNOWN" SEGMENTS

    - REFLECTED IN THE DSEG (AND KST)

    - MANAGED

      - AUTOMATICALLY BY THE DYNAMIC LINKER

      - IMPLICITLY, BY A CALL TO SOME SYSTEM COMMAND

        EXAMPLE: print my_dir>my_seg

      - EXPLICITLY, BY USER CALLS TO SYSTEM COMMANDS OR SUBROUTINES THAT MANAGE THE ADDRESS SPACE

MANIPULATING THE ADDRESS AND NAME SPACES

- NAME SPACE IS
  - THE PER-PROCESS COLLECTION OF "REFERENCE" NAMES (OPTIONALLY) ASSOCIATED WITH EACH "KNOWN" SEGMENT
  - EXPANDING AND (RARELY) CONTRACTING DURING A PROCESS' LIFE
  - REFLECTED IN THE REFERENCE NAME TABLE (RNT)
  - AN IMPORTANT PART OF SEARCH RULES (INITIATED SEGMENTS LIST)
  - MANAGED
    - AUTOMATICALLY BY THE DYNAMIC LINKER
    - EXPLICITLY, BY USER CALLS TO SYSTEM COMMANDS OR SUBROUTINES THAT MANAGE THE NAME SPACE

- MAKING-KNOWN INVOLVES
  - DEVELOPING A POINTER TO A SPECIFIED SEGMENT (ASSIGNING A SEGMENT NUMBER)
  - ADDING AN ENTRY TO THE KST AND DSEG

- INITIATING (A REFERENCE NAME) INVOLVES
  - EXPANDING THE PROCESS' NAME SPACE
  - ADDING AN ENTRY TO THE RNT

MANIPULATING THE ADDRESS AND NAME SPACES

- TERMINATING (A REFERENCE NAME) INVOLVES
  - CONTRACTING THE PROCESS' NAME SPACE
  - REMOVING AN ENTRY FROM THE RNT
- MAKING-UNKNOWN INVOLVES
  - MAKING A PREVIOUSLY VALID SEGMENT NUMBER INVALID
  - FREEING UP THAT SEGMENT NUMBER FOR FUTURE REASSIGNMENT

MANIPULATING THE ADDRESS AND NAME SPACES

- NOTES

  - INITIATING A REFERENCE NAME MAY TRIGGER THE MAKING-KNOWN OF A SEGMENT

  - TERMINATING A REFERENCE NAME MAY TRIGGER THE MAKING-UNKNOWN OF A SEGMENT

  - AN UNKNOWN SEGMENT CAN NOT HAVE A REFERENCE NAME

  - A KNOWN SEGMENT MAY HAVE A NULL REFERENCE NAME

  - PRESENCE IN THE RNT IMPLIES PRESENCE IN THE DSEG (AND KST)

MANIPULATING THE ADDRESS AND NAME SPACES

- TERMINATING SEGMENTS USING term_

  - term_$term_

    - call term_$term_ (dir_path, entryname, code);
      - REMOVES ALL REFERENCE NAMES FROM RNT
      - REMOVES SEGMENT FROM CALLER'S ADDRESS SPACE
      - REMOVES SEGMENT FROM COMBINED LINKAGE SECTION
      - UNSNAPS LNKS IN COMBINED LINKAGE QECTION THAT CONTAIN REFERENCES TO THE SEGMENT
      - USER SUPPLIES dir_path AND entryname
      - COMMAND INTERFACE: terminate (tm)

  - term_$seg_ptr

    - call term_$seg_ptr (seg_ptr, code);
      - LIKE term_$term_, BUT ACCEPTS A PTR TO SEGMENT
      - COMMAND INTERFACE: terminate_segno (tms)

  - term_$refname

    - call term_$refname (ref_name, code);
      - LIKE term_$term_, BUT ACCEPTS A REFERENCE NAME
      - COMMAND INTERFACE: terminate_refname (tmr)

- term_$single_refname
  - call term_$single_refname (ref_name, code);
    - REMOVES A SINGLE REFERENCE NAME FROM RNT
    - BEHAVES LIKE term_$refname (I.E. SEGMENT IS NOT MADE UNKNOWN) IFF REFNAME SPECIFIED WAS SEGMENT'S ONLY INITIATED REFNAME
    - COMMAND INTERFACE: terminate_single_refname (tmsr)

- term_$unsnap
  - call term_$unsnap (seg_ptr, code);
    - UNSNAPS LINKS ONLY
    - DOESN'T TERMINATE REFERENCE NAMES OR MAKE SEGMENT UNKNOWN
    - NO COMMAND LEVEL INTERFACE

MANIPULATING THE ADDRESS AND NAME SPACES

- initiate_file_
  - MAKES A SEGMENT KNOWN WITH A NULL REFERENCE NAME
  - (PREVIOUSLY DISCUSSED IN TOPIC 5)

- terminate_file_
  - TERMINATES A NULL REFERENCE NAME
  - (PREVIOUSLY DISCUSSED IN THIS TOPIC)

EXAMINING THE ADDRESS AND NAME SPACES

- hcs_$fs_get_path_name

  - call hcs_$fs_get_path_name (seg_ptr, dir_name, ldn, entryname, code);

  - GIVEN A POINTER TO A SEGMENT, RETURNS A PATHNAME FOR THE SEGMENT, WITH THE DIRECTORY AND ENTRYNAME PORTIONS SEPARATED (THE ENTRYNAME RETURNED IS THE PRIMARY NAME ON THE ENTRY)

- hcs_$fs_get_ref_name

  - call hcs_$fs_get_ref_name (seg_ptr, count, ref_name, code);

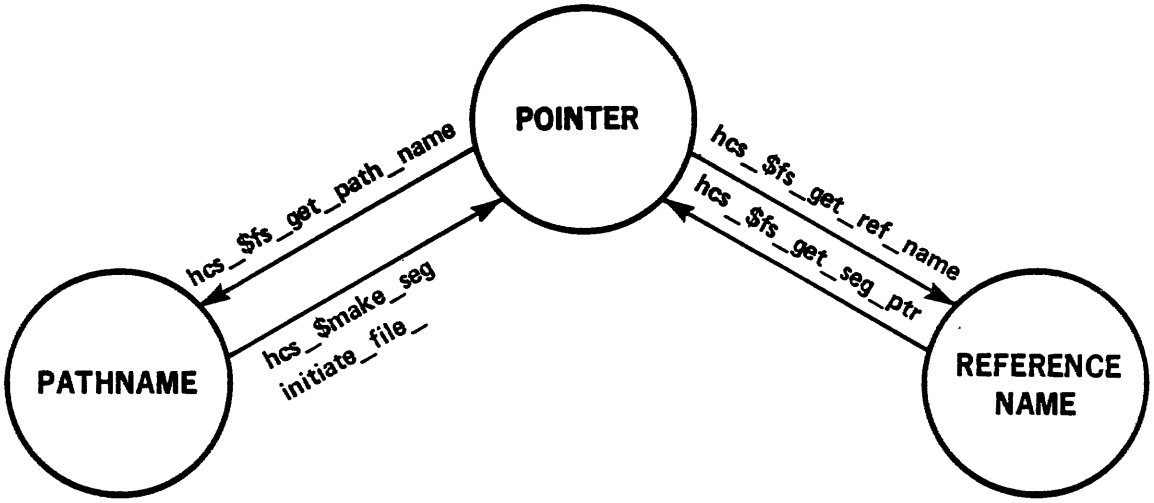
  - RETURNS A SPECIFIED (I.E., FIRST, SECOND, ETC.) REFERENCE NAME OF A SPECIFIED SEGMENT, GIVEN A POINTER TO THE SEGMENT

- hcs_$fs_get_seg_ptr

  - call hcs_$fs_get_seg_ptr (ref_name, seg_ptr, code);

  - GIVEN A REFERENCE NAME OF A SEGMENT, RETURNS A POINTER TO THE BASE OF THAT SEGMENT

PATHNAME, POINTER, REFERENCE NAME CONVERSION



YOU ARE NOW READY FOR WORKSHOP #8

TOPIC XII

Commands and Active Functions

OBJECTIVES:

Upon completion of this topic, students should be able to:

1. Describe the differences between a command and an active function.

2. Write a command which takes a varying number of arguments, validates them, and performs some task.

3. Write an active function which accepts a varying number of arguments, validates them, and returns an appropriate value.

4. Use Multics subroutines to report errors encountered during execution of a command or active function.

5. Use Multics subroutines to acquire and release temporary working storage.

6. Use the Multics clock and timer functions.

COMMANDS

CHARACTERISTICS OF A COMMAND

- A COMMAND PROCEDURE IS AN OBJECT PROGRAM WHICH IS DESIGNED TO BE INVOKED FROM COMMAND LEVEL

- A COMMAND PROCEDURE MUST OPERATE WITHIN STRICT OPERATIONAL LIMITATIONS, AND IT IS THESE LIMITATIONS THAT MAKE IT DIFFERENT FROM OTHER PROCEDURES

- MANY SYSTEM SUBROUTINES CALLED BY COMMAND PROCEDURES RETURN PL/I POINTER VALUES, THUS FORCING THE AUTHOR TO CODE THE COMMAND PROCEDURE IN PL/I

COMMANDS

DIFFERENCES BETWEEN A COMMAND AND A PROGRAM

- THE DIFFERENCES WHICH EXIST BETWEEN A COMMAND PROGRAM AND A REGULAR PROGRAM ARE DEFINED BY THE THREE RESTRICTIONS BELOW:

  - BECAUSE THE COMMAND IS CALLED BY THE MULTICS COMMAND PROCESSOR (OR A USER-DESIGNED COMMAND PROCESSOR)

    - INPUT ARGUMENTS ARE LIMITED TO 'nonvarying unaligned character strings'

    - HENCE, A COMMAND IS RESPONSIBLE FOR CONVERTING THESE STRINGS TO WHATEVER DATA TYPES ARE REQUIRED

  - A COMMAND CAN RECEIVE ONLY INPUT ARGUMENTS

    - THE COMMAND CANNOT CHANGE THE VALUE OF ANY OF THESE INPUT ARGUMENTS

  - THE COMMAND MUST BE PREPARED TO HANDLE AN ARBITRARY NUMBER OF ARGUMENTS - THERE ARE NO PARAMETER DECLARATIONS ALLOWED

    - IF THE COMMAND IS PASSED TOO MANY ARGUMENTS, IT MUST COMPLAIN AND ABORT (CONSIDER HOW THE SYSTEM HANDLES "pwd a")

  - OTHER RULES FOR MULTICS SYSTEM COMMANDS

    - USE com_err_ TO REPORT ERRORS

    - USE NO PL/I I/O (USE ioa_, iox_, AND command_query_)

REPORTING ERRORS

- WHEN A COMMAND PROCEDURE DETECTS SOME ERROR, IT IS RESPONSIBLE FOR REPORTING IT TO THE USER IN A STANDARD FASHION:

  - com_err_

    - THE PRINCIPAL SUBROUTINE USED BY COMMANDS FOR PRINTING ERROR MESSAGES

    - IT IS GENERALLY CALLED WITH A NONZERO STATUS CODE TO REPORT SOMETHING UNUSUAL

    - IT MAY ALSO BE CALLED WITH A CODE OF 0 TO REPORT AN ERROR NOT ASSOCIATED WITH A STATUS CODE

    - declare com_err_ entry options(variable);

      call com_err_ (code, caller, control_string, arg1, ..., argN);

      - control_string IS AN OPTIONAL ioa_ SUBROUTINE CONTROL STRING (INPUT)

      - arg1, ..., argN ARE ioa_ SUBROUTINE ARGUMENTS TO BE SUBSTITUTED INTO THE control_string (INPUT)

- THE ERROR MESSAGE PREPARED BY com_err_ HAS THE FORM:
  - caller: system_message user_message
    - FOR SYSTEM COMMANDS caller IS THE NAME OF THE PROGRAM DETECTING THE ERROR
  - EXAMPLE: (IF code = error_table_$wrong_no_of_args AND nargs = 5)
    - PL/I STATEMENT:

```
call com_err_ (code, "sample_command",
               "^/You furnished ^d args.", nargs);
```

    - RESULT:

```
sample_command:  Wrong number of arguments supplied.
You furnished 5 args.
```

  - IF CODE = 0 ONLY A user_message IS PRINTED

- IN WRITING COMMAND PROCEDURES NO LANGUAGE LEVEL I/O STATEMENTS ARE EVER USED

- STANDARD INPUT/OUTPUT IS DONE USING THE FOLLOWING SUBROUTINES:

  - ioa_
    - USED FOR FORMATTING A CHARACTER STRING

  - iox_
    - THE SUBROUTINE-LEVEL INTERFACE TO THE MULTICS I/O SYSTEM

  - command_query_
    - THE STANDARD SYSTEM PROCEDURE INVOKED TO ASK THE USER A QUESTION AND OBTAIN AN ANSWER
    - IT PRINTS THE QUESTION ON THE USER'S TERMINAL, AND THEN READS THE 'user_input' SWITCH TO OBTAIN THE ANSWER
    - declare command_query_ entry options(variable);

      call command_query_ (ptr, answer, caller, control_string, arg1, ..., argN);

      - ptr POINTS TO THE query_info STRUCTURE DESCRIBED ON THE FOLLOWING PAGE (INPUT)

```
/* BEGIN INCLUDE FILE query_info.incl.pl1 TAC June 1, 1973 */
/* Renamed to query_info.incl.pl1
                and cp_escape_control added, 08/10/78 WOS */
/* version number changed to 4, 08/10/78 WOS */
/* Version 5 adds explanation_(ptr len) 05/08/81 S. Herbst */
/* Version 6 adds literal_sw, prompt_after_explanation switch
                                        12715/82 S. Herbst */

dcl 1 query_info aligned,
        /* argument structure for command_query_ call */
    2 version fixed bin,
        /* version of this structure - must be set, see below */
    2 switches aligned,
        /* various bit switch values */
      3 yes_or_no_sw bit (1) unaligned init ("0"b),
          /* not a yes-or-no question, by default */
      3 suppress_name_sw bit (1) unaligned init ("0"b),
          /* do not suppress command name */
      3 cp_escape_control bit (2) unaligned init ("00"b),
          /* obey static default value */
          /* "01" -> invalid, "10" -> don't allow, "11" -> allow */
      3 suppress_spacing bit (1) unaligned init ("0"b),
          /* whether to print extra spacing */
      3 literal_sw bit (1) unaligned init ("0"b),
          /* ON => do not strip leading/trailing white space */
      3 prompt_after_explanation bit (1) unaligned init ("0"b),
          /* ON => repeat question after explanation */
      3 padding bit (29) unaligned init (""b),
          /* pads it out to t word */
    2 status_code fixed bin (35) init (0),
        /* query not prompted by any error, ay default */
    2 query_code fixed bin (35) init (0),
        /* currently has no meaning */

/*  Limit of data defined for version 2 */

    2 question_iocbp ptr init (null ()),
        /* IO switch to write question */
    2 answer_iocbp ptr init (null ()),
        /* IO switch to read answer */
    2 repeat_time fixed bin (71) init (0),
        /* repeat question every N seconds if no answer */
        /* minimum of 30 seconds required for repeat */
        /* otherwise, no repeat will occur */

/* Limit of data defined for version 4 */

    2 explanation_ptr ptr init (null ()),
        /* explanation of question to be printed if */
    2 explanation_len fixed bin (21) init (0);
        /* user answers "?" (disabled if ptr=null or len=0) */
```

```
dcl  query_info_version_3 fixed bin int static
                          options (constant) init (3);
dcl  query_info_version_4 fixed bin int static
                          options (constant) init (4);
dcl  query_info_version_5 fixed bin int static
                          options (constant) init (5);
dcl  query_info_version_6 fixed bin int static
                          options (constant) init (6);
       /* the current version number */

/* END INCLUDE FILE query_info.incl.pl1 */
```

OTHER SUBROUTINES USED IN WRITING COMMANDS

- cu_

  - USED TO MANIPULATE THE COMMAND ENVIRONMENT IN FUNCTIONS LIKE:

    - SETTING THE READY MESSAGE
    - CALLING THE COMMAND PROCESSOR
    - CHANGING THE COMMAND PROCESSOR
    - EXAMINING STACK FRAMES

OTHER SUBROUTINES USED IN WRITING COMMANDS

- THE FOLLOWING ENTRIES ARE USED TO OBTAIN THE ARGUMENTS PASSED TO THE COMMAND

  - cu_$arg_count

    - call cu_$arg_count (arg_count, code);

    - USED TO DETERMINE THE NUMBER OF ARGUMENTS SUPPLIED WHEN THE PROCEDURE WAS CALLED

  - cu_$arg_ptr

    - call cu_$arg_ptr (arg_no, arg_ptr, arg_len, code);

    - RETURNS A POINTER TO AND THE LENGTH OF ONE OF THE ARGUMENTS

      - arg_no IS AN INTEGER SPECIFYING THE NUMBER OF THE DESIRED ARGUMENT (INPUT)

    - NOTE THAT A BASED VARIABLE IS NORMALLY USED FOR INPUT ARGUMENTS AND IS DECLARED AS FOLLOWS:

      - declare argument char(arg_len) based(arg_ptr);

- EXAMPLES

```
sample_command: proc;

dcl   cu_$arg_count entry(fixed bin, fixed bin(35));
dcl   nargs fixed bin;
dcl   error_table_$wrong_no_of_args fixed bin(35) external;
dcl   com_err_ entry options(variable);
dcl   code fixed bin(35);
       . . .
      call cu_$arg_count (nargs, code);
      if nargs ^= 0
      then do;
             call com_err_(error_table_$wrong_no_of_args,
                           "sample_command");
             return;
           end /* then do */;
       . . .
```

```
sample_command2: proc;

dcl   cu_$arg_ptr entry (fixed bin,ptr,fixed bin(21),fixed bin(35));
dcl   argument char(arg_len) based(arg_ptr);
dcl   arg_len fixed bin(21);
dcl   arg_ptr ptr;
dcl   code fixed bin(35);
dcl (com_err_, ioa_) entry options(variable);
       . . .
      call cu_$arg_ptr (1, arg_ptr, arg_len, code);
      if code ^= 0
      then do;
             call com_err_ (code, "sample_command2");
             return;
           end /* then do */;
      call ioa_("First argument is ^a",argument);
      . . .
```

OTHER SUBROUTINES USED IN WRITING COMMANDS

- THE FOLLOWING SUBROUTINES ARE USED FOR ARGUMENT CONVERSION:

  - expand_pathname_

    - call expand_pathname_ (pathname, dirname, entryname, code);

    - PREVIOUSLY DISCUSSED IN TOPICS 5 AND 10

    - NOTE THAT SOME CRITICAL MULTICS SUBROUTINES REQUIRE A PATHNAME ARGUMENT SPECIFIED IN TWO PORTIONS, THE DIRECTORY PATHNAME AND THE ENTRYNAME, AND THIS IS ONE OF THE MAIN REASONS expand_pathname_ IS AVAILABLE

  - cv_ptr_

    - ptr_value = cv_ptr_ (vptr, code);

    - THIS FUNCTION CONVERTS A VIRTUAL POINTER TO A POINTER VALUE (A VIRTUAL POINTER IS A CHARACTER-STRING REPRESENTATION OF A POINTER VALUE, SUCH AS "foo$bar" OR ">udd>PROJ>PERS>seg|1200")

- OTHER CONVERSION SUBROUTINES AND FUNCTIONS
  - cv_bin_
    - cv_bin_$dec
    - cv_bin_$oct
  - cv_dec_, cv_dec_check_
  - cv_oct_, cv_oct_check_
  - cv_hex_, cv_hex_check_
  - cv_float_
  - cv_float_double_
  - cv_ptr_$terminate
  - cv_entry_
  - cv_mode_
  - cv_dir_mode_
  - cv_userid_
  - cv_error_
    - cv_error_$name

This Page Intentionally left Blank

```
how_long: proc;

dcl  cu_$arg_count     entry (fixed bin, fixed bin (35));
dcl  cu_$arg_ptr       entry (fixed bin, ptr, fixed bin(21), fixed bin (35));
dcl  expand_pathname_  entry (char (*), char (*), char (*), fixed bin (35));
dcl  hcs_$status_minf  entry (char(*), char(*), fixed bin(1), fixed bin(2),
                          fixed bin(24), fixed bin(35));
dcl  long              bit (1) init ("0"b);
dcl  arg               char (argl) based (argp);
dcl (i, nargs)         fixed bin;
dcl  argl              fixed bin(21);
dcl  argp              ptr;
dcl  type              fixed bin (2);
dcl  code              fixed bin (35);
dcl  dir               char (168);
dcl  entry             char (32);
dcl (com_err_,
     ioa_)             entry options (variable);
dcl  ME                char (8) static init ("how_long") options (constant);
dcl  bc                fixed bin (24);
dcl  null              builtin;
dcl  error_table_$wrong_no_of_args fixed bin(35) external;

/* check number of args */

call cu_$arg_count (nargs, code);
if (nargs < 1) | (nargs > 2)
then do;
        call com_err_ (error_table_$wrong_no_of_args, ME);
        return;
     end /* then do */;

/* evaluate args */

do i = 1 to nargs;
   call cu_$arg_ptr (i, argp, argl, code);

   if i = 1
   then do;
           call expand_pathname_ (arg, dir, entry, code);
           if code ^= 0
           then do;
                   call com_err_ (code, ME);
                   return;
                end /* then do */;
           call hcs_$status_minf (dir, entry, 1, type, bc, code);
           if code ^= 0
           then do;
                   call com_err_ (code, ME);
                   return;
                end /* then do */;
```

```
            bc = bc/36;
         end /* then do */;
      else do;
            /* second arg must be -long or -lg */
            if (arg = "-long") | (arg = "-lg")
            then long = "1"b;
            else do;
                  call com_err_ (0, ME, "Control arg must be -long or -lg"
                  return;
               end /* else do */;
         end /* else do */;
   end /* do i */;

   call ioa_("^[Number of words for ^a>^a is ^;^2s^]^i", long, dir, entry, bc)

   end /* how_long */;


   r 14:03 0.197 18

! how_long
   how_long: Wrong number of arguments supplied.
   r 14:04 0.183 11

! how_long how_long
   660
   r 14:04 0.105 0

! how_long how_long.pl1 -lg
   Number of words for >user_dir_dir>MED>Jackson>15c>how_long.pl1 is 544
   r 14:04 0.088 0

! how_long how_long.pl1 -short
   how_long: Control arg must be -long or -lg
   r 14:04 0.143 1
```

ACTIVE FUNCTIONS

- AN ACTIVE FUNCTION RETURNS A CHAR VARYING VALUE TO THE COMMAND PROCESSOR FOR SUBSTITUTION INTO THE COMMAND LINE

  - IT IS CALLED BY THE COMMAND PROCESSOR FOR THE PURPOSE OF RETURNING A VALUE TO THE COMMAND PROCESSOR

  - THE COMMAND PROCESSOR MUST PREPARE A LOCATION FOR THE RETURNED VALUE

  - THE ACTIVE FUNCTION MUST KNOW THIS LOCATION IN ORDER TO RETURN A VALUE

- AN ACTIVE FUNCTION DIFFERS FROM A STANDARD PROCEDURE IN THE THREE WAYS SPECIFIED FOR COMMANDS (TAKES ONLY CHARACTER-STRING ARGUMENTS, HANDLES ONLY INPUT ARGUMENTS, TAKES A VARYING NUMBER OF ARGUMENTS) AND HAS ONE ADDITIONAL DIFFERENCE:

  - AN ACTIVE FUNCTION MUST RETURN A VARYING CHARACTER-STRING ARGUMENT TO THE COMMAND PROCESSOR IN A LOCATION SPECIFIED BY THE COMMAND PROCESSOR

- A COMMAND PROCEDURE CAN BE WRITTEN TO IMPLEMENT EITHER A COMMAND OR AN ACTIVE FUNCTION

- SUCH A PROCEDURE'S EXECUTION DEPENDS ON THE MANNER IN WHICH IT WAS INVOKED

ACTIVE FUNCTIONS

SUBROUTINES USED FOR WRITING ACTIVE FUNCTIONS

- THE SUBROUTINES USED FOR WRITING AN ACTIVE FUNCTION MUST BE ABLE TO DETERMINE TWO VERY IMPORTANT THINGS:
  - THE LOCATION INTO WHICH IT SHOULD PLACE ITS RETURN VALUE
  - WHETHER OR NOT IT WAS INVOKED AS A ACTIVE FUNCTION

- cu_$af_arg_count
  - call cu_$af_arg_count (nargs, code);
  - RETURNS THE NUMBER OF INPUT ARGUMENTS
  - IF THE CALLER WAS NOT INVOKED AS AN ACTIVE FUNCTION, A NON-ZERO STATUS CODE IS RETURNED (error_table_$not_act_fcn)

- cu_$af_arg_ptr
  - call cu_$af_arg_ptr (arg_no, arg_ptr, arg_len, code);
  - OPERATES LIKE cu_$arg_ptr EXCEPT THAT IT RETURNS A NULL arg_ptr IF IT WAS NOT CALLED AS AN ACTIVE FUNCTION
  - USUALLY USED IN WRITING PROGRAMS THAT CAN ONLY BE CALLED AS ACTIVE FUNCTIONS

SUBROUTINES USED FOR WRITING ACTIVE FUNCTIONS

- cu_$af_return_arg

  - call cu_$af_return_arg (nargs, rtn_string_ptr, max_length, code);

  - PERFORMS THE JOB OF cu$af_arg_count AND

  - MAKES THE ACTIVE FUNCTION'S RETURN ARGUMENT AVAILABLE

    - rtn_string_ptr IS A POINTER TO THE VARYING STRING RETURN ARGUMENT OF THE ACTIVE FUNCTION (OUTPUT)

    - max_length IS THE MAXIMUM LENGTH OF THE VARYING STRING POINTED TO BY rtn_string_ptr (OUTPUT)

    - IF THE CALLER WAS NOT INVOKED AS AN ACTIVE FUNCTION, A NON-ZERO STATUS CODE IS RETURNED (error_table_$not_act_fcn)

    - NOTE THAT THE ACTIVE FUNCTION DECLARES ITS RETURN ARGUMENT AS FOLLOWS:

```
declare return_string char (max_length) varying
                        based (rtn_string_ptr);
```

ACTIVE FUNCTIONS

REPORTING ERRORS

- AN ACTIVE FUNCTION USES A DIFFERENT SUBROUTINE FOR REPORTING ERRORS TO THE USER:

  - active_fnc_err_

    - CALLED BY AN ACTIVE FUNCTION WHEN IT DETECTS AN ERROR

    - FORMATS AN ERROR MESSAGE MUCH LIKE com_err_ AND THEN SIGNALS THE 'active_function_error' CONDITION

    - USAGE

      - declare active_fnc_err_ entry options(variable);

      - call active_fnc_err_ (code, caller, control_string, arg1, ..., argN);

      - THE USAGE IS SIMILAR IN ALL RESPECTS TO com_err_

AN ACTIVE FUNCTION EXAMPLE

```
me: proc;

dcl  cu_$af_return_arg entry (fixed bin, ptr, fixed bin(21), fixed bin (35));
dcl  nargs             fixed bin;
dcl  return_arg        char (rslength) varying based (rsptr);
dcl  rslength          fixed bin (21);
dcl  rsptr             ptr;
dcl  code              fixed bin (35);
dcl  user_info_        entry (char (*), char (*), char (*));
dcl (active_fnc_err_,
     com_err_)         entry options (variable);
dcl  error_table_$wrong_no_of_args fixed bin (35) external;
dcl  person_id         char (22);
dcl  project_id        char (9);
dcl  acct              char (32);

/* DETERMINE IF INVOKED AS ACTIVE FUNCTION */

call cu_$af_return_arg (nargs, rsptr, rslength, code);
if code ^= 0
then do;
        call com_err_ (code, "me");
        return;
     end /* then do */;

if nargs ^= 0
then do;
        call active_fnc_err_(error_table_$wrong_no_of_args,"me");
        return;
     end /* then do */;

/* SO FAR, SO GOOD - GET PERSON_ID */

call user_info_ (person_id, project_id, acct);
return_arg = person_id;

end /* me */;
```

AN ACTIVE FUNCTION EXAMPLE

```
  r 15:19 0.143 0

! me
  me: This active function cannot be invoked as a command.
  r 15:19 0.197 5

! who [me]
  Jackson.MED

  r 15:20 0.524 5

! who [me Jackson]
  me: Wrong number of arguments supplied.

  Error:  Bad call to active function me
  r 15:20 0.206 9 level 2
```

COMMANDS AND ACTIVE FUNCTIONS

- THE SUBROUTINES DISCUSSED PREVIOUSLY ARE USED IN WRITING PROCEDURES THAT MAY BE CALLED AS BOTH COMMANDS AND ACTIVE FUNCTIONS

- THE FOLLOWING SUMMARIZES THE IDIOSYNCRASIES TO BE CONSIDERED IN CHOOSING APPROPRIATE SUBROUTINES

cu_ ENTRY	COMMAND	ACT. FUNC.	COMMENTS
arg_count	X	X	IF INVOKED AS AN ACTIVE FUNCTION COUNT INCLUDES RETURN ARGUMENT
arg_ptr	X	X	
af_arg_count	X	X	COUNT EQUALS INPUT ARGUMENTS ONLY
af_arg_ptr		X	NULL arg_ptr IF INVOKED AS A COMMAND
af_return_arg	X	X	COUNT EQUALS INPUT ARGUMENTS ONLY NULL rtn_ptr IF INVOKED AS A COMMAND

- IT IS ALWAYS POSSIBLE TO WRITE ANY COMMAND OR ACTIVE FUNCTION USING ONLY THE TWO ENTRY POINTS, cu_$af_return_arg AND cu_$arg_ptr

AN EXAMPLE OF A COMMAND/ACTIVE FUNCTION

```
how_long_both: proc;

dcl  expand_pathname_ entry (char (*), char (*), char (*), fixed bin (35));
dcl  cu_$arg_ptr entry (fixed bin, ptr, fixed bin(21), fixed bin(35));
dcl  cu_$af_return_arg entry(fixed bin, ptr, fixed bin(21), fixed bin (35));
dcl  active_fnc_err_ entry options (variable);
dcl  ncs_$status_minf entry (char(*), char(*), fixed bin(1), fixed bin(2),
                             fixed bin(24), fixed bin(35));
dcl  long bit (1) init ("0"b);
dcl  arg char (argl) based (argp);
dcl  complain entry variable options (variable);
dcl  af bit (1) init ("0"b);
dcl  return_string char (rslength) var based (rsptr);
dcl  rslength fixed bin (21);
dcl  rsptr ptr;
dcl (i, nargs) fixed bin;
dcl  argl fixed bin (21);
dcl  argp ptr;
dcl  type fixed bin (2);
dcl  code fixed bin (35);
dcl  dir char (168);
dcl  entry char (32);
dcl (com_err_, ioa_) entry options (variable);
dcl  ME char (13) static init ("how_long_both") options (constant);
dcl  bc fixed bin (24);
dcl  error_table_$wrong_no_of_args fixed bin(35) external;

/* check number of args */

call cu_$af_return_arg (nargs, rsptr, rslength, code);

/* command or active function invocation??? */

if code = 0
then do;
        af = "1"b;
        complain = active_fnc_err_;
     end /* then do */;
else complain = com_err_;

if (nargs < 1) | (nargs > 2)
then do;
        call complain (error_table_$wrong_no_of_args, ME);
        return;
     end /* then do */;

/* evaluate args */

do i = 1 to nargs;
   call cu_$arg_ptr (i, argp, argl, code);
```

AN EXAMPLE OF A COMMAND/ACTIVE FUNCTION

```
     /* relative pathname argument */

     if i = 1
     then do;
             call expand_pathname_ (arg, dir, entry, code);
             if code ^= 0
             then do;
                     call complain (code, ME);
                     return;
                  end /* then do */;

             call hcs_$status_minf (dir, entry, 1, type, bc, code);
             if code ^= 0
             then do;
                     call complain (code, ME);
                     return;
                  end /* the do */;
             bc = bc/36;
          end /* then do */;
     else do;

             /* second arg must be -long or -lg */

             if (arg = "-long") | (arg = "-lg")
             then long = "1"b;
             else do;
                     call complain (0, ME, "Control arg must be -long or -lg");
                     return;
                  end /* else do */;
          end /* else do */;

end /* do i */;

if af
then do;
        return_string = bc;
        return;
     end /* then do */;

call ioa_("^[Number of words for ^a>^a is ^;^2s^]^i", long, dir, entry, bc);

end /* how_long_both */;
```

```
  r 15:59 0.284 7

! how_long_both
  how_long_both: Wrong number of arguments supplied.
  r 15:59 0.152 11

! how_long_both foo -lg
  how_long_both: Entry not found.
  r 16:00 0.118 0

! how_long_both how_long_both
  776
  r 16:00 0.076 0

! how_long_both how_long_both -long
  Number of words for >user_dir_dir>MED>Jackson>f15c>how_long_both is 776
  r 16:01 0.098 0

! octal [how_long_both how_long_both]
  1410
  r 16:01 0.196 6

! octal [how_long_both]
  how_long_both: Wrong number of arguments supplied.

  Error:  Bad call to active function how_long_both
  r 16:01 0.169 7 level 2
```

- user_info_

  - RETURNS INFORMATION CONCERNING A USER'S LOGIN SESSION (ALL ARGUMENTS ARE OUTPUT ARGUMENTS)

    ```
    call user_info_ (person_id, project_id, acct);
    ```

  - OTHER ENTRY POINTS:

    ```
    call user_info_$absentee_queue (queue);

    call user_info_$absentee_request_id (request_id);

    call user_info_$absin (path);

    call user_info_$absout (path);

    call user_info_$attributes (attr);

    call user_info_$homedir (hdir);

    call user_info_$limits (mlim, clim, cdate, crf, shlim, msp,
                            csp, shsp);

    call user_info_$load_ctl_info (group, stby, preempt_time,
                                   weight);

    call user_info_$login_arg_count (count, max_length,
                                     total_length);

    call user_info_$login_arg_ptr (arg_no, arg_ptr, arg_len,
                                   code);
    ```

OTHER USEFUL SUBROUTINES

```
call user_info_$login_data (person_id, project_id, acct,
                            anon, stby, weight, time_login,
                            login_word);

call user_info_$logout_data (logout_channel, logout_pid);

call user_info_$outer_module (om);

call user_info_$process_type (process_type);

call user_info_$responder (resp);

call user_info_$rs_name (rs_name);

call user_info_$rs_number (rs_number);

call user_info_$service_type (type);

call user_info_$terminal_data (id_code, type, channel,
                               line_type, charge_type);

call user_info_$usage_data (nproc, old_cpu, time_login,
                            time_create, old_mem,
                            old_io_cps);

call user_info_$whoami (person_id, project_id, acct);
```

- value_

  - READS AND MAINTAINS VALUE SEGMENTS CONTAINING NAME-VALUE PAIRS ACROSS PROCESS BOUNDARIES

  - CREATING A VALUE SEGMENT

    - CREATE A SEGMENT WITH A SUFFIX OF .value

    - call value_$init_seg (seg_ptr, seg_type, remote_area_ptr, seg_size, code);

  - DEFAULT VALUE SEGMENT IS [home_dir]>[user_name].value

    - call value_$set_path (path, create_sw, code);

    - call value_$get_path (path, code);

  - CREATING AND MAINTAINING NAME-VALUE PAIRS

    - call value_$set (seg_ptr, switches, name, new_value, old_value, code);

    - call value_$test_and_set (seg_ptr, switches, name, new_value, old_value, code);

    - call value_$get (seg_ptr, switches, name, value_arg, code);

    - call value_$list (set_ptr, switches,match_info_ptr, area_ptr, value_list_info_ptr, code);

    - call value_$defined (seg_ptr, switches, name, code);

    - call value_$delete (seg_ptr, switches, name, code);

YOU ARE NOW READY FOR WORKSHOP #9

APPENDIX W

Workshops

WORKSHOP ONE

Controlled Variables and 'isub' Defining

1. Write a procedure called 'allocate_array.pl1' that will ask the user for the size of one dimensional fixed bin (17) arrays he/she wishes to allocate. For example, if the user provides the number 7, your program is to allocate an array with 7 fixed bin (17) elements.

   The program should loop, repeatedly asking for the size of the next array, allocating that array and then initializing all elements of that array to the current allocation level (i.e., the first array would be initialized to 1, the second array would be initialized to 2, etc.). Use the 'allocation' builtin to determine the depth.

   The program should continue allocating and initializing until the user responds with zero (0). Again using the 'allocation' builtin to determine the allocation depth, it should then free all the allocated arrays, printing each array just before freeing it.

   Test your program asking for arrays of size 1, 2, 3, and 4. Observe the order in which the arrays are freed.

2. The segment >udd>MEDclass>F15C>s1>printit.fortran contains a fortran subroutine that accepts a 2 by 3 array as an argument and prints it out a row at a time.

   Copy the segment, print it, compile it and write a PL/I procedure called 'call_fortran.pl1' declaring a 2 by 3 array and the 3 by 2 transpose of this array (use isubs). The program should:

   a. Initialize the 2 by 3 array as follows:

      1 2 3
      4 5 6

   b. Call the fortran subroutine, passing to it the untransposed array.

   c. Call the fortran subroutine, passing to it the transposed array.

Note:

1) 'printit' must be declared an entry, and since it will be passed both a 2 by 3 and a 3 by 2 array, its descriptor must use the star convention (dim(*,*)).

2) The elements of the array should be declared fixed bin (35) since that is the data type for fortran integers.

3) The final compilation of the PL/I program will still have a "by value" warning since 'isub' defined variables are always passed by value. Recall this means that the called procedure will not be able to change the variable passed to it. How can this warning be avoided? That is, how could the array be passed by reference?

4) When you compile the PL/I program with the table option (the default), you will receive a warning that the transposed array will not appear in the symbol table.

WORKSHOP TWO

Based Variables and Areas

This workshop has three parts. Be sure you understand the mechanism used in parts 1 and 2 (based variables), since they form the basis for workshop three and the remainder of this course.

1. The following declarations are in the segment >udd>MEDclass>F15C>s1>include>w2.incl.pl1.

```
/* Begin w2.incl.pl1 */

dcl string           char (10) varying;
dcl 1 string_parts   based (addr (string)),
      2 length       fixed bin (35),
      2 characters   char (10);

dcl number           float binary;
dcl 1 float_num      based (addr (number)),
      2 sign         bit (1) unal,
      2 exponent     bit (7) unal,
      2 m_sign       bit (1) unal,
      2 mantissa     bit (27) unal;

/* End w2.incl.pl1 */
```

   Write a short program that enters data into the two BASE variables (string and number) and then prints out the values in the BASED (overlay) variables in order to see exactly how 'char varying' and 'float binary' numbers are stored. (Use put data.)

2. Change your working directory to >udd>MEDclass>F15C>s1. Print the segment get_message.pl1. Execute the corresponding object segment and follow the directions given in the message.

3. In your working directory create an area named AREA (all caps) using the create_area command. In the segment, >udd>MEDclass>F15C>s1>fill_area.pl1, is a program that allocates 2 numbers in that area. Print the program and make sure you understand what it is doing. Execute the object segment. Use the dump_segment (ds) command to look at your AREA segment. Notice how the pointer values printed by the program correspond to locations in the segment. Also notice the extra area manager information in the segment.

WORKSHOP THREE

Gaining Direct Access to a Segment

The segment, >udd>MEDclass>F15C>s1>invoices, contains invoices for a number of different vendors. At the base of the segment is a header. The remainder of the segment is a series of linked structures, each one representing a single invoice for a particular vendor. The declaration to be used for the linked structure is:

```
dcl 1 invoice based (p),
      2 next           bit (18),
      2 invoice_number dec (3),
      2 vendor_number  dec (3),
      2 charge         fixed dec (8,2);
```

The structure member, invoice.next, is a non-standard offset (word offset from the base of the segment) indicating the location of the next structure in the linked list.

Write a program called 'get_invoices.pl1'. Your program should prompt the user for a vendor number (3 digits) and then print out all invoice numbers and the corresponding charges belonging to that vendor.

Actually, the segment does not contain just one linked list. There are, in fact, 10 linked lists below the header. The header is used to determine which list is to be searched for that particular vendor. The declaration to be used for the header is:

```
dcl 1 invoice_file_header  based (seg_ptr),
      2 number_of_invoices fixed bin,
      2 hash_table (0:9)   bit (18) unal;
```

The hash table is made up of 10 non-standard offsets. Each offset points to the start of one of the linked lists of invoice structures. Which linked list a particular vendor is found in is determined by the last digit in the vendor number. For example, invoices for vendor 357 would be in the list pointed to by 'hash_table(7)'.

Thus, when a user gives you a vendor number you must overlay the header structure at the base of the segment and get the offset for the start of the appropriate linked list. Then you must get a pointer to the start of the linked list and move the invoice structure down the list checking for the appropriate vendor. If the vendor matches, print out the invoice number and the charge. Continue scanning the list until you reach the end. The last invoice in any list is indicated by invoice.next = "0"b.

As an example, to find invoices for vendor 357, the statement p = ptr(seg_ptr,hash_table(7)) would generate a pointer 'p' which locates the first invoice for a vendor with low order digit 7. The vendor number for this invoice can be compared to 357, and printed out if matched. Then, the pointer p could be adjusted to the next invoice in this list by coding the statement p = ptr(seg_ptr, p ->next) and so on.

Test your program by printing out the invoice number and charges of all invoices for vendor number 029.

You may wish to use the following declarations which are in the segment, >udd>MEDclass>F15C>s1>include>w3.incl.pl1.

```
/* Begin w3.incl.pl1 */

dcl  initiate_file_ entry (char (*), char (*), bit (*), pointer,
                           fixed bin (24), fixed bin (35));
dcl  code           fixed bin (35);
dcl  bit_count      fixed bin (24);
dcl  seg_ptr        ptr;
dcl  p              ptr;

dcl  1 invoice_file_header  based (seg_ptr),
       2 number_of_invoices fixed bin,
       2 hash_table (0:9)   bit (18) unaligned;

dcl  1 invoice          based (p) aligned,
       2 next           bit (18),
       2 invoice_number dec (3),
       2 vendor_number  dec (3),
       2 charge         fixed dec (8,2);

dcl  com_err_       entry options (variable);
dcl (sysin,
     sysprint)      file;

/* End w3.incl.pl1 */
```

For the more curious, you may wish to study
>udd>MEDclass>F15C>s1>set_up>put_invoice.pl1.

WORKSHOP FOUR

The Multics Condition Handling Mechanism

1. Print the segment >udd>MEDclass>F15C>s1>test_any_other.pl1 (tao.pl1) and execute the corresponding object segment.

   Examine your user stack using the 'stack' request of 'probe'. Notice where, on the stack, the program you just executed is compared to the 'wall' laid down by default error handler.

   Using the 'signal' command, execute the following commands: "signal zerodivide", "signal any_other", "signal finish", "signal program_interrupt". How do you explain the difference in these four cases?

   Note: the above program is not well behaved in that it should have continued to signal the 'finish' condition.

   BE SURE TO DO A 'release -all' BEFORE PROCEEDING!!!

2. Print the segment >udd>MEDclass>F15C>s1>test_cleanup.pl1 (tcu.pl1) and execute the corresponding object segment TWO times. BE SURE YOU EXECUTE IT AT LEAST TWO TIMES (more than two won't hurt, but is wasteful).

   Examine the user stack using the 'stack' request of 'probe'. Notice the numerous occurrences of 'test_cleanup' on the stack. Now examine the stack using the 'trace_stack' (ts) command. Notice the 'cleanup' handlers in several stack frames. (While you are at it, also notice that 'initialize_process_' and 'default_error_handler_' have only one condition handler.)

   Execute a "release -all". Can you explain what happened?

3. Print the segment >udd>MEDclass>F15C>s1>test_finish_1.pl1 (tf1.pl1) and execute the corresponding object segment AT LEAST THREE TIMES.

   Signal the finish condition.

   Do a "release -all" and then repeat the above procedure using >udd>MEDclass>F15C>s1>test_finish_2.pl1 (tf2.pl1).

WORKSHOP FIVE

IOCB structure

1. Print the segment >udd>MEDclass>F15C>s1>examine_iocb.pl1 and read it carefully to see what it does.

2. Execute the print_attach_table (pat) command to examine the switches currently attached.

3. While in your own directory, execute the following command lines:

```
io_call attach zoo vfile_ zoo
io_call open zoo stream_output
pat
```

   Now execute the program >udd>MEDclass>F15C>s1>examine_iocb and carefully examine the results. Notice that all pointers and entry points printed are in one of 3 segments.

4. Recall that the list_reference_names (lrn) command, if given a segment number, will return the pathname and reference names of that segment. Use this command to determine the three segments whose numbers were found in the IOCB. Notice especially which entries in the IOCB point to iox_ and which point to the I/O module, vfile_. Do these make sense, considering the file is opened for stream output?

5. Execute the command line, 'io_call close zoo'. Again execute the 'pat' command. Run the program, examine_iocb, again and notice the different results. Can you explain what happened? If not, ask your instructor.

6. Now that you have looked directly at an iocb using an overlay, you should try using the command that gives you the same information. Execute the command line 'io_call print_iocb zoo'.

7. Using 'io_call print_iocb <switch>' one can easily look at the contents of an iocb. Try the following: delete the segment zoo, and then use io_call to open zoo "keyed_sequential_output" and to display the contents of the iocb.

WORKSHOP SIX

Multics I/O Workshop

Write a PL/I procedure called "lucky_number.pl1" which prompts the user for a 6 digit number, and uses that as a key into an indexed file of lucky numbers. The file of numbers is in the segment:

- >udd>MEDclass>F15C>s1>lucky_nos

The data records are 32 characters or less in length.

Display the records. Do not use any language-level I/O. Use only iox_ and ioa_ calls in your program.

Test your program with the numbers 780101, 780116, and 771225.

You may wish to use the following declarations which are in the segment, >udd>MEDclass>F15C>s1>include>w6.incl.pl1

```
/* Begin w6.incl.pl1 */

dcl iox_$attach_name entry (char (*), ptr, char (*), ptr,
                            fixed bin (35));
dcl iox_$close       entry (ptr, fixed bin (35));
dcl iox_$detach_iocb entry (ptr, fixed bin (35));
dcl iox_$open        entry (ptr, fixed bin, bit (1) aligned,
                            fixed bin(35));
dcl iox_$read_record entry (ptr, ptr, fixed bin (21),
                            fixed bin (21), fixed bin (35));
dcl iox_$seek_key    entry (ptr, char (256) varying,
                            fixed bin (21), fixed bin (35));
dcl iox_$get_line    entry (ptr, ptr, fixed bin (21),
                            fixed bin (21), fixed bin (35));
dcl iox_$user_input  external static ptr;
dcl (ioa_,
     com_err_)       entry options (variable);
dcl error_table_$no_record fixed bin (35) external;
dcl code             fixed bin (35);
dcl buff             char (32);
dcl buff_ptr         ptr;
dcl rec_len          fixed bin (21);
dcl iocb_ptr         ptr;
dcl n_read           fixed bin (21);
dcl number           char (256) varying;
dcl cleanup          condition;
dcl (addr,
     null,
     substr)         builtin;

/* End w6.incl.pl1 */
```

WORKSHOP SIX

Be sure that you provide an 'on unit' for the 'cleanup' condition. Also, you should check for the code, error_table_$no_record (indicating an invalid key), after doing the seek_key.

WORKSHOP SEVEN

A Storage System Workshop

Apply the concepts discussed in Topic Ten by writing a PL/I procedure called 'new_subsystem.pl1' which, when invoked, will do the following:

1. Determine whether or not a subdirectory called "F15C" exists in the callers default working directory. If it does, proceed to task 3 below. If it does not, proceed to task 2 below. If a segment or link called "F15C" exists in the caller's default working directory, delete/unlink it, notify the caller of your action, and proceed to step 2.

2. Since no "F15C" subdirectory exists in the caller's default working directory, create this directory. You should make sure that, besides the standard ACL entries, the directory also has an ACL entry giving "sma" access to Student_01.*.*. Report the creation of this directory to the caller.

3. Change the caller's working directory to the "F15C" directory, and notify the user of this action.

Compile and test out your procedure.

(CONTINUED ON NEXT PAGE)

WORKSHOP SEVEN

You may wish to use the following declarations which are in the segment, >udd>MEDclass>F15C>s1>include>w7.incl.pl1.

```
/* Begin w7.incl.pl1 */

dcl delete_$path entry (char (*), char (*), bit (6), char (*),
                        fixed bin (35));
dcl hcs_$add_dir_acl_entries entry (char (*), char (*), ptr,
                        fixed bin, fixed bin (35));
dcl hcs_$append_branchx entry (char (*), char (*), fixed bin (5),
          (3) fixed bin (3), char (*), fixed bin (1), fixed bin (1),
          fixed bin (24), fixed bin (35));
dcl hcs_$status_minf entry (char (*), char (*), fixed bin (1),
          fixed bin (2), fixed bin (24), fixed bin (35));
dcl get_group_id_$tag_star entry returns (char (32));
dcl get_default_wdir_      entry returns (char (168) aligned);
dcl change_wdir_           entry (char (168), fixed bin (35));
dcl absolute_pathname_     entry (char (*), char (*), fixed bin (35));
dcl (ioa_,
     com_err_)             entry options (variable);
dcl error_table_$nomatch   fixed bin (35) external;
dcl error_table_$noentry   fixed bin (35) external;
dcl addr                   builtin;
dcl rings (3)              fixed bin (3) internal static init (4, 4, 4)
                           options (constant);

dcl 1 dir_acl aligned,
       2 access_name       char (32) init ("Student_01.*.*"),
       2 dir_modes         bit (36) init ("111"b),
       2 status_code       fixed bin (35);

/* End w7.incl.pl1 */
```

WORKSHOP EIGHT

User Address and Name Space

1. Write a PL/I procedure called "my_tmsr.pl1" that will prompt the user for a reference name to be terminated. Using the appropriate entry point in term_, duplicate the action of the terminate single refname command (i.e. terminate the reference name, but do not make the segment unknown unless it was the last refname in the RNT for that segment). The program should end by notifying the user that the termination is complete. INCLUDE IN THE MESSAGE, THE ABSOLUTE PATHNAME OF THE SEGMENT ASSOCIATED WITH THAT REFNAME.

2. Execute a simple command (ex. who, memo, pwd, list). Test your program using that reference name as input.

3. Look at the contents of >udd>MEDclass>F15C>s1>call_sub1.pl1 and >udd>MEDclass>F15C>s1>sub1.pl1. At command level, initiate the object segment for the first program with the reference name "cs1" ("initiate >udd>MEDclass>F15C>s1>call_sub1 cs1"). Now execute the program by simply typing "cs1". This, of course, works no matter what your working directory is at the time of initiation or execution.

4. Use your "my_tmsr" procedure to terminate the reference name "sub1". Again execute the call_sub1 program using the name "cs1". It should work exactly as it did before.

WORKSHOP NINE

Writing a Command/Active Function

1. Write a procedure called 'parent.pl1' which can function either as a command or as an active function. It is to return the entryname portion of the parent directory of a segment supplied as an argument. That is, issuing the command

   ```
   ! parent >udd>MEDclass>F15C>s1>foo
   ```

   would result in 's1' being output to the terminal. Used as an active function

   ```
   ! [parent >udd>MEDclass>F15C>s1>foo]
   ```

   it would return the string 's1'.

   Note of course, the argument needn't be an absolute pathname.

2. Try your command out on various segments.

3. Test it's ability to work as an active function by issuing the command:

   ```
   status <[parent ??]
   ```

   where ?? is a segment in your working directory.

4. Test your program both as a 'command' and as an 'active function' giving it the wrong number of arguments.

You may wish to use the following declarations which are in the segment, >udd>MEDclass>F15C>s1>include>w9.incl.pl1.

```
/* Begin w9.incl.pl1 */

dcl cu_$arg_ptr         entry (fixed bin, ptr, fixed bin,
                               fixed bin (35));
dcl cu_$af_return_arg   entry (fixed bin, ptr, fixed bin (21),
                               fixed bin (35));
dcl expand_pathname_    entry (char (*), char (*), char (*),
                               fixed bin (35));
dcl complain            entry variable options (variable);
dcl (ioa_,
     com_err_,
     active_fnc_err_)   entry options (variable);
dcl error_table_$wrong_no_of_args external fixed bin (35);
dcl nargs               fixed bin;
dcl (arg_ptr,
     rtn_string_ptr)    ptr;
dcl rtn_string          char (max_length) varying
                        based (rtn_string_ptr);
dcl arg                 char (arg_len) based (arg_ptr);
dcl max_length          fixed bin (21);
dcl arg_len             fixed bin;
dcl code                fixed bin (35);
dcl af                  bit (1) init ("0"b);
dcl ME                  char (6) static init ("parent")
                        options (constant);
dcl entryname           char (32);
dcl dir_name            char (256);

/* End w9.incl.pl1 */
```