SOURCE DEBUGGING AND PROCESS ENVIRONMENT

COURSE CODE: F21

STUDENT HANDBOOK

FSO ISSUE DATE: JULY 1981

HONEYWELL INFORMATION SYSTEMS
MARKETING EDUCATION

Copyright ©

Honeywell Information Systems, Inc.

The information contained herein is the exclusive property of Honeywell Information Systems, Inc., except as otherwise indicated, and shall not be disclosed or reproduced, in whole or in part, without explicit written authorization from the company. The distribution of this material outside the company may occur only as authorized.

Printed in the United States of America
All rights reserved

CONTENTS

Page

Topic I — Debugging Programs on Multics 1-1
- Types of Programming Errors 1-1
- List of Debugging Tools 1-5
- Other Programming and Debugging Tools 1-6
- Source-Level Debugging. 1-8
- Object Level Debugging. 1-10

Topic II — SOURCE LEVEL DEBUGGING - AN INTRODUCTION TO THE probe COMMAND. 2-1
- The probe Environment 2-1
- A Sample Program. 2-4
- SCENARIO ONE: FUNDAMENTAL probe 2-11
- Scenario Two: More probe. 2-17
- Breaking Program Execution. 2-25
- Scenario three: Simple Break Processing . . . 2-29
- Additional Break Control. 2-36
- Probe Odds and Ends 2-54

Topic III — Other Source-Level Debugging Commands 3-1
- The trace Command 3-1
- Interaction of the Control Arguments. 3-5
- Tracing Example One 3-6
- Other trace Control Requests. 3-11
- Trace Example Two. 3-14
- The display pl1io error Command 3-20
- A display pl1io error Example 3-21

Topic IV — Advanced probe Requests 4-1
- Introduction. 4-1
- Scenario I - More probe Control 4-2
- Control of Output Processing. 4-7
- Scenario III - Program Manipualtion 4-9
- Scenario IV - probe Variables 4-14

Topic V — MULTICS USER RING RUNTIME STRUCTURES. 5-1
- Introduction. 5-1
- Supervisor Segments 5-3
- The Stack Segment - stack_n 5-6
- The area.linker Segment 5-11
  - Getting Space for Program Variables . 5-23

Topic VI — MULTICS DYNAMIC LINKING 6-1
- Introduction. 6-1

CONTENTS (con't)

		Page
	Multics Compiler Conventions.	6-8
	Mutlics Operating System Support.	6-11
	The Linker - Phase I.	6-13
	The Linker - Phase II	6-19
	By-Products of Dynamic Linking.	6-36
Topic VII	The Multics Programming Environment	7-1
	Destruction of the Programming Environment. .	7-1
	Error Recovery Techniques	7-8
Topic VIII	Other Useful Debugging Tools.	8-1
	list_external_variables	8-1
	list_external_variables	8-1
	reset_external_variables.	8-2
	reset_external_variables.	8-2
	delete_external_variables	8-3
	delete_external_variables	8-3
	print_bind_map.	8-4
	print_bind_map.	8-4
	print_link_info	8-5
	print_link_info, pli.	8-5
	resolve_linkage_error	8-7
	reslve_linkage_error, rle	8-7
	trace_stack	8-8
	trace_stack, ts	8-8
Appendix A	Debugging Tools	A-1
	area_status	A-1
	area_status	A-1
	cancel_cobol_program.	A-2
	cancel_cobol_program, ccp	A-2
	create_area	A-4
	create_area	A-4
	create_data_segment	A-5
	create_data_segment, cds.	A-5
	cumulative_page_trace	A-6
	cumulative_page_trace,cpt	A-6
	cv_ptr_	A-9
	cv_ptr_	A-9
	delete_external_variables	A-12
	delete_external_variables	A-12
	display_cobol_run_unit.	A-13
	display_cobol_run_unit, dcr	A-13
	display_pllio_err	A-14
	display_pllio_err, dpe.	A-14
	dump_segment.	A-15
	dump_segment, ds.	A-15
	io_call	A-18
	io_call, io	A-18
	list_external_variables	A-31

CONTENTS (con't)

Page

list_external_variables A-31
list_temp_segments. A-32
list_temp_segments. A-32
page_trace. A-34
page_trace, pgt A-34
print_bind_map. A-36
print_bind_map. A-36
print_link_info A-37
print_link_info, pli. A-37
print_linkage_usage A-39
print_linkage_usage, plu. A-39
probe A-40
probe, pb A-40
profile A-76
profile A-76
reset_external_variables. A-79
reset_external_variables. A-79
resolve_linkage_error A-80
reslve_linkage_error, rle A-80
run_cobol A-81
run_cobol, rc A-81
set_fortran_common. A-84
set_fortran_common, sfc A-84
set_system_storage. A-86
set_system_storage. A-86
set_user_storage. A-88
set_user_storage. A-88
stop_cobol_run. A-90
stop_cobol_run, scr A-90
trace A-91
trace A-91
trace_stack A-102
trace_stack, ts A-102

Appendix W Workshops W-1
Workshop One. W-1
Workshop Two. W-10
Workshop Three. W-13

TOPIC I

Debugging Programs on Multics 1-1
    Types of Programming Errors 1-1
    List of Debugging Tools 1-5
    Other Programming and Debugging Tools 1-6
    Source-Level Debugging. 1-8
    Object Level Debugging. 1-10

OBJECTIVES:

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

1. Describe the methods used to find and correct syntactical errors.

2. Describe the methods used to find and correct semantic errors.

3. List some of the more common source-level debugging tools.

4. Outline, in general terms, the concepts of source-level debugging as opposed to object-level debugging.

TYPES OF PROGRAMMING ERRORS

- TWO KINDS OF ERRORS (BUGS) COMMONLY OCCUR IN PROGRAMS
  - SYNTAX ERRORS
    - TYPING ERRORS
    - MISUSE OF A LANGUAGE STATEMENT
  - SEMANTIC ERRORS
    - SPECIFYING THE WRONG DATA
    - USING DATA INCORRECTLY
      - ATTEMPTING TO REFERENCE MORE DATA THAN IS PRESENT
      - PERFORMING INVALID OPERATIONS ON THE DATA
      - PERFORMING AN INCORRECT SEQUENCE OF OPERATIONS ON THE DATA

TYPES OF PROGRAMMING ERRORS

- SYNTAX ERRORS

  - CAN BE DETECTED BY

    - PROOFREADING THE PROGRAM

    - COMPILING THE PROGRAM AND OBSERVING THE ERROR DIAGNOSTIC MESSAGES

  - CAN BE CORRECTED BY

    - EDITING THE SOURCE PROGRAM TO CORRECT THE ERRORS (DIAGNOSED BY THE COMPILER OR FOUND DURING PROOFREADING)

    - RECOMPILING THE SOURCE

    - REPEATING THIS PROCESS UNTIL NO MORE ERRORS ARE DIAGNOSED

TYPES OF PROGRAMMING ERRORS

- SEMANTIC ERRORS
  - CAN BE DETECTED BY
    - PROOFREADING THE PROGRAM
    - INSERTING TEMPORARY STATEMENTS IN THE PROGRAM SOURCE TO PRINT INFORMATION ABOUT DATA VALUES, FLOW OF CONTROL, ETC.
      - ARE INTERMEDIATE DATA VALUES CORRECT?
      - DOES THE POINT OF EXECUTION FLOW THROUGH THE PROGRAM IN THE EXPECTED WAY?
    - RUNNING THE PROGRAM AND OBSERVING
      - WHETHER OR NOT THE PROGRAM RUNS TO COMPLETION
        - DOES THE PROGRAM GO INTO A LOOP?
        - DOES AN UNEXPECTED ERROR CONDITION OCCUR WHICH HALTS PROGRAM EXECUTION?
      - WHETHER OR NOT THE PROGRAM PRODUCES THE EXPECTED RESULTS
        - DOES THE PROGRAM PRODUCE CORRECT OUTPUT DATA?
        - DOES THE PROGRAM DIAGNOSE INCORRECT INPUT DATA?
  - CAN BE CORRECTED BY
    - IDENTIFYING THE POINT OF ERROR
    - EDITING THE SOURCE PROGRAM TO CORRECT THE ERRORS
    - RECOMPILING THE SOURCE

- REPEATING THIS PROCESS UNTIL THE PROGRAM OPERATES CORRECTLY

LIST OF DEBUGGING TOOLS

- MAJOR TOOLS FOR DIAGNOSING

  - SYNTAX ERRORS

    - THE COMPILERS

      - pl1
      - cobol
      - fortran

  - SEMANTIC ERRORS

    - SOURCE-LEVEL DEBUGGING TOOLS

      - probe
      - trace recursive calls
      - display_pl1 io_error dpe

    - OBJECT-LEVEL DEBUGGING TOOLS

      - debug
      - trace_stack
      - dump_segment
      - print_link_info
      - print_bind_map
      - display_component_name
      - print_linkage_usage

OTHER PROGRAMMING AND DEBUGGING TOOLS

- FILE MANIPULATION TOOLS
  - io_call
  - print_attach_table
  - close_file
  - vfile_status
  - vfile_adjust
  - adjust_bit_count
  - set_bit_count

- EXTERNAL REFERENCE MANIPULATION TOOLS
  - resolve_linkage_error
  - list_external_variables
  - delete_external_variables
  - reset_external_variables
  - set_fortran_common
  - create_data_segment
  - error_table_compiler

- COBOL RUN UNIT TOOLS
  - run_cobol
  - display_cobol_run_unit
  - stop_cobol_run
  - cancel_cobol_program

OTHER PROGRAMMING AND DEBUGGING TOOLS

- GENERAL RUN UNIT COMMANDS
  - run
  - stop_run

- SEARCH RULE AND DYNAMIC LINKING TOOLS
  - print_search_rules
  - add_search_rules
  - delete_search_rules
  - where (handwritten: hunt)
  - list_ref_names
  - initiate
  - terminate
  - terminate_refname
  - terminate_single_refname
  - terminate_segno (handwritten: lose knowledge of location)

- AREA MANIPULATION TOOLS
  - area_status
  - create_area
  - set_user_storage
  - set_system_storage
  - list_temp_segments

SOURCE-LEVEL DEBUGGING

- SOURCE-LEVEL DEBUGGING ALLOWS THE PROGRAMMER TO
  - DISPLAY PROGRAM SOURCE STATEMENTS
    - GIVEN A STATEMENT LABEL
    - GIVEN A LINE NUMBER
  - DISPLAY THE VALUE OF PROGRAM DATA VARIABLES
    - GIVEN THE NAME OF THE VARIABLE
  - DISPLAY THE VALUE OF PROGRAM DATA VARIABLES
    - GIVEN THE STORAGE LOCATION AND DATA FORMAT OF THE VARIABLE
  - DISPLAY THE DECLARATION OF A PROGRAM VARIABLE
  - DISPLAY THE LIST OF (USER RING) ACTIVE PROGRAMS

SOURCE-LEVEL DEBUGGING

- SOURCE-LEVEL DEBUGGING ALLOWS THE PROGRAMMER TO
  - SET BREAKPOINTS BEFORE OR AFTER STATEMENTS
    - TO INTERRUPT NORMAL FLOW OF EXECUTION
    - TO INTERROGATE THE STATE OF THE EXECUTING PROGRAM
    - TO CHANGE THE VALUE OF PROGRAM DATA
    - TO ALTER THE FLOW OF EXECUTION THROUGH THE PROGRAM
    - TO CONDITIONALLY PERFORM ANY OF THESE FUNCTIONS
      - ONLY IF A PROGRAM DATA VALUE MEETS SOME CONDITION
      - REPEATEDLY WHILE A PROGRAM DATA VALUE MEETS SOME CONDITION
  - TRACE CALLS TO A PARTICULAR PROGRAM
  - CALL PROGRAMS WHICH EXPECT NON-CHARACTER ARGUMENTS

OBJECT LEVEL DEBUGGING

- OBJECT-LEVEL DEBUGGING TOOLS ALLOW THE PROGRAMMER TO
  - PERFORM MOST SOURCE-LEVEL DEBUGGING FUNCTIONS, PLUS
  - CHANGE THE VALUE OF PROGRAM DATA VARIABLES
    - GIVEN THE STORAGE LOCATION AND DATA FORMAT OF THE VARIABLE
  - DISPLAY PROGRAM SOURCE STATEMENTS
    - GIVEN A LOCATION IN THE PROGRAM OBJECT SEGMENT
  - DISPLAY AND CHANGE THE VALUE OF MACHINE INSTRUCTIONS COMPILED TO EXECUTE A SOURCE STATEMENT
  - DISPLAY THE FORMATTED CONTENTS OF
    - THE PROGRAM ACTIVATION HISTORY SEGMENT (THE STACK)
    - AREA SEGMENTS

OBJECT LEVEL DEBUGGING

- OBJECT-LEVEL DEBUGGING TOOLS ALLOW THE PROGRAMMER TO

  - DISPLAY AND CHANGE THE CONTENTS OF ANY SEGMENT

    - GIVEN ITS PATHNAME

    - GIVEN ITS REFERENCE NAME

    - GIVEN ITS SEGMENT NUMBER

    - WHEN THE USER HAS ADEQUATE ACCESS TO PERFORM THE REQUESTED OPERATION

  - DISPLAY AND CHANGE THE CONTENTS OF HARDWARE REGISTER IMAGES

TOPIC II

SOURCE LEVEL DEBUGGING - AN INTRODUCTION TO THE probe COMMAND. 2-1
The probe Environment 2-1
A Sample Program. 2-4
SCENARIO ONE: FUNDAMENTAL probe 2-11
Scenario Two: More probe. 2-17
Breaking Program Execution. 2-25
Scenario three: Simple Break Processing 2-29
Additional Break Control. 2-36
Probe Odds and Ends 2-54

OBJECTIVES:

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

1. Use the appropriate PL/1 compiler control arguments to enable probe to function on an object segment.

2. Describe the different situations under which probe may be invoked.

3. Debug a program using the following probe requests:

    source (sc)

    value (v)

    symbol (sb)

    quit (q)

    help

    stack (sk)

4. Manipulate breakpoints in a program using the following probe requests:

    position (ps), status (st)

    before (b), after (a), reset (r)

    continue (c), continue_to (ct), step (s)

5. Use the probe builtin functions.

THE PROBE ENVIRONMENT

■ ENVIRONMENT OVERVIEW

- FUNCTIONS AS A SUBSYSTEM FOR PROGRAM RECOVERY

- DRIVEN BY INTERACTIVE REQUESTS

  - LONG AND SHORT FORMS AVAILABLE

  - REQUEST DELIMITER IS EITHER NEW LINE OR SEMI-COLON

- WORKS BEST WITH COMPILER GENERATED SYMBOL TABLE

  - CURRENTLY AVAILABLE FOR COBOL, FORTRAN, AND PL/I

  - USE -table OPTION

  - MAY ALSO USE -short_table OPTION

```
r 07:47 0.159 43

pl1 check_back_issues -sv2
PL/I 26a
r 07:48 3.391 221

probe check_back_issues
probe: Cannot get statement map for this procedure.
r 07:48 0.108 31

pl1 check_back_issues -sv2 -tb
PL/I 26a
r 07:48 3.486 217

probe check_back_issues
Using check_back_issues (no active frame).
source
check_back_issues:
          proc;
quit
r 07:49 0.173 21
```

THE PROBE ENVIRONMENT

- MAY BE INVOKED FROM SEVERAL SITUATIONS
  - AFTER AN UNHANDLED CONDITION (READY LEVEL NOT EQUAL TO ONE)
  - AT READY LEVEL ONE WITH NO PROGRAM SPECIFIED to examine stack
  - AT READY LEVEL ONE WITH PROGRAM SPECIFIED on conversion snap
  - IMPLICITLY AT A PREVIOUSLY SET BREAKPOINT

- MANAGES TWO IMPORTANT PIECES OF INFORMATION
  - SOURCE POINTER
    - FRAME OF PROGRAM (ONLY IF ACTIVE)
    - BLOCK OF CODE WITHIN PROGRAM
    - LINE OF CODE WITHIN PROGRAM
    - BASED UPON MANNER OF INVOCATION
  - CONTROL POINTER
    - LAST INSTRUCTION EXECUTED
    - BASED UPON MANNER OF INVOCATION
    - USUALLY AT BREAKPOINT, FAULTING INSTRUCTION, OR FIRST INSTRUCTION IN BLOCK

- USES A PERMANENT DATA BASE FOR OPERATION

THE PROBE ENVIRONMENT

- LOCATED AT >udd>[user project]>[user name]>[user name].probe

- CONTAINS PATH NAMES OF PROGRAMS WITH BREAKPOINTS SET IN THEM

- IS REFERENCED BY probe WHENEVER

  - A BREAKPOINT IS ESTABLISHED OR FREED

  - A BREAKPOINT IS ENCOUNTERED WHILE A PROGRAM IS RUNNING

- IF THIS DATA BASE IS DELETED, probe LOOSES INFORMATION ABOUT BREAKPOINTS

  - IF probe COMPLAINS ABOUT A "seg_fault" THE DATA BASE MAY BE DELETED

  - probe CANNOT FREE ANY BREAKPOINTS THAT HAVE BEEN PREVIOUSLY SET

  - TO FREE ANY "LOST" BREAKPOINTS, ONE MUST RECOMPILE THE AFFECTED PROGRAM

A SAMPLE PROGRAM

- THE EXAMPLE FOR THIS COURSE

  - IS WRITTEN IN PL/I

  - IS FAIRLY WELL STRUCTURED

  - IS EASY TO READ IF YOU ALREADY KNOW A FORTRAN OR COBOL RELATED LANGUAGE

  - HAS SOME BUGS IN IT

  - WILL BE USED IN THE DEBUGGING SCENARIOS THAT FOLLOW

A SAMPLE PROGRAM

- THE FOLLOWING PROGRAM IS SUPPOSED TO
  - KEEP TRACK OF BACK ISSUES OF MAGAZINES OF A SMALL COMPANY
    - EACH RECORD CONTAINS THE NUMBER OF ISSUES LEFT IN STOCK, HOW MANY ARE REQUESTED FOR SHIPPING, AND THE CURRENT COST OF PURCHASE
  - PRINT OUT A SUMMARY OF THIS DATA
  - ACCEPT TWO INPUT STRINGS
    - BOTH IN THE FORM OF volume:number
    - SPECIFY THE FIRST AND LAST ISSUES TO BE SUMMARIZED
  - PRINT OUT EACH RECORD AND CALCULATE RUNNING TOTALS

THE PROGRAM

```
check_back_issues:
          proc;

     /****************************************
      *   declarations for check_back_issues   *
      *   and its subroutines                  *
      ****************************************/

dcl back_issues file;
dcl (first_issue, last_issue) char (12);
dcl (first_issue_delim, last_issue_delim) fixed bin (24);
dcl index builtin;
dcl substr builtin;
dcl number_of_issues fixed bin;
dcl issue fixed bin;
dcl 1 issue_record,
          2 current_inventory fixed bin (17),
          2 pending_requests fixed bin (17),
          2 cost_of_issue fixed dec (8,2);
dcl total_number_pending fixed bin;
dcl total_number_stocked fixed bin;
dcl total_stock_value fixed dec (8,2);
dcl (current_volume, current_number) fixed bin;
dcl (last_issue_num, first_issue_num) fixed bin;
dcl (sysin, sysprint) file;
dcl (first_issue_volume, last_issue_volume) fixed bin;
dcl colon internal static options (constant)
          char (1) aligned init (":");

          open file (back_issues) keyed sequential input;

     /********************************************
      *   get number of the first and last issues   *
      *   the user wants to check.  the form is     *
      *   volume:number.  this routine will split   *
      *   the components up into issue_volume       *
      *   and issue_number, and position to that    *
      *   record in the file.                       *
      ********************************************/

          put list ("from (specify vol:num): ");
          get list (first_issue);
          put list ("to (specify vol:num): ");
          get list (last_issue);
```

```
        first_issue_delim = index (first_issue, colon);
        last_issue_delim = index (last_issue, colon);
        first_issue_volume =
             substr (first_issue, 1, first_issue_delim);
        last_issue_volume =
             substr (last_issue, 1, last_issue_delim);
        first_issue_num =
             substr (first_issue, first_issue_delim);
        last_issue_num =
             substr (last_issue, last_issue_delim);


        call position_file (first_issue_volume,
             first_issue_num);


        number_of_issues =
             (6*last_issue_volume + last_issue_num) -
             (6*first_issue_volume + first_issue_num);

        do issue = 1 to number_of_issues;
             call print_record ();
        end;

        call print_summary ();

        close file (back_issues);

        return;

    /*********************************
     *   begin support subroutines   *
     *********************************/

print_record:
        proc ();

    /**********************************************
     *   this subroutine obtains the next record  *
     *   from the back_issues file, calculates    *
     *   some totals, and outputs the current     *
     *   record in a formatted form.              *
     **********************************************/

        call get_record ();

        total_number_pending =
             total_number_pending +
             issue_record.pending_requests;
        total_number_stocked =
             total_number_stocked +
             issue_record.current_inventory;
        total_stock_value =
```

```
                total_stock_value +
                (issue_record.current_inventory*
                issue_record.cost_of_issue);

            put skip edit ("volume",
                current_volume,
                "number",
                current_number,
                "stocked:",
                issue_record.current_inventory,
                "outstanding requests:",
                issue_record.pending_requests,
                "cost of this issue:",
                issue_record.cost_of_issue,
                ".")
                (r (output_format_1))
                file (sysprint);

            return;

output_format_1:
            format (a(6), x(1), f(3,0), x(1),
                a(6), x(1), f(3,0), skip (1), x(11),
                a(8), x(1), f(6,0), x(1),
                a(21), x(1), f(6,0), x(1),
                a(19), x(1), p"$$$,$$9v.99", a(1));

        end print_record;


position_file:
        proc (first_vol, first_num);

       /**************************************************
        *   this subroutine positions the back_issues    *
        *   file to the record specified by the          *
        *   first issue specifier given by the user      *
        *   at the beginning of the program. to          *
        *   position to the record, we simply read       *
        *   records we don't want and do nothing with    *
        *   them.                                        *
        **************************************************/

dcl     (first_vol, first_num) fixed bin;

        do while (first_vol > current_volume);
            call get_record ();
        end;

        do while (first_num > current_number);
            call get_record ();
        end;
```

```
     return;

end position_file;

get_record: proc ();

dcl key char (8);

    /**********************************************
     *   this subroutine reads a record from the   *
     *   back_issue file into the issue record.    *
     *   the other necessary information, vol      *
     *   and num of the issue, was stored in       *
     *   the record's key.  we must extract this   *
     *   from our internally declared key and      *
     *   place it in the globally available        *
     *   current_volume and current_number vars    *
     **********************************************/

         read file (back_issues)
              into (issue_record)
              keyto (key);

         current_volume = substr (key, 1, 4);
         current_number = substr (key, 5, 4);

         return;

    end get_record;


print_summary:
          proc ();

    /*************************************
     *   a simple subroutine, all this    *
     *   does is print out the  totals    *
     *   calculated by the print_record   *
     *   subroutine.                      *
     *************************************/

          put skip (2)
               edit ("number of issues stocked:",
               total_number_stocked,
               "number of requests pending:",
               total_number_pending,
               "total stock value:",
               total_stock_value,
               ".")
               (r (output_format_2))
               file (sysprint);

          return;
```

```
output_format_2:
          format (a(25), x(1), f(6,0), skip (1),
               a(27), x(1), f(6,0), skip (1),
               a(18), x(1), p"$$$,$$9v.99", a(1));

end print_summary;

end check_back_issues;
```

- SOME PRIMARY probe REQUESTS

  - THE source REQUEST

    - PRINTS SOURCE STATEMENTS

    - USAGE:

      source

      sc

      source <number of lines>

      sc <number of lines>

    - EXAMPLES:

      source 7

      sc 3

  - THE value REQUEST

    - DISPLAYS THE VALUE OF A SINGLE VARIABLE, AN EXPRESSION, OR SECTION OF AN ARRAY

    - USAGE:

      value <expression>

      v <expression>

      value <array cross section>

      v <array cross section>

SCENARIO ONE: FUNDAMENTAL PROBE

- EXAMPLES:

  value x

  value array (1:5)

  value str1.mem2.elem

  v ptr1 -> some_based_var

  v ptr2 -> really_big.mem1.comp (2).z

- SPECIFYING THE EXPRESSION FOR THE value REQUEST

  - MADE FROM PROGRAM VARIABLES, CONSTANT VALUES AND probe DEFINED FUNCTIONS

    - PROGRAM VARIABLES MUST APPEAR EXACTLY AS THEY WERE TYPED IN YOUR PROGRAM

      loop_counter (PL/I)

      VECT(1) (FORTRAN)

      data-part (COBOL)

    - CONSTANTS SHOULD BE IN A FORM ACCEPTABLE TO YOUR PROGRAM

      -99

      3.2e5

      "abcde"

      'STRING'

    - probe MAINTAINS A SET OF BUILTIN FUNCTIONS THAT RETURN VALUES TO YOU

      addr - RETURNS THE ADDRESS OF ITS ARGUMENT

      addrel - RETURNS AN ADDRESS RELATIVE TO THE SPECIFICATION OF ITS ARGUMENTS

      baseptr - RETURNS THE ADDRESS OF THE BEGINNING OF A SEGMENT

      length - RETURNS THE LENGTH OF A BIT OR CHARACTER STRING

maxlength - RETURNS THE MAXIMUM ALLOWED LENGTH OF A STRING

null - RETURNS A SPECIAL INVALID ADDRESS

octal - RETURNS THE MACHINE REPRESENTATION OF ITS ARGUMENT

pointer - RETURNS AN ADDRESS BASED UPON ITS ARGUMENTS

rel - RETURNS THE ADDRESS WITHIN A SEGMENT INTO WHICH ITS ARGUMENT POINTS

segno - RETURNS THE NUMBER OF THE SEGMENT INTO WHICH ITS ARGUMENT POINTS

substr - RETURNS A PORTION OF A CHARACTER OR BIT STRING

unspec - RETURNS THE BINARY REPRESENTATION OF ITS ARGUMENT

- AN EXPRESSION CAN CONTAIN OPERATORS
  - FOUR DEFINED WITHIN PROBE

    ADDITION - USE A PLUS SIGN (+)

    SUBTRACTION - USE A MINUS SIGN (-)

    MULTIPLICATION - USE AN ASTERISK (*)

    DIVISION - USE A SLASH (/)
  - ORDER IS MULTIPLICATION AND DIVISION, THEN ADDITION AND SUBTRACTION
  - ORDER MAY BE OVERRIDDEN WITH PARENTHESES

- THE symbol REQUEST
  - SHOWS YOU THE DATA TYPE OF A PROGRAM VARIABLE
  - USAGE:

    symbol <name of variable>

    sb <name of variable>

- EXAMPLES:

  symbol x

  sb HYPTN

- THE quit REQUEST

  - CAUSES YOU TO LEAVE PROBE

  - USED TO GET BACK TO COMMAND LEVEL

  - USAGE:

    quit

    q

- THE SCENARIO

  - THE PROGRAM BLOWS UP

  - PROBE IS USED TO ASSESS THE DAMAGE

```
r 12:54 0.155 21

check_back_issues

from (specify vol:num):1:1
          to (specify vol:num):2:1

Error:  conversion condition by >udd>MEDmult>F21>doodle>bad_
\ccbi$check_back_issues|540 (line 48)
onsource = "1:", onchar = ":"
Invalid character follows a numeric field.
system handler for error returns to command level
r 12:54 0.273 52 level 2

probe
Condition conversion raised at line 48 of check_back_issues
\c(level 7).

source
first_issue_volume =
               substr (first_issue, 1, first_issue_delim);

value first_issue_delim
first_issue_delim =2

value first_issue
first_issue ="1:1         "

value substr (first_issue, 1, first_issue_delim);
"1:"

symbol first_issue_volume
 fixed bin (17) automatic
Declared in check_back_issues

v substr (first_issue, 1, first_issue_delim - 1);
"1"

q
r 07:03 0.591 210 level 2

rl
r 07:03 0.045 9
```

SCENARIO ONE: FUNDAMENTAL PROBE

- TECHNIQUE
  - INVOKE probe - IT TELLS YOU WHAT HAPPENED
  - FIND THE STATEMENT AT WHICH THE PROGRAM DIED
  - CHECK ALL THE VARIABLES IN THAT STATEMENT
    - NOTE THE IMPLICIT CONVERSION
    - THE SUBSTR BUILTIN WAS THE CULPRIT

- YOUR TURN
  - WHAT WAS THE PROGRAMMER TRYING TO DO WITH THE SUBSTRING BUILTIN?
  - HOW WOULD YOU CHANGE THE PROGRAM SO THAT THE IMPLICIT CONVERSION SUCCEEDS?
    first issue delim = 2
  - ARE THERE ANY OTHER STATEMENTS WHERE THIS CONDITION MAY HAPPEN?

SCENARIO TWO: MORE PROBE

- AS ERRORS ARE FOUND THEY ARE FIXED
  - MODIFY THE SOURCE ONLY
  - RECOMPILE INTO NEW OBJECT
  - CLEAN UP ANY FILES THAT MAY HAVE BEEN LEFT INCONSISTENT

```
qedx
rcheck_back_issues.pl1
48
            first_issue_volume =
 .+1
                substr (first_issue, 1, first_issue_delim);
s/)/ - 1)/p
                substr (first_issue, 1, first_issue_delim - 1);
w
q
r 07:55 0.549 88

pl1 check_back_issues -tb -sv2
PL/I 26a
r 07:55 3.317 83

r 07:55 0.049 8

close_file back_issues
r 07:58 0.082 23
```

- RUN THE NEW PROGRAM
  - BE AWARE OF NEW PROBLEMS

SCENARIO TWO: MORE PROBE

- MAKE SURE THE OLD PROBLEM IS FIXED

- LOOK FOR ANY EFFECTS YOUR CHANGE MAY HAVE ON OTHER PORTIONS OF THE PROGRAM

```
check_back_issues

from (specify vol:num):1:1
          to (specify vol:num):1:4

Error:  conversion condition by >user_dir_dir>MEDmult>F21>do
\codle>bad_cbi$check_back_issues|551 (line 50)
onsource = "1:", onchar = ":"
Invalid character follows a numeric field.
system handler for error returns to command level
r 07:58 0.367 34 level 2
probe
Condition conversion raised at line 50 of check_back_issues
\c (level 7).
source
          last_issue_volume =
               substr (last_issue, 1, last_issue_delim);
From Pandolf.MEDmult 05/19/81  0759.6 mst Tue:
The handbooks came in today. I have them if you want to see
\cthem.
sm Pandolf.MEDmult good, be there later
probe: Unknown request. "sm"
```

SOME INVALUABLE REQUESTS

- THE list_requests REQUEST

  - LISTS ALL THE ALLOWED REQUESTS IN probe

  - USAGE:

```
list_requests

lr
```

```
list_requests

Summary of probe requests:

after, a              Set a breakpoint after the specified statement.
args                  Print argument list for procedure.
before, b             Set a breakpoint before the specified statement.
call, cl              Call a subroutine.
continue, c           Continue after a breakpoint.
continue_to, ct       Resume execution from last breakpoint and stop
                        at specified statement.
declare, dcl          Create a probe variable.
display, ds           Display storage in various formats.
execute, e            Execute a Multics command line, usually
                        within a break request.
goto, g               Continue execution at a specified statement.
halt, h               Halt ands re-enter probe.
help                  Print info files for probe requests.
if                    Execute probe requests based on specified
                        condition.
input_switch, isw     Set the I/O switch used for probe input.
language, lng         Display or set the current language.
let, l                Change the value of a variable.
list_help, lh         List the available info topics for probe.
list_builtins, lb     Print a  summary listing of all probe builtins.
list_requests, lr     Print a summary listing of the probe requests.
list_variables, lsv   Print type and value of one or more probe
                        variables.
modes, mode           Set probe operation modes.
output_switch, osw    Set the I/O switch used for probe output.
pause, p              Reset the current breakpoint and halt.
position, ps          Move the probe pointer to a new location
                        and display the source.
quit, q               Leave probe and return to Multics command
                        level.
reset, r              Reset breakpoints.
source, sc            Display source of program.
stack, sk             Display the stack.
status, st            Display the status of breakpoints.
step, s               Execute one statement and halt.
symbol, sb            Display information about the specified
                        symbol.
use                   Move the probe pointer to a new location.
value, v              Print the value of a variable or expression.
where, wh             Display the current values of the probe
                        pointers.
while, wl             Execute probe requests while condition
                        is true.

Type "help" for more information.
```

SCENARIO TWO: MORE PROBE

- THE help REQUEST

  - USAGE:

    help

    help <request>

    help <feature>

  - EXAMPLES:

    help

    help quit

    help EXPRESSIONS

- THE execute REQUEST

  - ALLOWS A MULTICS COMMAND TO BE PROCESSED WHILE STILL IN probe

  - USAGE:

    execute <command line>

    e <command line>

  - EXAMPLES:

    execute "pwd"

    e "list *.pl1"

I USING THESE REQUESTS

```
help execute
09/27/79 The "execute" request.

Syntax: execute STRING

The contents of STRING are passed to the Multics command pro
\ccessor.
This request is chiefly useful in break request list, because
\ce the more
convenient escape to the Multics command processor is not av
\cailable
then.

The user can pass an arbitrary line to the Multics command p
\crocessor
by preceeding it with ".." on a new line.

Examples (6 lines).  More help?no
execute sm Pandolf.MEDmult good, be there later
probe (execute): The Multics command lines must be enclosed
\cin quotes.
e "sm Pandolf.MEDmult good, be there later"
```

I FINISHING UP THE CURRENT ERROR

```
source
          last_issue_volume =
               substr (last_issue, 1, last_issue_delim);
value substr (last_issue, 1, last_issue_delim)
"1:"
sb last_issue_volume
 fixed bin (17) automatic
Declared in check_back_issues
quit
r 08:10 2.508 492 level 2

qedx
rcheck_back_issues.pl1
50,51p
          last_issue_volume =
 .+1
               substr (last_issue, 1, last_issue_delim);
s/)/ - 1)/p
               substr (last_issue, 1, last_issue_delim - 1);
52,55p
          first_issue_num =
               substr (first_issue, first_issue_delim);
          last_issue_num =
               substr (last_issue, last_issue_delim);
53s/)/ + 1)/p
               substr (first_issue, first_issue_delim + 1);
55s/)/ + 1)/p
               substr (last_issue, last_issue_delim + 1);
w
q
r 08:12 0.423 80 level 2

pl1 check_back_issues -sv2 -tb
PL/I 26a
r 08:13 3.524 141 level 2

rl
r 08:15 0.046 29

close_file back_issues
r 08:15 0.051 21
```

- ADDITIONAL USE OF FUNDAMENTAL PROBE REQUESTS

  - SOME USEFUL INFORMATION ABOUT A FAILURE WILL COME INDIRECTLY FROM THE probe DIALOGUE

  - NOTE THE USE OF value IN THE NEXT WINDOW

```
check_back_issues

from (specify vol:num):1:1
          to (specify vol:num):1:4

Error:  illegal_procedure condition by >user_dir_dir>MEDmult
\c>F21>doodle>bad_cbi$print_record|675 (line 98)
(while in pl1 operator real_to_real_tr)
referencing stack_4|6503 (in process dir)

r 08:15 0.369 42 level 2

pb
Condition illegal_procedure raised at line 98 of print_recor
\cd (level 8).
sc
          total_stock_value =
               total_stock_value +
               (issue_record.current_inventory*
               issue_record.cost_of_issue);
v issue_record.cost_of_issue
 cost_of_issue = 5
v issue_record.current_inventory
 current_inventory = 23
v total_stock_value
total_stock_value = (invalid decimal data)
symbol total_stock_value
 fixed dec (8,2) automatic
Declared in check_back_issues
value octal (total_stock_value)
040040040040040040040040040
r 08:19 0.234 33 level 2

rl
r 08:19 0.037 2
```

SCENARIO TWO: MORE PROBE

```
qedx
rcheck_back_issues.pl1
/dcl total_stock_value/
dcl total_stock_value fixed dec (8,2);
s/;/ init (0);/p
dcl total_stock_value fixed dec (8,2) init (0);
w
q
r 08:20 0.207 24

pl1 check_back_issues -sv2 -tb
PL/I 26a
r 08:21 3.461 233

close_file back_issues
r 08:22 0.107 69
```

- THE illegal_procedure CONDITION ITSELF EXPLAINED LITTLE ABOUT THE ERROR

- THE PL/I STATEMENT WAS TOO COMPLEX TO QUICKLY DETERMINE THE IMMEDIATE CAUSE

- THE value REQUEST UNCOVERED THE FACT THAT ONE VARIABLE HAD INVALID DATA

  - BECAUSE OF THIS THE VALUE OF total_stock_value WAS UNPRINTABLE

  - THE octal BUILTIN FUNCTION ALLOWED US TO LOOK AT THE DATA THAT WAS THERE REGARDLESS OF THE DATA TYPE

  - THE PRESENCE OF SPACES (OCTAL 040) INDICATED THAT THE VARIABLE WAS NEVER INTIALIZED (TO OCTAL 060)

BREAKING PROGRAM EXECUTION

- BREAKPOINTS
  - AT TIMES, THE PROGRAMMER DESIRES TO VIEW INTERMEDIATE PROGRAM VALUES
    - ONE OPTION IS TO PLACE I/O STATEMENTS IN THE SOURCE PROGRAM
      - COSTS TOO MUCH IN TERMS OF RECOMPILES
      - STILL PROVIDES NO WAY TO SUSPEND THE PROGRAM
    - LET THE DEBUGGER DO IT
      - RECOMPILING NOT NECESSARY TO CHANGE DEBUGGING BEHAVIOR
      - THE PROGRAM MAY BE SUSPENDED OR CONTINUE ON AFTER PRINTING OUT SOME DIAGNOSTIC

- PROBE IMPLEMENTATION
  - REVOLVES AROUND THE "BREAKPOINT"
    - A LIST OF ONE OR MORE probe REQUESTS TO BE PERFORMED WHEN A STATEMENT IS REACHED
    - LIKE A SMALL PROGRAM FOR EACH STATEMENT
    - MODIFIABLE AT WILL BY THE PROGRAMMER
  - probe BREAKPOINTS CAN BE SPECIFIED TO BE EXECUTED EITHER BEFORE OR AFTER AN EXECUTABLE STATEMENT

BREAKING PROGRAM EXECUTION

- BEFORES AND AFTERS ARE SEPARATE BREAKS, AND BOTH WILL BE EXECUTED IF ENCOUNTERED CONSECUTIVELY

- COBOL COMPILER RESTRICTIONS LIMIT COBOL PROGRAMS TO THE USE OF BREAKPOINTS BEFORE THE STATEMENT ONLY

BREAKPOINT REQUESTS

- THE before REQUEST

  - USAGE:

    ```
    before <line>

    b <line>

    before <line>: (<request list>)

    b <line>: (<request list >)
    ```

  - EXAMPLES:

    ```
    before

    before 50

    before 5: (value x; value y)

    before: value comp-val
    ```

    (value x; value y; halt) will stop exec

  - NOTES:

    - IF NO LINE NUMBER IS SPECIFIED, THE CURRENT LINE IS ASSUMED

    - IF ONLY ONE REQUEST IS DESIRED AT THE BREAK, THEN THE PARENTHESES MAY BE OMITTED

    - IF NO REQUEST IS SPECIFIED, THEN A "halt" REQUEST IS ASSUMED, CAUSING THE PROGRAM TO STOP AND PROBE TO BE ENTERED

- THE stack REQUEST
  - NOT SPECIFICALLY A BREAK REQUEST
  - DISPLAYS LIST OF ALL PROGRAMS THAT HAVE NOT FINISHED YET
  - ALLOWS YOU TO SEE HOW FAR YOUR PROGRAM HAS RUN
  - SHOWS RELATIONSHIPS BETWEEN PROGRAMS
  - MORE ON STACKS LATER
  - USAGE:

    ```
    stack
    sk
    stack <amount>
    sk <amount>
    stack <first frame, amount>
    sk <first frame, amount>
    ```

  - EXAMPLES:

    ```
    sk
    stack 4
    sk 12,3
    ```

- THE position REQUEST
  - POSITIONS TO AND PRINTS A SPECIFIED SOURCE STATEMENT

I USAGE:

```
position <label>
ps <+,-line>
ps <line>
ps <"string">
```

I EXAMPLES:

```
position do_label
ps +3
ps -4
ps 68
ps "i=5"
ps /i=5/
```

SCENARIO THREE: SIMPLE BREAK PROCESSING

- LET'S RETURN TO OUR EXAMPLE

```
check_back_issues

from (specify vol:num):1:1
          to (specify vol:num):1:4

volume   1 number   1
           stocked:     23 outstanding requests:      0 cost
\c of this issue:
  $5.00.
volume   1 number   2
           stocked:     30 outstanding requests:      2 cost
\c of this issue:
  $3.00.
volume   1 number   3
           stocked:     27 outstanding requests:      0 cost
\c of this issue:
  $3.00.
Error:  size condition by >user_dir_dir>MEDmult>F21>doodle>b
\cad_cbi$print_summary|1255 (line 193)
Precision of target insufficient for number of integral digi
\cts assigned to it.
system handler for error returns to command level
r 08:23 0.420 40 level 2
```

```
probe
Condition size raised at line 193 of print_summary (level 8).
source
          put skip (2)
               edit ("number of issues stocked:",
               total_number_stocked,
               "number of requests pending:",
               total_number_pending,
               "total stock value:",
               total_stock_value,
               ".")
               (r (output_format_2))
               file (sysprint);
stack
 15          simple_command_processor|12265
 14          command_processor_|11070
 13          abbrev_|5336
 12          release_stack|10031
 11          unclaimed_signal|27064
 10          wall|4436
  9          wall|4407                                  error
  8          print_summary (line 193)                   size
  7          check_back_issues (line 70)
  6          simple_command_processor|12265
  5          command_processor_|11070
  4          abbrev_|5336
  3          listen_|10031
  2          project_start_up_|41747
  1          user_init_admin_|42452 (alm)
sc
          put skip (2)
               edit ("number of issues stocked:",
               total_number_stocked,
               "number of requests pending:",
               total_number_pending,
               "total stock value:",
               total_stock_value,
               ".")
               (r (output_format_2))
               file (sysprint);
v total_number_stocked
total_number_stocked = 4.30337e9
position "total_number_stocked ="
          total_number_stocked =
               total_number_stocked +
               issue_record.current_inventory;
before
Break set before line 95
quit
r 08:27 0.968 317 level 2
```

- THE PROGRAM WILL NOW STOP BEFORE LINE 95 IS REACHED
  - probe IS AUTOMATICALLY INVOKED
  - THE REQUEST LIST IS PROCESSED
    - IN THIS CASE, WE ARE PLACED AT probe REQUEST LEVEL
    - IF THE halt REQUEST WAS NEITHER IMPLICITLY NOR EXPLICITLY STATED, EXECUTION WOULD CONTINUE WITH LINE 95

```
rl
r 08:27 0.041 10

close_file back_issues
r 08:28 0.049 17

check_back_issues

from (specify vol:num):1:1
          to (specify vol:num):1:4
Stopped before line 95 of print_record. (level 8)
source
          total_number_stocked =
               total_number_stocked +
               issue_record.current_inventory;
v total_number_stocked
total_number_stocked = 4.30337e9
symbol total_number_stocked
 fixed bin (17) automatic
Declared in check_back_issues
q
r 08:30 0.441 115
```

- TECHNIQUE:

SCENARIO THREE: SIMPLE BREAK PROCESSING

- NOTE THE CONDITION: size
  - RESULT TOO BIG TO BE PLACED IN TARGET STORAGE LOCATION
  - DISAGREES WITH THE WAY THE PROGRAMMER THOUGHT TO USE IT
- FIND OUT WHERE THE VARIABLE IS ASSIGNED
  - SET BREAKPOINT THERE
  - LOOK AT VALUE OF VARIABLE AT THAT STATEMENT
- THE VARIABLE HAD A RANDOM VALUE IN IT
  - IT WAS NOT INITIALLY SET
  - THIS CARRIED THROUGH UNTIL IT BLEW UP AT THE OUTPUT FORMATTING

```
qedx
rcheck_back_issues.pl1
/dcl total_number_stocked/
dcl total_number_stocked fixed bin;
s/;/ init (0);/p
dcl total_number_stocked fixed bin init (0);
w
q
r 08:30 0.256 30

pl1 check_back_issues -sv2 -tb
PL/I 26a
r 08:31 3.549 111

close_file back_issues
r 08:31 0.057 8

check_back_issues

from (specify vol:num):1:1
          to (specify vol:num):1:4

volume   1 number   1
          stocked:  23 outstanding requests:  0 cost of
this issue:
  $5.00.  volume   1 number   2
          stocked:  30 outstanding requests:  2 cost of
this issue:
  $3.00.  volume   1 number   3
          stocked:  27 outstanding requests:  0 cost of
this issue:
  $3.00.  Error:  size condition by
>user_dir_dir>MEDmult>F21>doodle>bad_cbi$print_summary|1304
(line 193) Precision of target insufficient for number of
integral digits assigned to it.  system handler for error
returns to command level
r 08:32 0.467 22 level 2
```

SCENARIO THREE: SIMPLE BREAK PROCESSING

- YOUR TURN

  - THIS LOOKS LIKE THE SAME ERROR. IS IT? HOW DO YOU KNOW?

  - LIST THE STEPS YOU WOULD TAKE TO RESOLVE THIS ERROR.

```
pb
Condition size raised at line 193 of print_summary (level 8).
sc
          put skip (2)
               edit ("number of issues stocked:",
               total_number_stocked,
               "number of requests pending:",
               total_number_pending,
               "total stock value:",
               total_stock_value,
               ".")
               (r (output_format_2))
               file (sysprint);
v total_number_stocked
total_number_stocked = 80
v total_number_pending
total_number_pending = 4.30337e9
v total_stock_value
total_stock_value = 286
sb total_number_pending
 fixed bin (17) automatic
Declared in check_back_issues
q
r 08:33 0.386 79 level 2

qx
rcheck_back_issues.pl1
/dcl total_number_pending/
dcl total_number_pending fixed bin;
s/;/ init (0);/p
dcl total_number_pending fixed bin init (0);
w
q
r 08:34 0.216 35 level 2

pl1 check_back_issues -tb -sv2
PL/I 26a
r 08:34 3.462 219 level 2

rl
r 08:34 0.040 6

close_file back_issues
r 08:35 0.042 7
```

- MORE REQUESTS

  - THE after REQUEST

    - SETS UP A BREAKPOINT AFTER A GIVEN STATEMENT IN A PROGRAM

    - OPERATES EXACTLY LIKE THE before REQUEST

    - USAGE:

      after

      a

      after <line>: (<request list>)

      a <line>: (<request list>)

    - EXAMPLES:

      after

      after 100

      a 50: (value x; value y)

  - THE status REQUEST

    - LISTS BREAKPOINTS YOU HAVE SET IN YOUR PROGRAMS

      - USAGE:

        status

        st

        status <program name>

        st <program name>

        status <line in current program>

st <line in current program>

status -all

st -all

status *

st *

- EXAMPLES:

  status

  status at 50

  status other-prog

  status -all

- NOTES:

  "status -all" LISTS ALL BREAKS SET

  "status *" LISTS THE NAMES OF PROGRAMS WITH BREAKS SET

  AN OPTIONAL CONTROL ARGUMENT OF "-long" IS ALLOWED, WHICH PRINTS THE PROBE REQUEST LIST ASSOCIATED WITH THE BREAKPOINT

- THE continue REQUEST

  - ALLOWS THE PROGRAMMER TO RESUME THE PROGRAM AFTER A BREAK THAT INVOKED probe

  - USAGE:

    continue

    c

  - EXAMPLES:

    continue

    c

- BACK TO THE SCENARIO

```
check_back_issues

from (specify vol:num):1:1
          to (specify vol:num):1:4

volume   1 number   1
      stocked:   23 outstanding requests:    0 cost of this issue:
  $5.00.
volume   1 number   2
      stocked:   30 outstanding requests:    2 cost of this issue:
  $3.00.
volume   1 number   3
      stocked:   27 outstanding requests:    0 cost of this issue:
  $3.00.

number of issues stocked:     80
number of requests pending:      2
total stock value:    $286.00.
r 08:36 0.264 7
```

- A NEW PROBLEM: NOT ENOUGH RECORDS PRINTED OUT

  - WITH NO CONDITION SIGNALLED, BREAKPOINTS ARE THE ONLY WAY TO GET INTO PROBE WHILE THE PROGRAM IS RUNNING

  - WITHOUT SETTING TOO MANY BREAKS, ATTEMPT TO STOP THE PROGRAM AT APPROPRIATE PLACES AND LOOK AT LOOP VALUES

```
probe check_back_issues
Using check_back_issues (no active frame).
ps "do issue"
          do issue = 1 to number_of_issues;
b
Break set before line 66
quit
r 08:39 0.342 137

check_back_issues
          from (specify vol:num):1:1

to (specify vol:num):1:4
Stopped before line 66 of check_back_issues. (level 7)

v number_of_issues
number_of_issues = 3

ps "number_of_issues ="
          number_of_issues =
               (6*last_issue_volume + last_issue_num) -
               (6*first_issue_volume + first_issue_num);

v last_issue_volume
last_issue_volume = 1

v last_issue_num
last_issue_num = 4

v first_issue_volume ; v first_issue_num
first_issue_volume = 1
first_issue_num = 1


position +2
               call print_record ();

after
Break set after line 67

continue
```

```
volume   1 number   1
      stocked:   23 outstanding requests:    0 cost of this issue:
  $5.00.Stopped after line 67 of check_back_issues. (level 7)
v issue
issue = 1
c

volume   1 number   2
      stocked:   30 outstanding requests:    2 cost of this issue:
  $3.00.Stopped after line 67 of check_back_issues. (level 7)
v issue
issue = 2
c

volume   1 number   3
      stocked:   27 outstanding requests:    0 cost of this issue:
  $3.00.Stopped after line 67 of check_back_issues. (level 7)
v issue
issue = 3
status
Break exists after line 67 in check_back_issues
Break exists before line 66 in check_back_issues
c


number of issues stocked:     80
number of requests pending:      2
total stock value:    $286.00.
r 08:46 1.185 380
```

- TECHNIQUE:
  - IDENTIFY THE LOOP THAT PRINTS THE RECORDS
  - SET A BREAK WITHIN THE LOOP AND CHECK THE LOOP VARIABLE
  - LOCATE THE STATEMENT AT WHICH THE LOOP VARIABLE WAS INCORRECTLY SET
  - FIX THE SOURCE

```
qedx
rcheck_back_issues.pl1
/number_of_issues/
dcl number_of_issues fixed bin;
//
          number_of_issues =
s/$/ 1 +/p
          number_of_issues =  1 +
w
q
r 09:09 0.270 80

pl1 check_back_issues -sv2 -tb
PL/I 26a
r 09:10 3.668 234

check_back_issues
from (specify vol:num):  1:1

to (specify vol:num):1:4

volume    1 number   1
           stocked:     23 outstanding requests:      0 cost
\cof this issue:
  $5.00.
volume    1 number   2
           stocked:     30 outstanding requests:      2 cost
\cof this issue:
  $3.00.
volume    1 number   3
           stocked:     27 outstanding requests:      0 cost
\cof this issue:
  $3.00.
volume    1 number   4
           stocked:     20 outstanding requests:      1 cost
\cof this issue:
  $3.00.

number of issues stocked:    100
number of requests pending:      3
total stock value:    $346.00.
r 09:10 0.362 28
```

I LOOKS GOOD; NOW TEST IT WITH DIFFERENT DATA.

```
check_back_issues
from (specify vol:num):  2:1

to (specify vol:num):3:1

volume   1 number   1
      stocked:   23 outstanding requests:    0 cost of this issue:
  $5.00.
volume   1 number   2
      stocked:   30 outstanding requests:    2 cost of this issue:
  $3.00.
volume   1 number   3
      stocked:   27 outstanding requests:    0 cost of this issue:
  $3.00.
volume   1 number   4
      stocked:   20 outstanding requests:    1 cost of this issue:
  $3.00.
volume   1 number   5
      stocked:   40 outstanding requests:    0 cost of this issue:
  $3.00.
volume   1 number   6
      stocked:   35 outstanding requests:    4 cost of this issue:
  $3.00.
volume   2 number   1
      stocked:   30 outstanding requests:    2 cost of this issue:
  $3.00.

number of issues stocked:    205
number of requests pending:      9
total stock value:    $661.00.
r.09:13 0.443 96
```

- ONE MORE BREAK CONTROL REQUEST

  - THE continue_to REQUEST

    - CAUSES probe TO CONTINUE EXECUTING, BUT ONLY UNTIL LINE SPECIFIED

    - USAGE:

      continue_to <line>

      ct <line>

    - EXAMPLES:

      continue_to 75

      ct +1

    - NOTES:

      THE FIRST EXAMPLE RESUMES EXECUTION OF THE PROGRAM AND STOPS IN probe AT LINE 75 OF THE PROGRAM

      THE SECOND EXAMPLE RESUMES EXECUTION, BUT STOPS BEFORE THE NEXT STATEMENT (I.E. EXECUTE ONE STATEMENT); SEE THE step REQUEST LATER

Not To Be Reproduced

```
probe check_back_issues
Using check_back_issues (no active frame).
position position_file
position_file:
     proc (first_vol, first_num);
a
Break set after line 129
status
Break exists after line 129 in check_back_issues
q
r 09:20 0.311 120

check_back_issues
from (specify vol:num):  2:1

to (specify vol:num):3:1
Stopped after line 129 of position_file. (level 8)
v first_vol
first_vol = 2
v first_num
first_num = 1
ps +3
     end;
a
Break set after line 146
continue_to 132
probe (continue_to): Using line 144 of check_back_issues instead.
Stopped before line 144 of position_file. (level 8)
sc
     do while (first_vol > current_volume);
v first_vol
first_vol = 2
v current_volume
current_volume = 4.30337e9
sb current_volume
 fixed bin (17) automatic
Declared in check_back_issues
q
r 09:25 0.831 265
```

```
qedx
rcheck_back_issues.pl1
/dcl (current_volume/
dcl (current_volume, current_number) fixed bin;
s/;/ init (0);/p
dcl (current_volume, current_number) fixed bin init (0);
w
q
r 09:27 0.288 83

pl1 check_back_issues -sv2 -tb
PL/I 26a
r 09:27 3.520 230

close_file back_issues
r 09:28 0.044 48
```

ADDITIONAL BREAK CONTROL

```
check_back_issues
          from (specify vol:num):2:1

to (specify vol:num):3:1

volume   2 number   2
      stocked:   36 outstanding requests:    1 cost of this issue:
  $3.00.
volume   2 number   3
      stocked:   46 outstanding requests:    7 cost of this issue:
  $3.00.
volume   2 number   4
      stocked:   31 outstanding requests:    0 cost of this issue:
  $3.00.
volume   2 number   5
      stocked:   36 outstanding requests:    0 cost of this issue:
  $3.00.
volume   2 number   6
      stocked:   33 outstanding requests:    1 cost of this issue:
  $3.00.
volume   3 number   1
      stocked:   47 outstanding requests:    5 cost of this issue:
  $3.00.
volume   3 number   2
      stocked:   50 outstanding requests:    4 cost of this issue:
  $3.00.

number of issues stocked:    279
number of requests pending:     18
total stock value:    $837.00.
r 09:28 0.395 30
```

- TWO MORE BREAK REQUESTS
  - THE reset REQUEST
    - DELETES SPECIFIED BREAKPOINTS

      (RESET BEING A HARDWARE TERM FOR TURNING A SWITCH OFF)

- USAGE:

  reset

  r

  reset <location>

  r <location>

  reset <program name>

  r <program name>

  reset -all

  r -all

  reset *

  r *

- EXAMPLES:

  reset after 75

  reset

  reset other-prog

  r *

- NOTES:

  THE "reset *" and "reset -all" ARE IDENTICAL: THEY BOTH DELETE ALL BREAKPOINTS IN ALL PROGRAMS

- THE step REQUEST

  - EXECUTES ALL THE INSTRUCTIONS UP TO, BUT NOT INCLUDING, THE NEXT STATEMENT

  - USAGE:

    step

s

I EXAMPLES:

step

s

I NOTES:

ACTS JUST LIKE EITHER OF THE FOLLOWING:

```
continue_to +1

before +1:(reset; halt)
```

IS DEFINITELY MORE CONVENIENT

```
pb check_back_issues
Using check_back_issues (no active frame).
ps position_file
position_file:
     proc (first_vol, first_num);
a
Break set after line 129
q
r 09:30 0.247 98

check_back_issues
from (specify vol:num):  2:1

to (specify vol:num):3:1
Stopped after line 129 of position_file. (level 8)

sk 8,2
  8           position_file (line 129)
  7           check_back_issues (line 58)

sc
position_file:
     proc (first_vol, first_num);

step
Stopped before line 144 of position_file. (level 8)

sc
     do while (first_vol > current_volume);

ps "return"
     return;

b
Break set before line 152
```

```
status
Break exists before line 152 in check_back_issues
Break exists before line 144 in check_back_issues
Break exists after line 129 in check_back_issues

reset 144
Break reset before line 144 in check_back_issues

continue
Stopped before line 152 of position_file. (level 8)

v current_volume
current_volume = 2

v current_number
current_number = 1

q
r 09:39 1.492 420
```

ADDITIONAL BREAK CONTROL

- TECHNIQUE

  - DETERMINE THAT CORRECT NUMBER OF RECORDS ARE BEING PRINTED OUT, BUT THAT THE STARTING POINT IS INCORRECT

  - DECIDE THAT THE PROGRAM "position_file" IS THE PROCEDURE THAT SETS THE STARTING POINT

  - SET BREAKPOINTS WITHIN "position_file" TO CHECK THE SETTING OF THE STARTING POINT

  - SEE THAT "position_file" POSITIONS TO THE FIRST DESIRED RECORD

  - CONCLUDE THAT "print_record" DOES NOT HAVE TO READ THE FIRST RECORD

```
qx
rcheck_back_issues.pl1
/call get_record/
            call get_record ();
i
            if have_used_record then
\f
/return/
            return;
-1

a
            have_used_record = "1"b;
\f
a

\f
/position_file/
position_file:
/return/
      return;
-1

a
have_used_record = "0"b;

\f
1n/dcl/
dcl back_issues file;
i
dcl have_used_record bit (1) aligned;
\f
w
q
r 09:43 0.637 104

pl1 check_back_issues -sv2 -tb
PL/I 26a
r 09:43 3.814 183
```

```
close_file back_issues
r 09:43 0.050 29

check_back_issues
          from (specify vol:num):2:1

to (specify vol:num):3:1

volume   2 number   1
      stocked:   30 outstanding requests:    2 cost of this issue:
  $3.00.
volume   2 number   2
      stocked:   36 outstanding requests:    1 cost of this issue:
  $3.00.
volume   2 number   3
      stocked:   46 outstanding requests:    7 cost of this issue:
  $3.00.
volume   2 number   4
      stocked:   31 outstanding requests:    0 cost of this issue:
  $3.00.
volume   2 number   5
      stocked:   36 outstanding requests:    0 cost of this issue:
  $3.00.
volume   2 number   6
      stocked:   33 outstanding requests:    1 cost of this issue:
  $3.00.
volume   3 number   1
      stocked:   47 outstanding requests:    5 cost of this issue:
  $3.00.

number of issues stocked:    259
number of requests pending:     16
total stock value:    $777.00.
r 09:44 0.459 14
```

A FEW MORE COMMANDS

- THE halt REQUEST

  - CAUSES PROBE TO BE ENTERED AT BREAKPOINT EXECUTION TIME

  - USAGE:

    halt

    h

  - EXAMPLES:

    halt

    h

  - NOTES:

    ONLY USEFUL IF NOT EXECUTING IN PROBE ALREADY

    A BREAKPOINT SET IN THE FORM OF "after" IS REALLY "after:halt"

- THE pause REQUEST

  - ACTS LIKE THE halt REQUEST, BUT ALSO RESETS THE BREAKPOINT

  - USAGE:

    pause

    p

- EXAMPLE:

  pause

  p

- NOTES:

  LIKE THE halt REQUEST, IS ONLY USEFUL AT A BREAKPOINT

  the BREAKPOINT "after:pause" IS EQUIVALENT TO "after:(halt;reset)"

- THE list_builtins REQUEST

  - LISTS THE BUILTIN FUNCTIONS AVAILABLE FROM WITHIN probe

  - USAGE:

    list_builtins

    lb

  - EXAMPLES:

    list_builtins

    lb

- THE list_help REQUEST

  - LISTS ALL THE HELP FILES AVAILABLE THROUGH THE probe "help" REQUEST

  - USAGE: list_help

    lh

Not To Be Reproduced

- EXAMPLES:

  list_help

  lh

  NOTES:

  YOU CAN GET MORE THAN JUST HELP ON THE REQUESTS: DESCRIPTIONS ARE ALSO PROVIDED FOR THE ARGUMENTS TO probe REQUESTS

SPECIFYING LINES

- SEVERAL PROBE REQUESTS ACCEPT LINE NUMBERS AS THEIR ARGUMENTS
  - before
  - after
  - reset
  - status
- SPECIFICATION OF A LINE CAN TAKE ON MANY FORMS
  - ABSOLUTE LINE NUMBER

    5

    100

    4-21
  - RELATIVE EXECUTABLE STATEMENT

    +1

+50

-5

I USING LABELS

get_record

place (3)

somewhere,4

$100

I SPECIAL SYMBOLS

$c

$c,7

$b

$b,3

TOPIC III

Other Source-Level Debugging Commands 3-1
The trace Command 3-1
Interaction of the Control Arguments. 3-5
Tracing Example One 3-6
Other trace Control Requests. 3-11
Trace Example Two 3-14
The display pllio error Command 3-20
A display pllio error Example 3-21

OBJECTIVES:

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

1. Add and remove procedures to and from the trace table.

2. Modify the tracing of a particular procedure in the trace table.

3. Use the trace command to perform metering on selected procedures.

4. Monitor recursion of selected procedures.

THE TRACE COMMAND

- trace COMMAND

  - SOURCE-LEVEL, PROCEDURE-CALL MONITOR

  - CAN BE USED WITH PROGRAMS WHICH DO NOT HAVE SYMBOL TABLES

  - CAPABILITIES INCLUDE

    - PRINTING ARGUMENTS AT PROCEDURE ENTRY AND/OR EXIT

    - EXECUTING A MULTICS COMMAND LINE AT PROCEDURE ENTRY AND/OR EXIT

    - STOPPING (BY CALLING THE COMMAND PROCESSOR) AT PROCEDURE ENTRY AND/OR EXIT

    - CONTROLLING THE FREQUENCY AT WHICH TRACING MESSAGES ARE PRINTED

    - WATCHING UP TO 16 STORAGE LOCATIONS FOR CHANGES AT EVERY PROCEDURE ENTRY AND/OR EXIT

  - LIMITATIONS

    - ONLY EXTERNAL PROCEDURES COMPILED BY PL/I OR FORTRAN CAN BE TRACED

    - ONLY USER-RING PROCEDURES CAN BE TRACED, NOT SUPERVISOR OR GATE PROCEDURES

- A PROCEDURE IN A BOUND SEGMENT CAN BE TRACED ONLY IF ITS ENTRY POINT HAS BEEN "RETAINED" IN THE BOUND SEGMENT

- USAGE

  trace -control_args

  OR

  trace procedure_names

  OR

  trace -control_args procedure_names

  - procedure_names GIVE THE PATHNAME OR REFERENCE NAME OF A PROCEDURE ENTRY POINT TO BE TRACED

    directory_path>entryname

    directory_path>entryname$entry_point_name

    reference_name

    reference_name$entry_point_name

  - control_args CONTROL THE TRACING FUNCTIONS PERFORMED ON THE TRACED PROCEDURE

- OPERATION

  - trace COUNTS

    - HOW MANY TIMES A PROCEDURE IS CALLED (N = NUMBER OF CALLS) IN THIS PROCESS SINCE COUNTERS WERE LAST RESET

    - HOW MANY TIMES A PROCEDURE IS MONITORED WHILE A PREVIOUS ACTIVATION STILL EXISTS (R = RECURSION DEPTH)

I OPERATION (Continued)

- I trace MONITORS A PROCEDURE CALL
  - I WHEN N AND R MEET CERTAIN CRITERIA
  - I BY PRINTING MONITORING MESSAGES

    Call N.R of PROCEDURE from CALLING_PROC, ap=244|1746.

    Return N.R from PROCEDURE.

  - I BY OPTIONALLY PRINTING PROCEDURE ARGUMENTS BEFORE ENTRY OR AFTER EXIT
  - I BY OPTIONALLY GOING TO Multics COMMAND LEVEL OR INVOKING A USER-SPECIFIED PROCEDURE BEFORE ENTRY OR AFTER EXIT

- I MONITORING CRITERIA
  - I ARE STORED IN A TRACE CONTROL TEMPLATE (TCT), AN INTERNAL STATIC DATABASE IN THE PROCESS DIRECTORY
  - I FOR EACH TRACED PROCEDURE ARE STORED IN TEMPLATES FASHIONED AFTER THE TCT
  - I IN THE TCT ARE PRINTED BY

    trace -template

  - I ARE SET BY GIVING A trace COMMAND WITH THE FOLLOWING control_args:

```
-first F, -ft F
     MONITOR WHEN F<=N

-last L, -lt L
     MONITOR WHILE N<=L

-every E, -ev E
     MONITOR EVERY Eth CALL (WHEN mod(N,E)=0)

-before B
     STOP BEFORE ENTERING PROCEDURE IF B^=0 AND mod(N,B)=0
     AND mod(N,E)=0

-after A
     STOP AFTER  EXITING PROCEDURE IF  A^=0 AND mod(N,A)=0
     AND mod(N,E)=0
```

-argument AG, -ag AG
PRINT ARGUMENTS IF AG^=0 AND mod(N,AG)=0 AND mod(N,E)=0

-in PRINT ARGUMENTS ONLY BEFORE ENTRY

-out PRINT ARGUMENTS ONLY AFTER EXIT

-inout
PRINT ARGUMENTS BEFORE ENTRY AND AFTER EXIT

-depth D, -dh D
MONITOR ONLY IF R<=D AND GOVERNING IS OFF

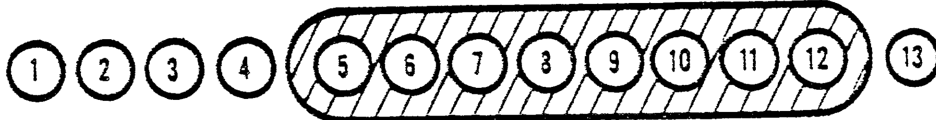
-return_value {on|off}, -rv {on|off}
PRINT FUNCTION RETURN VALUE AFTER EXIT

-govern {on|off}, -gv {on|off}
DISABLE RECURSION DEPTH CHECKING; INSTEAD, PRINT THE CALL MESSAGE ONLY WHEN THE RECURSION DEPTH REACHES A NEW MAXIMUM. ALSO, STOP WHEN RECURSION DEPTH IS A MULTIPLE OF 10 & A NEW MAXIMUM.

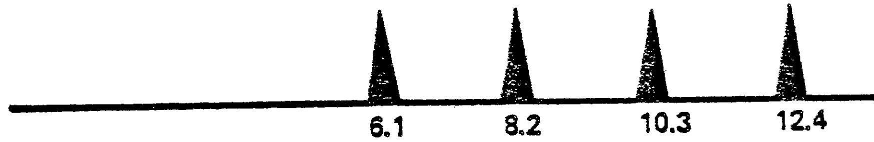
-meter {on|off}, -mt {on|off}
DISABLE MONITORING AND ENABLE PERFORMANCE METERING OF THE TRACED PROCEDURES

trace -first 5 -last 12 -every 2 -before 3 -argument 4

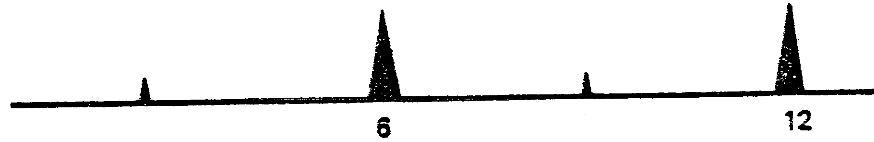
CALLS OF tt
(N)



MONITOR EVERY
2nd CALL
(-every 2)



STOP BEFORE
EVERY 3rd CALL
(IF IT'S MONITORED)
(-before 3)



PRINT INPUT
ARGUMENTS EVERY
4th CALL (IF IT'S
MONITORED)
(-argument 4)

TRACING EXAMPLE ONE

```
1  fact_: procedure (n) returns (fixed dec (12));
2  dcl (n, f, r) fixed dec (12);
3            if n <= 1 then r = 1;
4            else do;
5                 f = fact_ (n-1);
6                 r = f * n;
7            end;
8            return (r);
9       end fact_;
```

```
 1  factorial: procedure;
 2  dcl  result fixed dec (12);
 3  dcl  fact_ entry (fixed dec (12)) returns (fixed dec (12));
 4  dcl  n fixed dec (12);
 5  dcl  cleanup condition;
 6  dcl  (sysin, sysprint) file;
 7            open file(sysin) stream input,
 8                 file(sysprint) stream output
 9                      env(interactive);
10            on cleanup close file (sysin), file (sysprint);
11            get file (sysin) list (n);
12            do while (n >= 0);
13                 result = fact_ (n);
14                 put file (sysprint) list (result);
15                 get file (sysin) list (n);
16            end;
17            close file (sysin), file (sysprint);
18       end factorial;
```

TRACING EXAMPLE ONE

```
 1      !  pl1 fact_
 2         PL/I
 3         r 1720 1.381 21.776 148
 4
 5      !  pl1 factorial
 6         PL/I
 7         r 1720 0.964 1.332 36
 8
 9      !  factorial
10      !  3
11                       6
12      !  4
13                      24
14      !  5
15                     120
16      !  6
17                     720
18      !  10
19                 3628800
20      !  -1
21         r 1721 0.303 0.342 18
22
```

```
23      !  trace -template
24         first: 1, last: 9999999999, every: 1,
25         before: 0, after: 0, args: 0, depth: 9999999999,
26         meter: off, govern: off, return_value off
27         r 1721 0.067 0.042 6
28
29      !  trace -arguments 1 -out -return_value on
30         r 1721 0.038 0.002 1
31
32      !  trace fact_
33         r 1721 0.155 0.506 22
34
35      !  factorial
36      !  5    N.R
37         Call 1.1 of fact_ from factorial|235, ap = 244|5254
38         Call 2.2 of fact_ from fact_|43, ap = 244|5600
39         Call 3.3 of fact_ from fact_|43, ap = 244|6120
40         Call 4.4 of fact_ from fact_|43, ap = 244|6440
41         Call 5.5 of fact_ from fact_|43, ap = 244|6760
42         Return 5.5 from fact_
43         ARG  1 @ 244|6750 = 1
44         ARG  2 @ 244|6740 = 1
45         Return 4.4 from fact_
46         ARG  1 @ 244|6430 = 2
47         ARG  2 @ 244|6420 = 2
48         Return 3.3 from fact_
49         ARG  1 @ 244|6110 = 3
50         ARG  2 @ 244|6100 = 6
51         Return 2.2 from fact_
52         ARG  1 @ 244|5570 = 4
53         ARG  2 @ 244|5560 = 24
54         Return 1.1 from fact_
55         ARG  1 @ 244|5144 = 5
56         ARG  2 @ 244|5140 = 120
57                   120
58      !  -1
59         r 1721 0.395 1.632 47
60
```

TRACING EXAMPLE ONE

```
61      !  trace -every 3
62         r 1722 0.021 0.000 0
63
64      !  trace fact_
65         r 1722 0.024 0.000 0
66
67      !  factorial
68      !  10
69         Call 6.1 of fact_ from factorial|235, ap = 244|5254
70         Call 9.2 of fact_ from fact_|43, ap = 244|6100
71         Call 12.3 of fact_ from fact_|43, ap = 244|6720
72         Call 15.4 of fact_ from fact_|43, ap = 244|7540
73         Return 15.4 from fact_
74         ARG  1 @ 244|7530 = 1
75         ARG  2 @ 244|7520 = 1
76         Return 12.3 from fact_
77         ARG  1 @ 244|6710 = 4
78         ARG  2 @ 244|6700 = 24
79         Return 9.2 from fact_
80         ARG  1 @ 244|6070 = 7
81         ARG  2 @ 244|6060 = 5040
82         Return 6.1 from fact_
83         ARG  1 @ 244|5144 = 10
84         ARG  2 @ 244|5140 = 3628800
85                 3628800
86      !  -1
87         r 1722 0.210 0.002 1
88
```

TRACING EXAMPLE ONE

```
 89     !  trace -status fact_
 90        fact_:
 91            N = 15
 92            R = 0, max R = 0
 93            F = 1
 94            L = 9999999999
 95            E = 3
 96            B = 0
 97            A = 0
 98           AG = 1(o)
 99            D = 9999999999
100            return_value
101
102        r 1722 0.052 0.000 0
103
104     !  trace -status *
105             15.0 fact_
106        r 1723 0.027 0.000 0
107
108     !  trace -reset fact_ -status fact_
109        fact_:
110            N = 0
111            R = 0, max R = 0
112            F = 1
113            L = 9999999999
114            E = 3
115            B = 0
116            A = 0
117           AG = 1(o)
118            D = 9999999999
119            return_value
120
121        r 1723 0.055 0.006 2
122
123     !  factorial
124     !  4
125        Call 3.1 of fact_ from fact_|43, ap = 244|5560
126        Return 3.1 from fact_
127        ARG  1 @ 244|5550 = 2
128        ARG  2 @ 244|5540 = 2
129                     24
130     !  -1
131        r 1724 0.103 0.000 0
132
```

OTHER TRACE CONTROL REQUESTS

- OTHER trace CONTROL ARGUMENTS

  - CONTROL THE GENERAL OPERATION OF trace

  - INCLUDE

    -status procedure_name, -st procedure_name
        PRINTS THE TRACE CONTROL PARAMETERS AND COUNTERS FOR THE NAMED PROCEDURE

    -status *, -st *
        LISTS THE PROCEDURES BEING TRACED, THEIR INVOCATION COUNTS AND RECURSION DEPTHS

    -reset procedure_name, -rs procedure_name
        ZEROES THE INVOCATION COUNT OF THE GIVEN PROCEDURE

    -off procedure_name
        STOPS MONITORING THE GIVEN PROCEDURE; PROCEDURE REMAINS IN TRACE TABLE AND COUNTING OF INVOCATIONS CONTINUES, HOWEVER

    -on procedure_name
        RESUMES MONITORING THE GIVEN PROCEDURE

    -remove procedure_name, -rm procedure_name
        STOPS TRACING THE GIVEN PROCEDURE, DELETING ALL COUNTERS FOR THE PROCEDURE

  - THE procedure_name OPERAND FOLLOWING CONTROL ARGUMENTS MUST HAVE THE FORM:

    ```
    entryname

    entryname$entry_point_name

    reference_name

    reference_name$entry_point_name

    *
    ```

    THE CONTROL ARGUMENT APPLIES TO ALL TRACED PROCEDURES WHEN * IS GIVEN

```
133      ! trace -off fact_
134        r 1724 0.022 0.000 0
135
136      ! factorial
137      ! 12
138               479001600
139      ! -1
140        r 1725 0.072 0.000 0
141
142      ! trace -on fact_
143        r 1725 0.025 0.000 0
144
145      ! factorial
146      ! 2
147        Call 18.1 of fact_ from fact_|43, ap = 244|5420
148        Return 18.1 from fact_
149        ARG  1 @ 244|5410 = 1
150        ARG  2 @ 244|5400 = 1
151                      2
152      ! -1
153        r 1726 0.055 0.000 0
154
155      ! trace -status *
156              18.0 fact_
157        r 1726 0.028 0.000 0
158
159      ! trace -remove fact_
160        r 1726 0.027 0.002 1
161
162      ! trace -status *
163        trace: Trace table is empty.
164        r 1727 0.036 0.000 0
165
166      ! factorial
167      ! 10
168                 3628800
169      ! -1
170        r 1727 0.070 0.000 0
```

OTHER TRACE CONTROL REQUESTS

- OTHER trace CONTROL ARGUMENTS

  - CONTROL THE GENERAL OPERATION OF trace

    -brief, -bf
        SHORTENS THE MONITOR MESSAGES

    -long, -lg
        PRINTS LONGER MONITOR MESSAGES AGAIN

    -io_switch switch_name, -is switch_name
        PRINTS MONITOR MESSAGES ON THE NAMED I/O SWITCH, WHICH MUST BE ATTACHED & OPENED FOR STREAM_OUTPUT

    -execute command_line, -ex command_line
        EXECUTES THE COMMAND LINE WHENEVER A PROCEDURE IS MONITORED

    -stop_proc procedure_name, -sp procedure_name
        CHANGES THE PROCEDURE CALLED TO STOP BEFORE ENTRY OR AFTER EXIT TO THE GIVEN PROCEDURE

  - CONTROL PERFORMANCE MONITORING

    -meter {on|off}, -mt {on|off}
        STARTS/STOPS METERING OF TRACED PROCEDURES

    -total, -tt
        PRINTS PERFORMANCE MEASUREMENTS AND CLEARS THE METERING STATISTICS

    -subtotal, -stt
        PRINTS PERFORMANCE MEASUREMENTS BUT DOES NOT CLEAR THE METERING STATISTICS

  Now a Command

  - WATCH STORAGE LOCATIONS FOR CHANGES AS PROCEDURES ARE TRACED AND STOP IF THE LOCATIONS CHANGE

    -watch location, -wt location
        WATCHES THE ONE WORD LOCATION. UP TO 16 LOCATIONS CAN BE WATCHED AT ANY TIME. location HAS THE FORM:

        segment_number|offset

    -watch off, -wt off
        TURNS OFF THE WATCH FACILITY

TRACE EXAMPLE TWO

```
1  ! print tt.pl1 1
2  tt:  proc;
3  dcl  ioa_ entry options (variable);
4  dcl  d$ external static;
5  dcl  mod builtin;
6  dcl  cleanup condition;
7     on cleanup begin;
8          counter = 0;
9          call ioa_ ("counter initialized back to zero.");
10           goto bottom;
11       end;
12  dcl  counter fixed bin internal static init (0);
13       counter = counter +1;
14       call ioa_ ("..^i", counter);
15       if mod (counter, 5) = 0 then d$ = counter;
16       call tt;
17  bottom:
18        end tt;
19
20
21  ! trace -ft 5 -last 12 -every 2 -before 3 -argument 4 tt
22  ! trace -status tt
23  tt:
24            N = 0
25            R = 0, max R = 0
26            F = 5
27            L = 12
28            E = 2
29            B = 3
30            A = 0
31           AG = 4
32            D = 9999999999
33
34  ! trace -template
35  first: 5, last: 12, every: 2, before: 3, after: 0, args: 4,
36  depth: 9999999999, meter: off, govern: off, return_value off
37  ! tt
38  ..1
39  ..2
40  ..3
41  ..4
42  ..5
43  Call 6.1 of tt from tt|113, ap = 244|5476
44  trace: stop before
45  ! hmu
46
47  Multics MR6.5+, load 32.0/150.0; 40 users
48  Absentee users 0/4
49
50  ! start
51  ..6
52  ..7
53  Call 8.2 of tt from tt|113, ap = 244|6156
54  No arguments.
55  ..8
56  ..9
57  Call 10.3 of tt from tt|113, ap = 244|6636
```

```
58  ..10
59  ..11
60  Call 12.4 of tt from tt|113, ap = 244|7316
61  No arguments.
62  trace: stop before
63  ! sr
64  ..12
65  ..13
66  ..14
67  ..15
68  ..16
69  ..17
70  ..18
71  ..19
72  ..20
73  QUIT
74  ! trace -status tt
75  tt:
76               N = 282
77               R = 4, max R = 0
78               F = 5
79               L = 12
80               E = 2
81               B = 3
82               A = 0
83              AG = 4
84               D = 9999999999
85
86  ! release -all
87  counter initialized back to zero.
88  Return 12.4 from tt
89  Return 10.3 from tt
90  Return 8.2 from tt
91  Return 6.1 from tt
92  ! trace -status tt
93  tt:
94               N = 282
95               R = 0, max R = 0
96               F = 5
97               L = 12
98               E = 2
99               B = 3
100              A = 0
101             AG = 4
102              D = 9999999999
103
104  ! .l tracerev
105  b tracerev trace -ft 1 -lt 9999999999 -ev 1 -before 0
106  -after 0 -ag 0 -dh 9999999999 -mt off -gv off -wt off
107  -return_value off
108  ! tracerev tt
109  ! trace -st tt
110  tt:
111              N = 282
112              R = 0, max R = 0
113              F = 1
114              L = 9999999999
```

```
115              E = 1
116              B = 0
117              A = 0
118             AG = 0
119              D = 9999999999
120
121  ! list_ref_names d
122
123    357
124  d
125  ! trace -watch 357|0
126  ! trace tt
127  ! tt
128  ..1
129  ..2
130  ..3
131  ..4
132  ..5
133  trace_print_: 357|0 = 000000000005
134  Call 288.6 of tt from tt|113, ap = 244|6556
135  trace: stop before
136  ! start
137  ..6
138  ..7
139  ..8
140  ..9
141  ..10
142  trace_print_: 357|0 = 000000000012
143  Call 293.11 of tt from tt|113, ap = 244|10576
144  trace: stop before
145  ! start
146  ..11
147  ..12
148  ..13
149  ..14
150  ..15
151  trace_print_: 357|0 = 000000000017
152  Call 298.16 of tt from tt|113, ap = 244|12616
153  trace: stop before
154  ! rl -all
155  counter initialized back to zero.
156  ! tracerev tt
157  ! trace -st tt
158  tt:
159              N = 298
160              R = 0, max R = 0
161              F = 1
162              L = 9999999999
163              E = 1
164              B = 0
165              A = 0
166             AG = 0
167              D = 9999999999
168
169  ! trace -reset tt
170  ! trace -govern on tt
171  ! tt
```

```
172  Call 1.1 of tt from command_processor_|3304 (read_list),
          ap = 244|4600
173  ..1
174  Call 2.2 of tt from tt|113, ap = 244|5056
175  ..2
176  Call 3.3 of tt from tt|113, ap = 244|5376
177  ..3
178  Call 4.4 of tt from tt|113, ap = 244|5716
179  ..4
180  Call 5.5 of tt from tt|113, ap = 244|6236
181  ..5
182  trace_print_: 357|0 = 000000000005
183  Call 6.6 of tt from tt|113, ap = 244|6556
184  ..6
185  Call 7.7 of tt from tt|113, ap = 244|7076
186  ..7
187  Call 8.8 of tt from tt|113, ap = 244|7416
188  ..8
189  Call 9.9 of tt from tt|113, ap = 244|7736
190  ..9
191  Call 10.10 of tt from tt|113, ap = 244|10256
192  trace: stop before
193  ! trace -brief tt
194  ! start
195  ..10
196  trace_print_: 357|0 = 000000000012
197  Call 11.11 of tt
198  ..11
199  Call 12.12 of tt
200  ..12
201  Call 13.13 of tt
202  ..13
203  Call 14.14 of tt
204  ..14
205  Call 15.15 of tt
206  ..15
207  trace_print_: 357|0 = 000000000017
208  Call 16.16 of tt
209  ..16
210  Call 17.17 of tt
211  ..17
212  Call 18.18 of tt
213  ..18
214  Call 19.19 of tt
215  ..19
216  Call 20.20 of tt
217  trace: stop before
218  QUIT
219  ! trace -watch off tt
220  ! sr
221  ! ready
222  r 1842 9.009 21.098 724 level 2, 51
223
224  ! sr
225  ..20
226  Call 21.21 of tt
227  ..21
```

```
228  Call 22.22 of tt
229  ..22
230  Call 23.23 of tt
231  ..23
232  Call 24.24 of tt
233  ..24
234  Call 25.25 of tt
235  ..25
236  Call 26.26 of tt
237  ..26
238  Call 27.27 of tt
239  ..27
240  Call 28.28 of tt
241  ..28
242  Call 29.29 of tt
243  ..29
244  Call 30.30 of tt
245  trace: stop before
246  QUIT
247  ! rl -all
248  counter initialized back to zero.
249  ! trace -st tt
250  tt:
251               N = 30
252               R = 0, max R = 30
253               F = 1
254               L = 9999999999
255               E = 1
256               B = 0
257               A = 0
258              AG = 0
259               D = 9999999999
260               govern
261
262  ! trace -govern off tt
263  ! trace -st tt
264  tt:
265               N = 30
266               R = 0, max R = 30
267               F = 1
268               L = 9999999999
269               E = 1
270               B = 0
271               A = 0
272              AG = 0
273               D = 9999999999
274
275  ! trace -reset tt
276  ! trace -st tt
277  tt:
278               N = 0
279               R = 0, max R = 30
280               F = 1
281               L = 9999999999
282               E = 1
283               B = 0
284               A = 0
```

TRACE EXAMPLE TWO

```
285          AG = 0
286           D = 9999999999
```

Not To Be Reproduced

■ display_pl1io_error COMMAND

- PRINTS ADDITIONAL INFORMATION ABOUT THE MOST RECENT ERROR CONDITION SIGNALLED BY THE PL/1 INPUT/OUTPUT FACILITY

- USAGE

  display_pl1io_error

  OR

  dpe

- PL/1 I/O ERROR CONDITIONS INCLUDE

  - endfile
  - key
  - name
  - record
  - transmit
  - undefinedfile

A DISPLAY PL1IO ERROR EXAMPLE

```
 1   write_file: procedure;
 2
 3   dcl  f file record output;
 4   dcl  rec1 char (10),
 5        rec2 char (30) varying;
 6   dcl  cleanup condition;
 7
 8
 9             on cleanup close file (f);
10             open file (f);
11
12             rec1 = "ABCDEfghij";
13             write file (f) from (rec1);
14
15             rec2 = "abcdeFGHIJ";
16             write file (f) from (rec2);
17
18             close file (f);
19
20        end write_file;
```

```
 1   read_file: procedure;
 2
 3   dcl  f file record input;
 4   dcl  sysprint file;
 5   dcl  rec1 char (10);
 6   dcl (cleanup, endfile) condition;
 7
 8             on cleanup close file (f), file (sysprint);
 9             open file (f),
10             file (sysprint) output stream env (interactive);
11
12             on endfile (f) go to DONE;
13             do while ("1"b);
14                  read file (f) into (rec1);
15                  put file (sysprint) list (rec1);
16             end;
17
18   DONE:
19             close file (f), file (sysprint);
20
21        end read_file;
```

A DISPLAY PL1IO ERROR EXAMPLE

```
 1  !  pl1 read_file -table
 2     PL/I
 3     r 1033 1.353 37.579 272
 4
 5  !  pl1 write_file -table
 6     PL/I
 7     r 1034 0.767 40.527 287
 8
 9  !  read_file
10
11     Error:  undefinedfile condition
12         by >udd>F19d>Friedman>read_file|177 (line 9)
13     occurred while doing I/O on file f
14     File cannot be opened: call to iox_$open fails.
15     system handler for error returns to command level
16     r 1034 0.169 5.250 74 level 2, 16
17
18  !  display_pl1io_error
19
20     Error on file f, status code: Entry not found.
21     Title: vfile_ f
22     Attributes: input notkeyed record sequential
23     Permanent attributes: input record
24     Error in opening or closing f
25     r 1035 0.069 1.470 37 level 2, 16
26
27  !  probe
28     Condition undefinedfile raised at line 9 of read_file.
29  !  source
30          open file (f),
31          file (sysprint) output stream env (interactive);
32  !  quit
33     r 1035 0.126 5.726 97 level 2, 16
34
35  !  release
36     r 1035 0.028 0.328 17
37
```

```
38  !  write_file
39
40     Error:  undefinedfile condition
41         by >udd>F19d>Friedman>write_file|133 (line 10)
42     occurred while doing I/O on file f
43     File cannot be opened:
44        input and output attributes conflict.
45     system handler for error returns to command level
46     r 1035 0.140 3.660 60 level 2, 16
47
48  !  dpe
49
50     Error on file f
51     Title: vfile_ f
52     Attributes: input output record
53     Permanent attributes: input output record
54     Error in opening or closing f
55     The output attribute conflicts with the input attribute.
56     r 1036 0.061 1.190 34 level 2, 16
57
58  !  new_proc
59     r 1037 0.184 8.372 91
60
61  !  write_file
62     r 1037 0.376 6.532 80
63
64  !  read_file
65
66     Error:  undefinedfile condition
67         by >udd>F19d>Friedman>read_file|177 (line 9)
68     occurred while doing I/O on file f
69     File cannot be opened:
70        input and output attributes conflict.
71     system handler for error returns to command level
72     r 1038 0.627 25.122 192 level 2, 16
73
74  !  new_proc
75     r 1039 0.236 4.830 69
76
```

```
 77  !  read_file
 78     ABCDEfghij
 79
 80     Error:  record condition
 81         by >udd>F19d>Friedman>read_file|241 (line 14)
 82     occurred while doing I/O on file f
 83     "read into(XX)":
 84         record in data set larger than variable XX.
 85     Type "start" to continue.
 86     Data will be truncated to record's length.
 87     system handler for error returns to command level
 88     r 1039 0.887 25.244 219 level 2, 16
 89
 90  !  dpe
 91
 92     Error on file f, status code: Record is too long.
 93     Title: vfile_ f
 94     Attributes: open input notkeyed record sequential
 95     Permanent attributes: input record
 96     Last i/o operation attempted: read into
 97     r 1040 0.106 3.540 59 level 2, 16
 98
 99  !  probe
100     Condition record raised at line 14 of read_file.
101  !  source
102                    read file (f) into (recl);
103  !  v recl
104        "\000\000\000
105     abcdeF"
106  !  quit
107     r 1041 0.368 11.650 192 level 2, 16
108
109  !  start
110     \000\000\000
111     abcdeF
112     r 1041 0.119 3.492 59
113
```

A DISPLAY PL1IO ERROR EXAMPLE

```
114  !  print_attach_table f
115     f                    (not attached)
116     r 1041 0.069 1.640 40
117
118  !  io_call attach f vfile_ f
119     r 1041 0.080 3.306 57
120
121  !  io open f sequential_input
122     r 1042 0.087 3.422 58
123
124  !  io read_length f
125     io_call: len=10.
126     r 1042 0.039 0.006 2
127
128  !  io read f 10
129     io_call: 10 characters returned. ABCDEfghij
130     r 1042 0.077 1.980 44
131
132  !  io read_length f
133     io_call: len=34.
134     r 1042 0.032 1.332 36
135
136  !  io read f 40
137     io_call: 34 characters returned. \000\000\000
138     abcdeFGHIJ\400#\400\000\000\000write_file
139     r 1042 0.050 1.980 44
140
141  !  io (close detach) f
142     r 1043 0.045 2.162 46
```

TOPIC IV

Advanced probe Requests 4-1
Introduction. 4-1
Scenario I - More probe Control 4-2
Control of Output Processing. 4-7
Scenario III - Program Manipualtion 4-9

OBJECTIVES:

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

1. Use the following probe requests to good advantage:

    modes

    if

    language (lng)

    display (ds)

    goto (g)

    where (wh)

    use

    call (cl)

    declare (dcl)

    list_variables (lsv)

- MORE ABOUT probe

  - MOST DEBUGGING CAN BE DONE USING THE TECHNIQUES DESCRIBED IN CHAPTER TWO

  - THE REQUESTS DESCRIBED HERE GIVE THE USER MUCH MORE CONTROL OVER THE PROBE ENVIRONMENT

  - THE CONTROL CAN BE THOUGHT OF AS COVERING DIFFERENT ASPECTS OF probe

    - CONTROL OF THE INTERACTION
      - modes
      - if
      - language
      - display

    - CONTROL OF PROGRAM USAGE
      - goto
      - where
      - use
      - call

    - CONTROL OF PROBE VARIABLES
      - declare
      - list_variables

■ NEW REQUESTS

- THE modes REQUEST

  - ALTERS THE WAY probe INTERACTS WITH THE PROGRAMMER

  - USAGE:

    modes

    mode

    modes <mode type> <mode value>

    mode <mode type> <mode value>

  - EXAMPLES:

    modes prompt on

    modes prompt_string "pb: "

    mode value_separator " is equal to "

  - NOTES:

    SUPPORTED MODES ARE:

      error_messages, em

      qualification, qf

      value_print, vp

      value_separator, vs

      prompt

      prompt_string

SCENARIO I - MORE PROBE CONTROL

- THE language REQUEST

  - ALLOWS THE PROGRAMMER TO LET probe INTERACT IN DIFFERENT DIALECTS

  - USAGE:

    language

    lng

    language <language>

    lng <language>

  - EXAMPLES:

    language

    lng

    language fortran

  - NOTES:

    THE LANGUAGES CURRENTLY SUPPORTED ARE pl1, fortran, AND cobol

- THE goto REQUEST

  - GIVES BETTER ERROR PROCESSING CONTROL

  - USAGE:

    goto <line>

    g <line>

  - EXAMPLES:

    goto 50

    g 1-23

Not To Be Reproduced

SCENARIO I - MORE PROBE CONTROL

g $c

g $b+2

- NOTES:
  - A TRICKY RQUEST TO USE
  - COMPILER OPTIMIZATION MAY NOT LET THE goto PERFORM AS IT SEEMS IT SHOULD
  - DEFINITELY MORE DEPENDABLE THAN COMMAND LEVEL start

- THE SCENARIO - BACK TO OUR PROGRAM

```
r 13:57 0.332 51

check_back_issues
          from (specify vol:num):1:1
          to (specify vol:num):2:1

Error:  conversion condition by
>user_dir_dir>FSOEP>Pandolf>wkd>check_back_issues|540 (line 48)
onsource = "1:", onchar = ":"
Invalid character follows a numeric field.
system handler for error returns to command level
r 13:57 0.469 44 level 2

pb
Condition conversion raised at
line 48 of check_back_issues (level 7).
modes prompt true
probe:  language
Current language is PL/I
probe:  sc
          first_issue_volume =
               substr (first_issue, 1, first_issue_delim);
probe:  modes prompt false
modes prompt on
probe:  modes prompt off
v substr (first_issue, 1, first_issue_delim)
"1:"
let first_issue_volume = "1"
v first_issue_volume
first_issue_volume = 1
q
r 14:01 0.431 61 level 2
```

SCENARIO I - MORE PROBE CONTROL

```
start

Error:  conversion condition by
>user_dir_dir>FSOEP>Pandolf>wkd>check_back_issues|540 (line 48)
onsource = "1:", onchar = ":"
Invalid character follows a numeric field.
system handler for error returns to command level
r 14:01 0.257 8 level 2

pb
Condition conversion raised at
line 48 of check_back_issues (level 7).
v first_issue_delim
first_issue_delim = 2
let first_issue_delim = 1
v substr (first_issue, 1, first_issue_delim)
"1"
q
r 14:03 0.244 1 level 2

start

Error:  conversion condition by
>user_dir_dir>FSOEP>Pandolf>wkd>check_back_issues|540 (line 48)
onsource = "1:", onchar = ":"
Invalid character follows a numeric field.
system handler for error returns to command level
r 14:03 0.235 0 level 2

pb
Condition conversion raised at line 48 of check_back_issues
(level 7).
v substr (first_issue, 1, first_issue_delim)
"1"
goto $c

Error:  conversion condition by
>user_dir_dir>FSOEP>Pandolf>wkd>check_back_issues|550 (line
50)
onsource = "2:", onchar = ":"
Invalid character follows a numeric field.
system handler for error returns to command level
r 14:05 0.348 2 level 2
```

THE PREVIOUSLY DESCRIBED value REQUEST CAN BE USED TO DISPLAY A NAMED STORAGE LOCATION

- THE display REQUEST
  - SHOWS ANY ACCESSIBLE LOCATION ON ONE OF FOUR FORMS
  - USAGE:

    display <address> <format> <count>

    ds <address> <format> <count>
  - EXAMPLES:

    display var-one octal 2

    ds 260|14430 pointer 1

    ds tmp_strng ascii 12
  - NOTES:

    FOUR MODES ARE AVAILABLE

    octal, o

    ascii, a, character, ch, c

    instruction, i

    pointer, ptr, its

    code

I THE SCENARIO

```
r 14:56 0.325 16

check_back_issues
from (specify vol:num):  1:1
          to (specify vol:num):1:4

Error:  illegal_procedure condition by
>user_dir_dir>FSOEP>Pandolf>wkd>check_back_issues$print_record|675
(line 96)
(while in pl1 operator real_to_real_tr)
referencing stack_4|6363 (in process dir)

r 14:56 1.145 34 level 2

pb
Condition illegal_procedure raised at
line 96 of print_record (level 8).
sc
          total_stock_value =
               total_stock_value +
               (issue_record.current_inventory*
               issue_record.cost_of_issue);
v cost_of_issue
 cost_of_issue = 5
v current_inventory
 current_inventory = 23
v total_stock_value
total_stock_value = (invalid decimal data)
v octal (total_stock_value)
040040040040040040040040040 -> uninitialized
v unspec (total_stock_value)
"000100000000100000000100000000100000000100000000100000000
100000000100000000100000"b
display total_stock_value a 8

display total_stock_value o 2  2 words
 040040040040 040040040040
q
r 14:59 0.928 72 level 2
```

- MORE TOOLS

  - THE where REQUEST

    - THIS REQUEST TELLS THE PROGRAMMER THE VALUES OF probe's TWO DEBUGGING POINTERS

    - USAGE:

      where

      wh

      where <pointer>

      wh <pointer>

    - EXAMPLES:

      where

      wh sc

      where control

    - NOTES:

      THE TWO POINTER SPECIFICATIONS ARE:

        source, sc

        control, ctl

      THE position AND use REQUESTS CHANGE THE VALUE OF THE SOURCE POINTER

    - THE use REQUEST

SCENARIO III - PROGRAM MANIPUALTION

- MOVES THE SOURCE POINTER TO A NEW LOCATION

- UNLIKE THE position REQUEST, THIS DOES NOT DISPLAY THE FINAL LOCATION

- USAGE:

  use

  use <absolute line number>

  use <relative line number>

  use level <number>

  use <program name>

  use <character string>

- EXAMPLES:

  use

  use level 5

  use 138

  use foo

  use +3

  use "vl = 5"

- NOTES: THIS REQUEST CANNOT BE USED WITHOUT THE TABLE OPTION

- THE call REQUEST

- INVOKES ANOTHER PROGRAM JUST AS IF IT HAD BEEN A SUBROUTINE CALL

- USAGE:

  call <program name> (<parameters>)

SCENARIO III - PROGRAM MANIPUALTION

I EXAMPLE:

```
call my_prog (arg1, arg2)
call com_err_ (code, "from probe")
```

I NOTES:

probe PERFORMS VALUE CONVERSION AS PART OF THE CALL

I THE NEXT EXAMPLE

```
pb check_back_issues
Using check_back_issues (no active frame).
ps get_record
get_record: proc ();
a:(sk;halt)
Break set after line 153
q
r 15:25 0.745 236

check_back_issues
from (specify vol:num):  1:1

to (specify vol:num):1:4
  9           get_record (line 153)
  8           print_record (line 88)
  7           check_back_issues (line 65)
  6           simple_command_processor|12265
  5           command_processor_|11070
  4           abbrev_|5336
  3           listen_|10031
  2           process_overseer_|40055
  1           user_init_admin_|42452 (alm)
Stopped after line 153 of get_record. (level 9)
where
line 153 in get_record (level 9)
Control at line 153 of get_record.
use level 8
sc
          call get_record ();
use level 7
where
line 65 in check_back_issues (level 7)
Control at line 153 of get_record.
sc
               call print_record ();
value issue
issue = 1
c
```

```
volume   1 number   1
            stocked:  23 outstanding requests:  0 cost
\cof this issue:
  $5.00.  9          get_record (line 153)
  8            print_record (line 88)
  7            check_back_issues (line 65)
  6            simple_command_processor|12265
  5            command_processor_|11070
  4            abbrev_|5336
  3            listen_|10031
  2            process_overseer_|40055
  1            user_init_admin_|42452 (alm)
Stopped after line 153 of get_record. (level 9)
v issue
issue = 2
v number_of_issues
number_of_issues = 3
quit
r 17:12 1.216 128
```

SCENARIO IV - PROBE VARIABLES

■ MANAGING YOUR OWN VARIABLES

- probe ALLOWS THE PROGRAMMER TO SET UP VARIBLES KNOWN TO PROBE ONLY, BUT AVAILABLE FOR USE DURING ALL OF probe's PROCESSING (BREAKPOINTS, ETC.)

  - ALMOST LIKE HAVING A PL/I INTERPRETER

- THE declare REQUEST

  - USAGE:

    declare <name> <type>

    dcl <name> <type>

  - EXAMPLES:

    dcl pb_counter_1 fixed

    dcl TOTPCT real

    dcl sum-calc comp-6 -force

  - NOTES:

    THREE DATA TYPES ARE SUPPORTED:

    fixed bin 35

    fixed, integer, int, comp-6

    float, real

    pointer, ptr

    USE THE -force CONTROL ARGUMENT TO REDEFINE A PROBE VARIABLE

    IF A probe VARIABLE IS THE SAME NAME AS A PROGRAM VARIABLE, PREFIX THE probe VARIABLE WITH A PERCENT SIGN

Not To Be Reproduced

SCENARIO IV - PROBE VARIABLES

- THE list_variables REQUEST

  - LISTS THE NAMES, DATA TYPES AND VALUES OF probe VARIABLES

  - USAGE:

    list_variables

    lsv

  - EXAMPLES:

    list_variables

    lsv

- ONE MORE REQUEST

- THE if REQUEST

  - CONDITIONALLY EXECUTES A SET OF probe REQUESTS

  - USAGE:

    if <conditional> : (<request list>)

  - EXAMPLES:

    if a=b : (value a; halt)

    if var1 = 4.56 : let var2 = 0

  - NOTES:

    CURRENT IMPLEMENTATION ALLOWS FOR ONLY SIMPLE EXPRESSION EVALUATION; USE THE help REQUEST TO CHECK ON NEW DEVELOPMENTS

    THE USE OF RELATIONAL OPERATORS IN THE EXPRESSION DEPENDS UPON THE LANGUAGE SPECIFIED TO probe

SCENARIO IV - PROBE VARIABLES

(E.G. PL/I USES =, FORTRAN USES .eq.)

I AN EXAMPLE

```
r 17:16 0.122 4

pb check_back_issues/Using check_back_issues (no active
frame). /ps get_record /get_record: proc (); declare
times_get_record_called fixed/list_variables
times_get_record_called fixed 0 a:(let
times_get_record_called = times_get_record_called + 1;call
ioa_ ("get_record called ^i times",
times_get_record_called);) Break set after line 153/q/r
17:20 0.390 85

check_back_issues
        from (specify vol:num):1:1
        to (specify vol:num):1:4 get_record called 1 times

volume   1 number   2
          stocked:  30 outstanding requests:  2 cost of
this issue:
  $3.00.get_record called 2 times

volume   1 number   3
          stocked:  27 outstanding requests:  0 cost of
this issue:
  $3.00.get_record called 3 times

volume   1 number   4
          stocked:  20 outstanding requests:  1 cost of
this issue:
  $3.00.get_record called 4 times

volume   1 number   5
          stocked:  40 outstanding requests:  0 cost of
this issue:
  $3.00.

number of issues stocked:    117
number of requests pending:      3
total stock value:    $351.00.
r 17:21 0.660 61
```

SCENARIO IV - PROBE VARIABLES

I THE let REQUEST

I ASSIGNS THE VALUE OF AN EXPRESSION TO A GIVEN VARIBLE

I USAGE:

```
let variable = expression

let cross_section = expression
```

I EXAMPLES:

```
let a  = 5 let array  (2,i) = a - 5  let substr (alpha,2,3) =
"abc"
```

```
r 18:08 0.156 4

pb check_back_issues Using check_back_issues (no active
frame). ps get_record get_record: proc (); ps "return"
         return;

b: if current_volume=last_issue_volume : if
current_number=last_issue_num :call ioa_ ("just positioned
to last desired record")
Break set before line 175
list_variables
times_get_record_called fixed 4
let times_get_record_called = 0
q
r 18:11 0.386 50

check_back_issues
from (specify vol:num): 1:3

to (specify vol:num):1:6

volume  1 number   4
        stocked: 20 outstanding requests: 1 cost of
this issue:
  $3.00. volume  1 number  5
        stocked: 40 outstanding requests: 0 cost of
this issue:
  $3.00.just positioned to last desired record

volume  1 number  6
        stocked: 35 outstanding requests: 4 cost of
this issue:
  $3.00. volume  2 number  1
        stocked: 30 outstanding requests: 2 cost of
this issue:
  $3.00.

number of issues stocked:   125
number of requests pending:    7
total stock value:   $375.00.
r 18:11 0.795 24
```

SCENARIO IV - PROBE VARIABLES

```
pb check_back_issues
Using check_back_issues (no active frame).
ps get_record
get_record: proc ();
a: (let times_get_record_called = times_get_record_called + 1;
v times_get_record_called)
Break set after line 153
q
r 18:13 0.245 2


check_back_issues
from (specify vol:num):  1:3

to (specify vol:num):1:6
1
2
3
4

volume    1 number   4
             stocked:  20 outstanding requests:  1 cost of this
issue:
  $3.00.5

volume    1 number   5
             stocked:  40 outstanding requests:  0 cost of this
issue:
  $3.00.6 just positioned to last desired record

volume    1 number   6
             stocked:  35 outstanding requests:  4 cost of this
issue:
  $3.00.7

volume    2 number   1
             stocked:  30 outstanding requests:  2 cost of this
issue:
  $3.00.

number of issues stocked:    125
number of requests pending:      7
total stock value:    $375.00.
r 18:14 1.105 6
```

SCENARIO IV - PROBE VARIABLES

```
pb check_back_issues
Using check_back_issues (no active frame).
st
Break exists after line 153 in check_back_issues
Break exists before line 175 in check_back_issues
ps 153
get_record: proc ();
st at 153
Break exists after line 153 :let times_get_record_called =
times_get_record_called + 1; v times_get_record_called

a:  (let times_get_record_called = times_get_record_called +
1;call ioa_ ("get_record called ^d times",
times_get_record_called))
Break set after line 153
ps 1
check_back_issues:
          proc;
a
Break set after line 1
a:let times_get_record_called = 0
Break set after line 1
q
r 18:17 0.450 2
```

TOPIC V

MULTICS USER RING RUNTIME STRUCTURES. 5-1
Introduction. 5-1
Supervisor Segments 5-3
The Stack Segment - stack_n 5-6
The area.linker Segment 5-11
Getting Space for Program Variables . 5-23

OBJECTIVES:

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

1. Describe some of the ways in which processes can be inadvertently destroyed.

2. Describe the functions of the following process_directory segments:

   dseg

   kst

   pds

   stack_1 - stack_7 (as appropriate)

   [unique.area.linker]

3. Describe the format of the following structures:

   linkage offset table (LOT)

   internal static offset table (ISOT)

   reference_name_table (RNT)

4. Name the sections of a standard Multics object segment and give the functions of each.

INTRODUCTION

- THERE IS NO CENTRALIZED LOCATION FOR ALL PROGRAM SUPPORT TABLES AND DATA IN MULTICS

- NATIVE MULTICS USES SEVERAL SEGMENTS TO MANAGE RUNTIME INFORMATION

- MOST ARE FOUND IN THE PROCESS DIRECTORY

  - THE PROCESS DIRECTORY IS CREATED FOR A USER AT LOGIN TIME

  - IT IS PART OF THE HIERARCHY, JUST AS THE HOME DIRECTORY IS

  - IT IS GIVEN A SHRIEK NAME AS ITS IDENTITY

- THESE TABLES ARE MODIFIABLE BY PROGRAMS IN A PROCESS

- THEIR MISUSE IS THE MAIN CAUSE OF PROCESS FAILURE

```
r 04:38 0.163 1

pd
>process_dir_dir>!BXNCwXCBBBBBBB
r 04:38 0.044 0

cwd [pd]
r 04:38 0.047 0

list

Segments = 7, Lengths = 0.

rew     0  !BBBJLFKcGzxlDq.temp.0326
rew     0  !BBBJLFKcGxPLpJ.area.linker
rew     0  stack_4
re      0  pit
        0  pds
        0  kst
        0  dseg

r 04:38 0.196 0
```

SUPERVISOR SEGMENTS

- dseg
  - DESCRIPTOR SEGMENT
  - RESIDES IN RING 0
  - USED BY THE HARDWARE TO CALCULATE MEMORY ADDRESSES
  - IS NOT ACCESSIBLE TO THE USER
- kst
  - KNOWN SEGMENT TABLE
  - RESIDES IN RING 0
- IS NOT ACCESSIBLE TO THE USER
  - IS USED INDIRECTLY BY VIRTUAL MEMORY OPERATIONS
    - IS AN ARRAY OF BLOCKS, EACH BLOCK CONTAINING INFORMATION ABOUT EACH SEGMENT THE PROCESS IS CAPABLE OF REFERENCING

- FOR A SEGMENT TO BE USED IN A PROCESS (E.G. PRINTED, EDITED, EXECUTED) IT MUST HAVE AN ENTRY IN THE kst

- IF A SEGMENT HAS AN ENTRY IN THE kst IT IS CONSIDERED "KNOWN"

- pds

  - PROCESS DATA SEGMENT

  - RESIDES IN RING 0

  - CONTAINS MANY THINGS THE SUPERVISOR WANTS TO KNOW ABOUT YOUR PROCESS

    - YOUR PROCESS ID

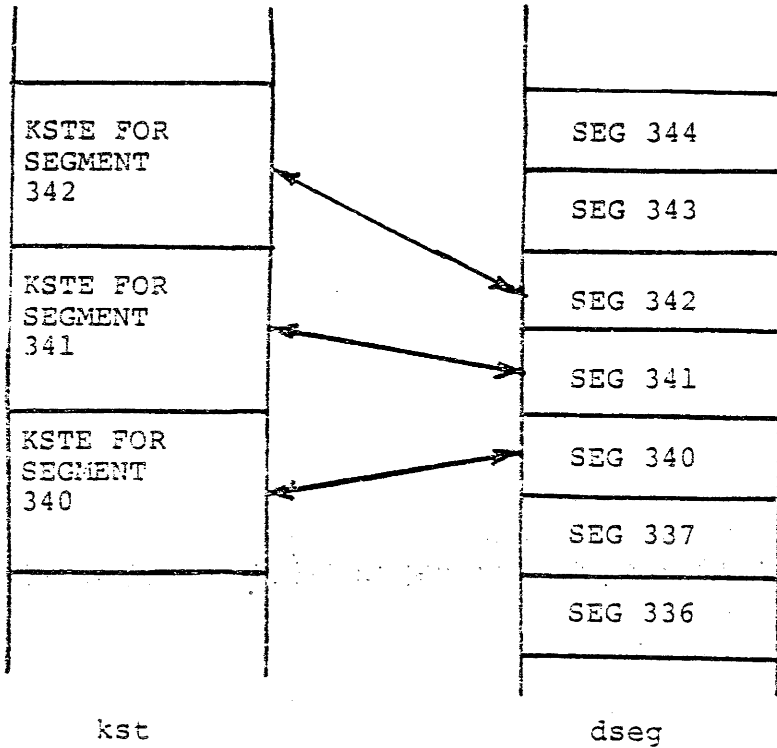
    - YOUR USER ID

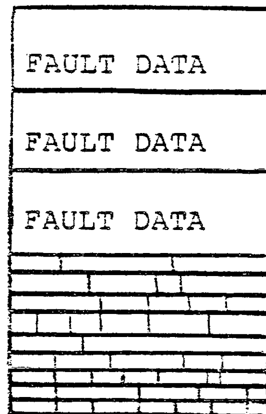
    - PROCESSOR INFORMATION FOR FAULT AND CONDITION PROCESSING

    - RING INFORMATION

    - MORE

  - NOT ACCESSIBLE TO THE USER





DATA CELLS

- USER STACK

  - FUNCTION

    - IS THE BACKBONE OF THE MULTICS PROGRAMMING ENVIRONMENT

    - CONTAINS VARIOUS TABLES, POINTERS, AND DATA

    - USED TO DEFINE THE REST OF THE PROGRAMMING ENVIRONMENT

  - THE STACK IS DIVIDED INTO TWO FUNCTIONAL PARTS

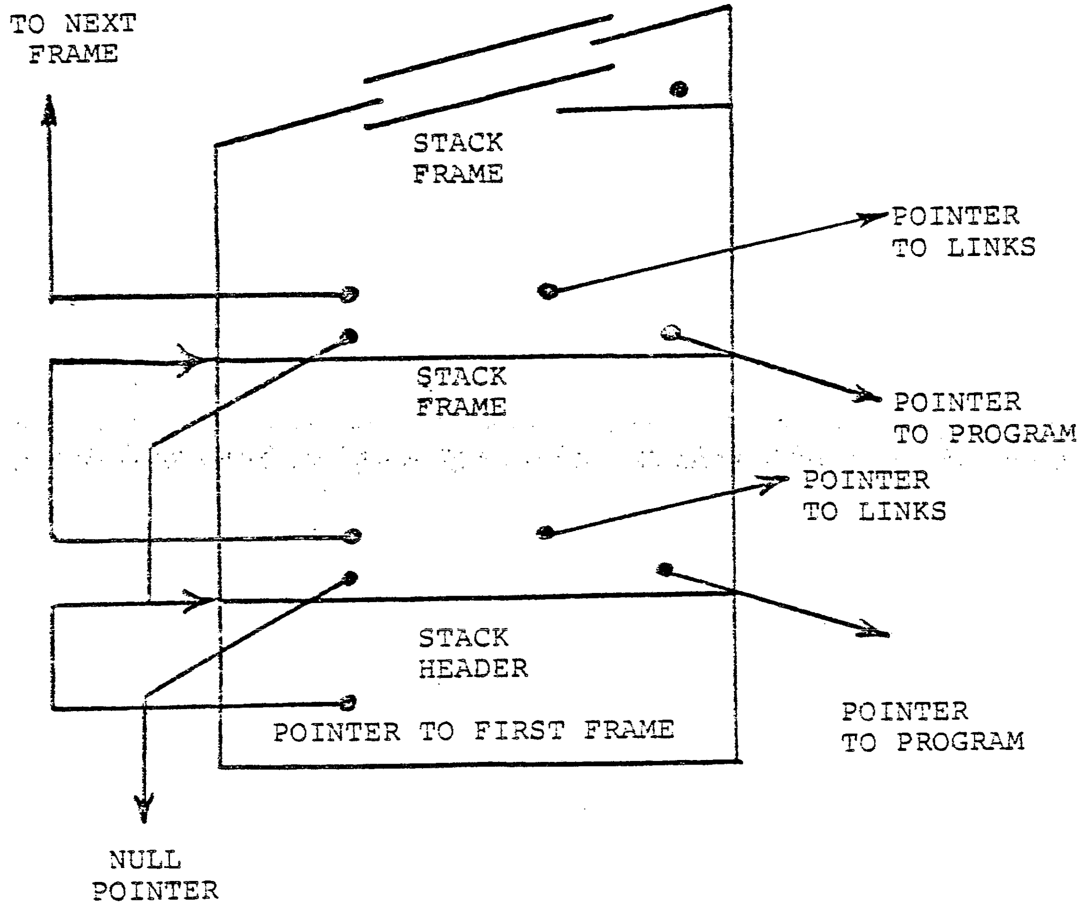
    - AT THE BEGINNING OF THE STACK IS THE STACK HEADER

    - THE HEADER CONTAINS POINTERS OF ALL THE OTHER TABLES USED

    - AT SOME POINT INTO THE STACK SEGMENT (DEPENDING ON THE STACK ITSELF) ACTIVATION FRAMES WILL BE FOUND

    - EACH FRAME CONTAINS INFORMATION ABOUT VARIABLES OF A CURRENTLY ACTIVE (CALLED, BUT NOT YET RETURNED) PROGRAM

  - THE SIZE OF THE STACK IS NOT PREDICTABLE, BECAUSE AS PROGRAMS ARE CALLED AND RETURN THE STACK WILL GROW AND SHRINK


TO NEXT
FRAME
STACK
FRAME
POINTER
TO LINKS
STACK
FRAME
POINTER
TO PROGRAM
POINTER
TO LINKS
STACK
HEADER
POINTER TO FIRST FRAME
POINTER
TO PROGRAM
NULL
POINTER

- THE LINKAGE OFFSET TABLE - THE LOT

  - IS USED BY THE DYNAMIC LINKER AND BY PROGRAMS TO FIND THEIR LINKAGE INFORMATION

  - IS QUITE SIMPLY AN ARRAY OF ONE WORD ADDRESSES SHOWING WHERE VARIOUS LINKAGE SECTIONS ARE

    - TO FIND OUT WHERE THE LINKAGE INFORMATION FOR A PROGRAM (CALL IT foo), FIRST OBTAIN ITS SEGMENT NUMBER

    - COUNT UP THAT MANY WORDS FROM THE BEGINNING OF THE LOT AND THE WORD AT WHICH YOU ARRIVE CONTAINS THE ADDRESS OF foo'S LINKAGE INFORMATION

  - THE LOT HAS AN INITIAL SIZE OF 512 WORDS AND IS ACTUALLY OVERLAID UPON THE BEGINNING OF THE STACK

- THE INTERNAL STATIC OFFSET TABLE - THE ISOT

  - THE ISOT CONTAINS ONE WORD ADDRESSES OF THE STATIC SECTIONS OF ALL THE ACTIVE PROGRAMS

  - IT TOO IS AN ARRAY OF THESE ADDRESSES

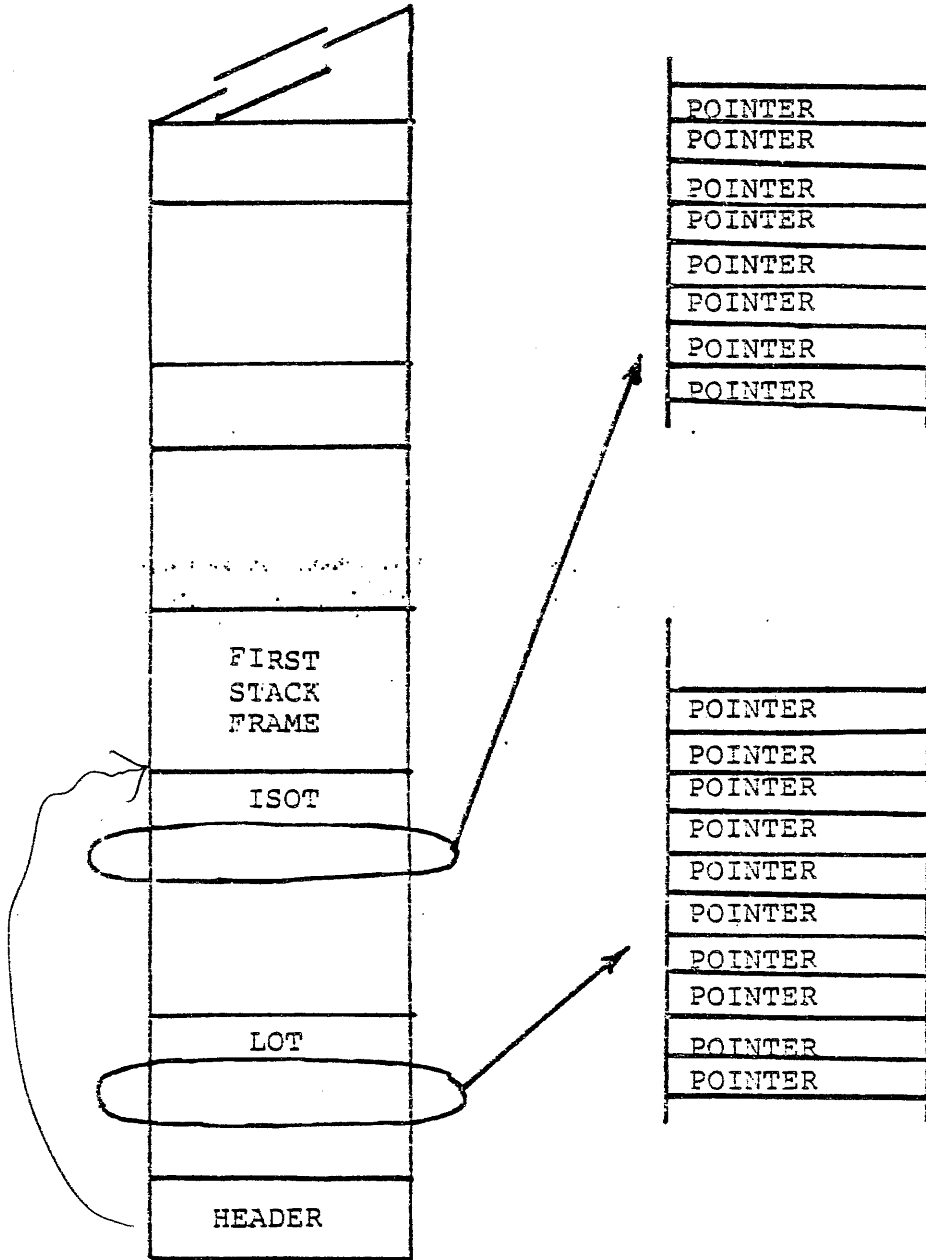
  - TO FIND THE LOCATION OF SOME PROGRAM'S STATIC SECTION, ONE COUNTS UP ITS SEGMENT NUMBER WORTH OF WORDS AS IN THE LOT

Not To Be Reproduced

THE STACK SEGMENT - STACK N

- THE SIZE OF THE ISOT IS ALSO 512 WORDS LONG AND IT IS FOUND RIGHT AFTER THE LOT ON THE STACK

  - BECAUSE OF THE LOT AND ISOT, THE FIRST STACK FRAME USUALLY BEGINS RIGHT AFTER THE ISOT


FIRST
STACK
FRAME
ISOT
LOT
HEADER
POINTER
POINTER
POINTER
POINTER
POINTER
POINTER
POINTER
POINTER
POINTER
POINTER
POINTER
POINTER
POINTER
POINTER
POINTER
POINTER
POINTER
POINTER

THE AREA.LINKER SEGMENT

- THE COMBINED LINKAGE AREA

  - AS WILL BE SEEN IN DYNAMIC LINKING, SOME INFORMATION FROM AN OBJECT SEGMENT NEEDS TO BE COPIED OUT INTO A WRITEABLE AREA

  - THE COMBINED LINKAGE AREA IS ONE OF THE TWO AREAS THAT HOLD THIS COPIED DATA

  - THE COMBINED LINKAGE AREA IS A PL/I TYPE AREA - A MANAGED POOL OF STORAGE FOR ALLOCATING AND FREEING DATA

  - HISTORICALLY THE COMBINED LINKAGE AREA WAS A PHYSICALLY SEPARATE AREA APART FROM OTHER RUNTIME AREAS.

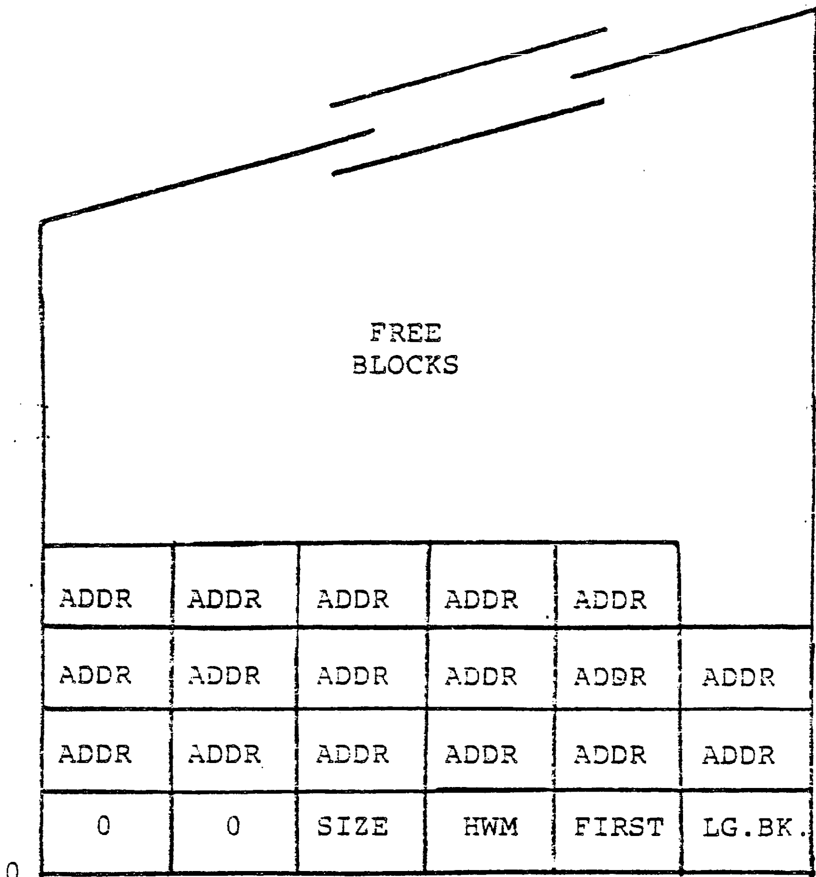
  - NOW IT IS JUST A "SYNONYM" FOR THE area.linker

  - THEREFORE THE WHOLE area.linker CAN BE THOUGHT OF AS THE COMBINED LINKAGE AREA

- THE COMBINED STATIC AREA

  - MODIFIABLE STATIC DATA IS MAPPED OUT IN THE OBJECT SEGMENT WHEN IT IS CREATED, BUT NEEDS TO BE MODIFIED

  - TO PREVENT THE MODIFICATION OF THE OBJECT ITSELF, THE STATIC DATA TEMPLATE IS COPIED FROM THE OBJECT TO THE COMBINED STATIC SECTION (COMBINED BECAUSE IT IS COMBINED WITH THE STATIC SECTIONS OF OTHER PROGRAMS)



VERSION
TWO
AREA

THE AREA.LINKER SEGMENT

- THE COMBINED STATIC AREA IS AGAIN A PL/I TYPE AREA

- THE COMBINED STATIC AREA IS ANOTHER SYNONYM FOR THE area.linker SEGMENT

- THE REFERENCE NAME TABLE - THE RNT

  - THE REPOSITORY FOR A SET OF ATTRIBUTES CALLED REFERENCE NAMES

    - A REFERENCE NAME IS AN ATTRIBUTE OF A SEGMENT FOR PROGRAMMING PURPOSES

    - A REFERENCE NAME EXISTS ONLY WITHIN A PROCESS - IT IS NOT PERMANENT

    - IT IS A SYNONYM FOR A SEGMENT THAT IS THE OBJECT OF A SEARCH

    - IT MAY OR MAY NOT BE RELATED TO THE ACTUAL NAME OF THE SEGMENT

    - IT IS CREATED IMPLICITLY OR EXPLICITLY

      - WHEN A PROGRAM IS CALLED IT IS GIVEN A REFERENCE NAME

      - WHEN THE RING 0 initiate PROGRAM IS CALLED (THROUGH A GATE, OF COURSE)

  - THE RNT ALSO MAINTAINS THE ASSOCIATION BETWEEN A REFERENCE NAME AND THE SEGMENT NUMBER OF A SEGMENT

THE AREA.LINKER SEGMENT

- THE RNT IS IN THE FORM OF A RATHER INVOLVED SERIES OF LINKED LISTS

  - THE RNT IS DEFINED BY A HEADER WHICH CONTAINS TWO HASH TABLES, ONE FOR SEGMENT NUMBERS AND ONE FOR REFERENCE NAMES

  - EACH ENTRY IN THE RNT IS IN TWO LINKED LISTS - A REFERENCE NAME LIST AND A SEGMENT NUMBER LIST

- THE RNT RESIDES IN THE MIDDLE PORTION OF THE area.linker SEGMENT

- IT MANAGES ITS OWN AREA IN THIS SEGMENT

■ THE USER FREE AREA

- USED FOR ALLOCATING CONTROLLED AND SOME BASED VARIABLES, FOR FORTRAN COMMON AND FOR COBOL DATA

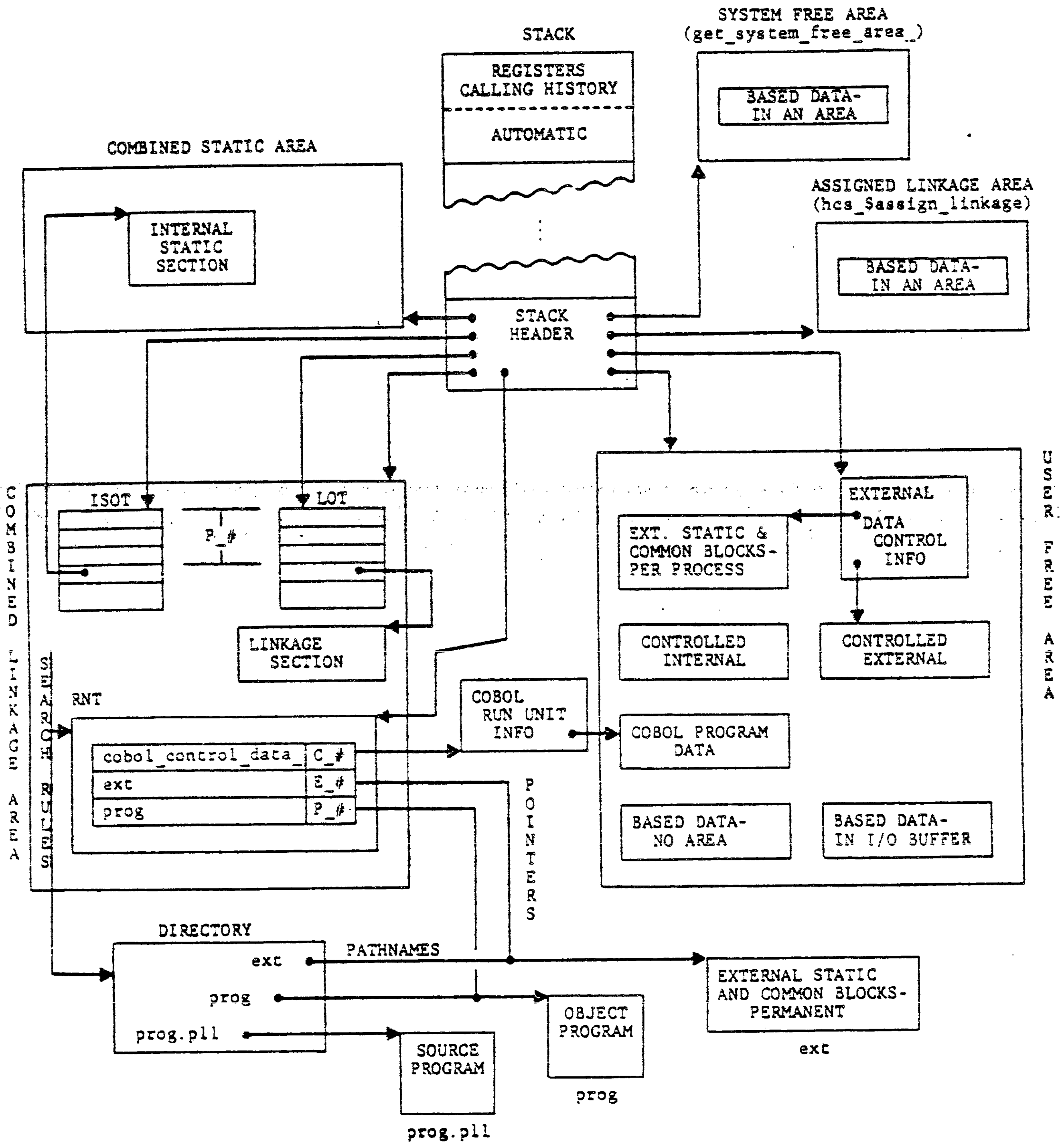
- OBVIOUSLY, BY NOW, IT IS A PL/I AREA

- THE USER FREE AREA ALSO IS A "SYNONYM" FOR THE area.linker SEGMENT

- TO RUN A PROGRAM THE USER MUST

  - CREATE AN OBJECT SEGMENT ACCEPTABLE TO THE MULTICS LINKERS

  - CALL THE PROGRAM



STACK
REGISTERS
CALLING HISTORY
AUTOMATIC
STACK HEADER
SYSTEM FREE AREA
(get_system_free_area_)
BASED DATA-
IN AN AREA
ASSIGNED LINKAGE AREA
(hcs_$assign_linkage)
BASED DATA-
IN AN AREA
COMBINED STATIC AREA
INTERNAL
STATIC
SECTION
COMBINED LINKAGE AREA
ISOT
LOT
P_#
LINKAGE
SECTION
SEARCH RULES
RNT
cobol_control_data
C_#
ext
E_#
prog
P_#
COBOL
RUN UNIT
INFO
POINTERS
USER FREE AREA
EXTERNAL
DATA
CONTROL
INFO
EXT. STATIC &
COMMON BLOCKS-
PER PROCESS
CONTROLLED
INTERNAL
CONTROLLED
EXTERNAL
COBOL PROGRAM
DATA
BASED DATA-
NO AREA
BASED DATA-
IN I/O BUFFER
DIRECTORY
ext
prog
prog.pl1
PATHNAMES
EXTERNAL STATIC
AND COMMON BLOCKS-
PERMANENT
ext
OBJECT
PROGRAM
prog
SOURCE
PROGRAM
prog.pl1

- FROM ANOTHER PROGRAM

  ```
  call prog$entry ();
  ```

- OR, BY INVOKING IT AT COMMAND LEVEL

  ```
  prog$entry
  ```

- THE COMMAND LEVEL INVOCATION DOES NOT CALL IN THE DYNAMIC LINKER, WHILE THE CALL STAEMENT MAY

- THE OBJECT SEGMENT FUNCTIONS TO

- PROVIDE INSTRUCTIONS AND DATA IN THE MACHINE'S LANGUAGE

  - INSTRUCTIONS ARE CURRENTLY IN L68 MACHINE CODE

  - THE DATA IS PROVIDED FOR THE MULTICS LINKER, PRELINKER, BINDER, AND DEBUGGERS

- COMPOSED OF 7 SECTIONS

  - TEXT SECTION

  - DEFINITION SECTION

  - LINKAGE SECTION TEMPLATE

  - STATIC SECTION TEMPLATE (OPTIONAL)

  - SYMBOL SECTION

  - OBJECT MAP

  - OBJECT MAP POINTER

OBJECT SEGMENT

~~THE AREA.LINKER SEGMENT~~

- TEXT SECTION
  - PURE PART OF AN OBJECT PROGRAM
- CONTAINS:
  - INSTRUCTIONS (NO SELF-MODIFYING INSTRUCTIONS)
  - ENTRY SEQUENCES
  - READ-ONLY DATA
    relative pointers

- DEFINITION SECTION
  - NONEXECUTABLE, READ-ONLY SYMBOLIC INFORMATION
    - USED FOR DYNAMIC LINKING
    - USED FOR SYMBOLIC DEBUGGING
  - CONTAINS
    - DEFINITIONS
    - OFFSETS OF NAMED ENTITIES IN TEXT AND OTHER SECTIONS
    - DEFINITION HASH TABLE (OPTIONAL) TO EXPEDITE THE LINKER'S SEARCHES
    - SYMBOLIC NAMES OF EXTERNAL REFERENCES

THE AREA.LINKER SEGMENT

- LINKAGE SECTION TEMPLATE
  - INITIAL CONTENTS OF THE IMPURE, NONEXECUTABLE PART OF A PROGRAM
  - USED FOR DYNAMIC LINKING
- CONTAINS
  - UNSNAPPED LINKS TO EXTERNAL REFERENCES
  - DATA ALLOCATED ONCE PER-PROCESS (INTERNAL STATIC DATA)
  - COPIED INTO COMBINED LINKAGE AREA IN THE PROCESS DIR WHEN OBJECT SEGMENT IS FIRST REFERENCED

THE AREA.LINKER SEGMENT

- STATIC SECTION TEMPLATE

  - THE INITIAL CONTENTS OF IMPURE, NONEXECUTABLE DATA FOR OBJECT PROGRAM

  - DATA IS

    - ALLOCATED ONLY ONCE PER PROCESS

    - INITIALIZED ONLY ONCE PER PROCESS

  - USUALLY INCLUDED AS PART OF LINKAGE SECTION UNLESS THE -separate_static CONTROL ARGUMENT IS USED WHEN COMPILING THE PROGRAM

  - COPIED INTO COMBINED STATIC AREA IN PROCESS DIR WHEN OBJECT SEGMENT IS FIRST REFERENCED

THE AREA.LINKER SEGMENT

- SYMBOL SECTION

  - IS PURE

  - CONTAINS INFORMATION NOT BELONGING IN OTHER SECTIONS

    - USED FOR SYMBOLIC DEBUGGING

    - USED FOR OBJECT PROGRAM STATUS COMMANDS (SUCH AS pli)

    - INFORMATION DOCUMENTING CREATION OF OBJECT PROGRAM

    - RELOCATION INFORMATION

    - SOURCE SYMBOL NAMES AND STORAGE LOCATIONS (PRESENT ONLY IF -table OPTION SPECIFIED)

  - NOTE: IN THE CASE OF BOUND OBJECT, THIS SECTION MIGHT BE FURTHER STRUCTURED INTO A THREADED LIST OF VARIABLE LENGTH SYMBOL BLOCKS

THE AREA.LINKER SEGMENT

- OBJECT MAP

  - DEFINES THE LOCATION (OFFSET) AND LENGTH OF OTHER SECTIONS

- DEFINES OBJECT SEGMENT FORMAT

  - SINGLE (UNBOUND) OBJECT PROGRAM, OR

  - SEVERAL OBJECT PROGRAMS, BOUND TOGETHER

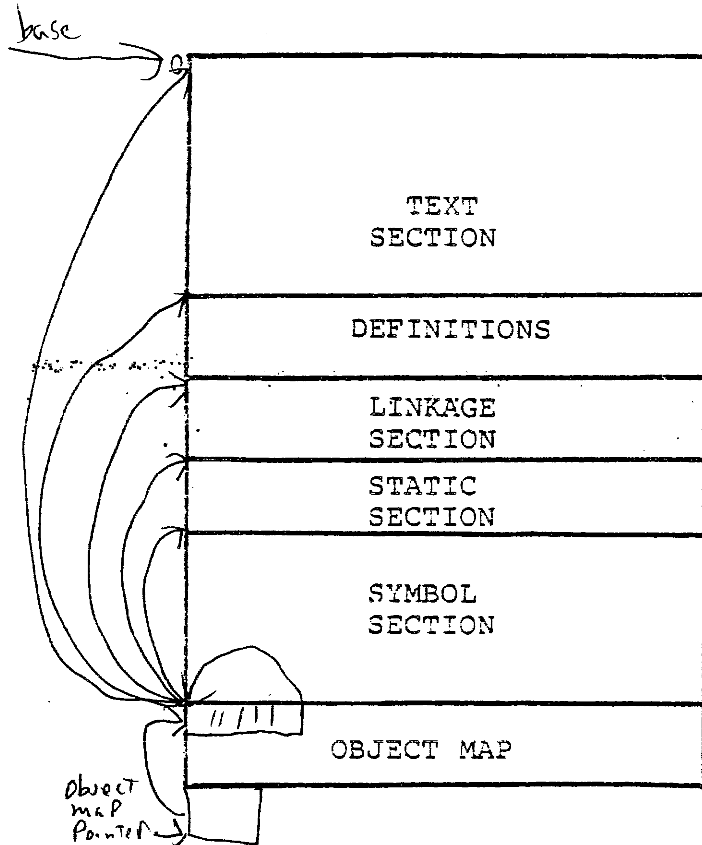
- OBJECT MAP POINTER

  - AN 18-BIT OFFSET IN THE UPPER HALF WORD OF THE LAST WORD IN THE OBJECT SEGMENT

  - GIVES LOCATION OF OBJECT MAP, RELATIVE TO BASE OF OBJECT SEGMENT

  - FOUND USING THE BIT COUNT

THE AREA.LINKER SEGMENT



GETTING SPACE FOR PROGRAM VARIABLES

- CONSTANTS

  - CONSTANT DATA VALUES, KNOWN ONLY TO ONE PROGRAM

  - DECLARATION

    - PL/1: dcl con fixed bin internal static options(constant) initial(3);

    - COBOL: CONSTANT SECTION.
      77 CON; PIC IS 99; VALUE IS 3.

    - FORTRAN: parameter con=3

  - LOCATION

    - IN THE TEXT SECTION OF THE PROGRAM

  - ALLOCATED AND INITIALIZED

    - ONCE BY THE COMPILER, WHEN THE SOURCE PROGRAM IS COMPILED

  - FREED

    - NEVER

GETTING SPACE FOR PROGRAM VARIABLES

- INTERNAL STATIC

  - PER PROCESS DATA, KNOWN ONLY TO ONE PROGRAM

  - DECLARATION

    - PL/1: declare is internal static;

    - FORTRAN: save is

  - LOCATION

    - IN INTERNAL STATIC SECTION OF PROGRAM, WHICH IS THEN COPIED TO [unique].area.linker

    - NOTE: SUCH VARIABLES CAUSE THE SIZE OF THE OBJECT TO GROW

  - ALLOCATED AND INITIALIZED

    - FIRST TIME OBJECT SEGMENT IS CALLED IN THE PROCESS BY COPYING OBJECT'S LINKAGE SECTION (OR SEPARATE STATIC SECTION)

  - FREED

    - WHEN PROCESS TERMINATES, OR WHEN OBJECT SEGMENT IS EXPLICITLY TERMINATED (terminate COMMAND)

GETTING SPACE FOR PROGRAM VARIABLES

- AUTOMATIC
  - PER PROGRAM-ACTIVATION DATA, KNOWN ONLY TO ONE PROGRAM
  - DECLARATION
    - PL/1: declare a automatic;
    - FORTRAN: automatic a
  - LOCATION
    - "ALLOCATED" IN STACK FRAME WHEN FRAME IS PUSHED
  - ALLOCATED AND INITIALIZED
    - EACH TIME PROGRAM IS CALLED
  - FREED
    - WHEN PROGRAM RETURNS

THE AREA LINKER SEGMENT

GETTING SPACE FOR PROGRAM VARIABLES

- EXTERNAL STATIC - PER PROCESS

  - PER PROCESS DATA, SHARED BETWEEN PROGRAMS, STORED IN TEMPORARY SEGMENTS IN THE PROCESS DIRECTORY

  - DECLARATION

    - PL/1: declare e external static;

    - FORTRAN: common b,c
      common /e/b,c

  - LOCATION

    - ALLOCATED IN USER FREE AREA

  - ALLOCATED AND INITIALIZED

    - WHEN FIRST REFERENCED

  - FREED

    - WHEN PROCESS TERMINATES, OR EXPLICITLY (SEE reset_external_variables AND delete_external_variables COMMANDS)

THE AREA.LINKER SEGMENT

GETTING SPACE FOR PROGRAM VARIABLES

- EXTERNAL STATIC - PERMANENT

  - PERMANENT DATA, SHARED BETWEEN PROGRAMS, STORED IN USER-SUPPLIED SEGMENTS

  - DECLARATION

    - PL/1: declare ext$ external static,
      ext$e external static;

    - FORTRAN: common /ext$/b,c
      common /ext$e/b,c

  - LOCATION

    - IN PERMANENT SEGMENT ext, FOUND BY LINKER (USING OBJECT SEARCH RULES)

    - SEGMENT MUST EXIST PRIOR TO EXECUTION

  - ALLOCATED AND INITIALIZED

    - WHEN SEGMENT ext IS CREATED

  - FREED

    - EXPLICITLY BY DELETING THE CONTAINING SEGMENT

THE AREA.LINKER SEGMENT

GETTING SPACE FOR PROGRAM VARIABLES

- CONTROLLED STORAGE - INTERNAL

  - EXPLICITLY-ALLOCATED DATA, KNOWN TO ONE PROGRAM

  - DECLARATION

    - PL/1:
      ```
      dcl c controlled int;  /* int is default */
      allocate c;
      allocate c;
      ```

  - LOCATION

    - ALLOCATED IN USER FREE AREA IN [unique].area.linker

  - ALLOCATED AND INITIALIZED

    - EXPLICITLY BY PL/1 allocate STATEMENT

  - FREED

    - EXPLICITLY BY PL/1 free STATEMENT

THE AREA.LINKER SEGMENT

GETTING SPACE FOR PROGRAM VARIABLES

- CONTROLLED STORAGE - EXTERNAL

  - EXPLICITLY-ALLOCATED DATA, SHARED BETWEEN PROGRAMS

  - DECLARATION

    - PL/1: declare ce controlled external;
      allocate ce;

  - LOCATION

    - ALLOCATED IN USER FREE AREA IN [unique].area.linker

  - ALLOCATED AND INITIALIZED

    - EXPLICITLY BY PL/1 allocate STATEMENT

  - FREED

    - EXPLICITLY BY PL/1 free STATEMENT

THE AREA.LINKER SEGMENT

GETTING SPACE FOR PROGRAM VARIABLES

- BASED - IN AN AREA

  - EXPLICITLY-ALLOCATED DATA, KNOWN ONLY TO ONE PROGRAM, QUALIFIED BY A LOCATOR

  - DECLARATION

    - PL/1:

```
dcl area area,
    b    based(p),
    p    ptr;
allocate b in (area);
```

  - LOCATION

    - DEPENDS WHERE THE USER SPECIFIES THE AREA TO BE (PERHAPS THE SYSTEM FREE AREA SUPPLIED BY INVOKING get_system_free_area_)

  - ALLOCATED AND INITIALIZED

    - EXPLICITLY BY PL/1 allocate STATEMENT

  - FREED

    - EXPLICITLY BY PL/1 free STATEMENT

THE AREA.LINKER SEGMENT

GETTING SPACE FOR PROGRAM VARIABLES

- BASED - NO AREA

  - EXPLICITLY-ALLOCATED DATA, KNOWN ONLY TO ONE PROGRAM, QUALIFIED BY A POINTER

  - DECLARATION

    - PL/1:
```
declare b based(p),
        (p,pl) ptr;
allocate b;
allocate b set(pl);
```

  - LOCATION

    - IN USER FREE AREA WITHIN [unique].area.linker

  - ALLOCATED AND INITIALIZED

    - EXPLICITLY BY PL/1 allocate STATEMENT

  - FREED

    - EXPLICITLY BY PL/1 free STATEMENT

THE AREA.LINKER SEGMENT

GETTING SPACE FOR PROGRAM VARIABLES

- BASED - IN AN I/O BUFFER

  - EXPLICITLY-ALLOCATED DATA, KNOWN ONLY TO ONE PROGRAM, QUALIFIED BY A POINTER

  - DECLARATION

    - PL/1:
      ```
      declare b based(p), f file;
      read file(f) set(p);
      locate b file(f) set(p);
      ```

  - LOCATION

    - IN AN I/O BUFFER ALLOCATED BY PL/1 IN USER FREE AREA IN [unique].area.linker

  - ALLOCATED

    - EXPLICITLY BY PL/1 read (WITH set OPTION) OR locate STATEMENT

  - INITIALIZED

    - BY locate STATEMENT

  - FREED

    - BY SUBSEQUENT I/O OPERATION ON THE FILE

GETTING SPACE FOR PROGRAM VARIABLES

- COBOL DATA

  - INTERNAL STATIC-LIKE DATA, KNOWN ONLY TO ONE PROGRAM

  - DECLARE

    - COBOL: WORKING SECTION.
      77 CB PIC IS 99.

  - LOCATION

    - ALLOCATED IN USER FREE AREA

  - ALLOCATED AND INITIALIZED

    - WHEN THE PROGRAM IS FIRST CALLED

  - FREED

    - WHEN THE PROCESS OR COBOL RUN UNIT TERMINATES, OR EXPLICITLY (BY A cancel_cobol_program COMMAND OR A CANCEL STATEMENT)

TOPIC VI

MULTICS DYNAMIC LINKING 6-1
Introduction. 6-1
Multics Compiler Conventions. 6-8
Mutlics Operating System Support. 6-11
The Linker - Phase I. 6-13
The Linker - Phase II 6-19
By-Products of Dynamic Linking. 6-36

OBJECTIVES:

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

1. Compare conventional linking with Multics dynamic linking.

2. List the functions performed by the Multics dynamic linker

3. Trace the operation of the dynamic linker from the time it first encounters an unsnapped link until it resolved the linkage fault.

4. Discuss the side-effects of dynamic linking and some of the dangers related to it.

5. Explain what happens when binding occurs and why it can be used to great advantage.

- LINKING

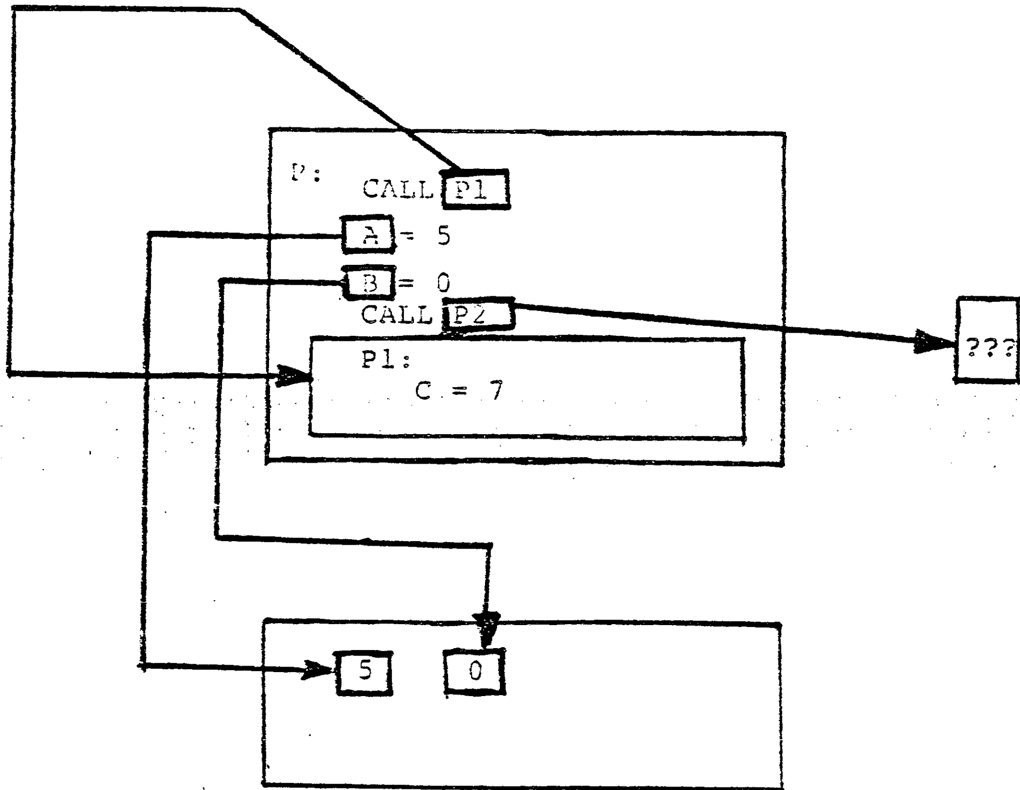
  - A LINKER IS BASICALLY A POST COMPILER

  - SYMBOLIC REFERENCES TO LOCATIONS NOT WITHIN THE OBJECT NORMALLY CAUSE THE TRANSLATOR TO PRINT AN ERROR MESSAGE

  - TELLING THE COMPILER OR ASSEMBLER THAT A SYMBOL IS EXTERNAL IS JUST A FUDGE SO THAT IT WON'T COMPLAIN

  - THE TRANSLATOR STILL CAN'T CALCULATE AN ADDRESS, SO IT BUILDS A STRUCTURE TELLING WHAT IT WAS LOOKING FOR

  - IT IS THE JOB OF THE LINKER TO RESOLVE ALL THESE UNLINKED REFERENCES


P:
CALL P1
A = 5
B = 0
CALL P2
P1:
C = 7
???
5
0

- THE TRADITIONAL LINKER

  - PHILOSOPHY

    - LINKING MUST BE DONE BEFORE THE PROGRAM RUNS

    - LINKING IS ASSOCIATED WITH LOADING - PLACING THE PROGRAM IN MEMORY

    - ALL EXTERNAL REFERENCES MUST BE PRESENT AT LINKING TIME

    - THE LINKER PRODUCES A "LOAD UNIT" OR "BOUND UNIT" - ABLE TO BE PLACED IN MAIN MEMORY WITH ALL ADDRESSES RESOLVED

    - THE LOAD UNIT IS THEN RUN BY THE USER

  - LIMITATIONS

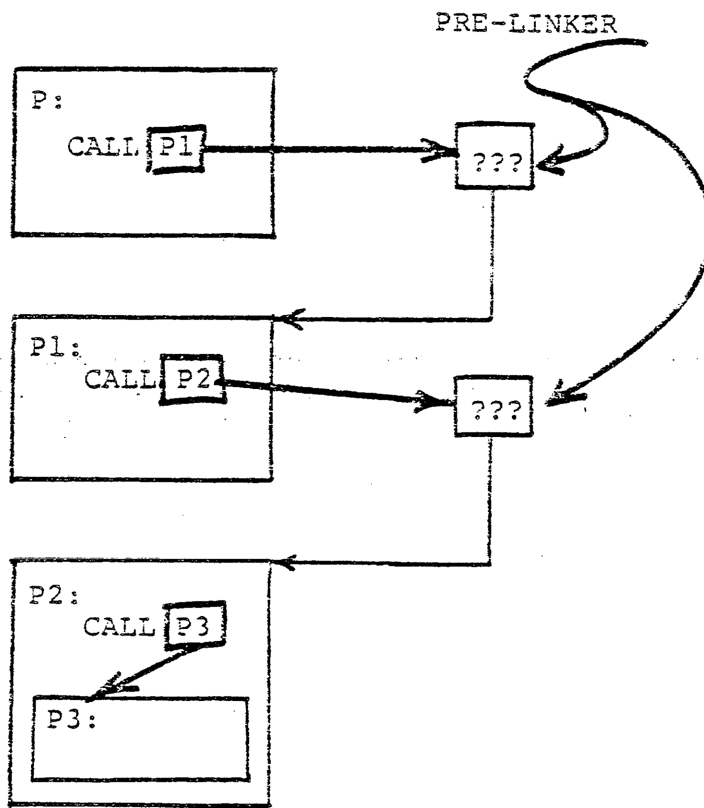
    - THE LINKER IS THE LAST STEP IN ADDRESS RESOLUTION - IF IT CAN'T DO IT, NOTHING CAN

    - BECAUSE TRADITIONAL LINKING IS ASSOCIATED WITH REAL MEMORY, ALL PROGRAMS HAVE TO BE PRESENT TO GET A SPOT IN THE BOUND UNIT, AND HENCE THE MEMORY

    - IF THE LINKER CAN'T RESOLVE A LINK, THE LINKING FAILS

    - CHANGING ONE PROGRAM IMPLIES LINKING EVERYTHING OVER AGAIN

    - SUBSTITUTING A PROGRAM MID-EXECUTION IS IMPOSSIBLE


PRE-LINKER
P:
CALL P1
???
P1:
CALL P2
???
P2:
CALL P3
P3:

- THE DYNAMIC LINKER

  - PHILOSOPHY

    - IN AN OPERATING SYSTEM WITH MANY (100 PLUS) SEGMENTS, LOADING AND RELOCATION BECOME PART OF THE JOB OF THE HARDWARE, NOT THE SOFTWARE

    - RESOLVING REFERENCES BECOMES EASIER BECAUSE THE TRANSLATOR GENERATES ADDRESSES RELATIVE TO THE BASE OF THE PROGRAM, AND HENCE THE SEGMENT

    - THE JOB OF THE LINKER HAS BEEN REDUCED TO FINDING A SEGMENT NUMBER AND AN OFFSET WITHIN THE SEGMENT TO COMPLETE THE EXTERNAL REFERENCE

    - WITH RELOCATION ALREADY TAKEN CARE OF, AND LINKING REDUCED TO THE CALCULATION OF TWO NUMBERS, LINKING CAN BE POSTPONED UNTIL THE EXTERNAL REFEENCE IS MADE

    - ALTHOUGH DYNAMIC LINKING IS POSSIBLE USING A REAL MEMORY, UNSEGMENTED MACHINE, THE BY-PRODUCTS OF SEGMENTATION MAKE IT WORTH WHILE TO IMPLEMENT

    - WITH THE DRUDGERY OF PRE-LINKING TAKEN AWAY FROM THE PROGRAMMER, S/HE CAN SPEND MORE TIME WORRYING ABOUT THE PROGRAM, AND NOT ABOUT THE MEMORY MANAGEMENT

  - LIMITATIONS

    - EXCESSIVE USE OF DYNAMIC LINKING CAN SLOW THE OVERALL SYSTEM THROUGHPUT

    - BY GIVING THE JOB OF LINKING TO THE OPERATING SYSTEM, THE PROGRAMMER HAS LESS SAY OVER WHICH VERSION OF A PROGRAM IS TO BE USED

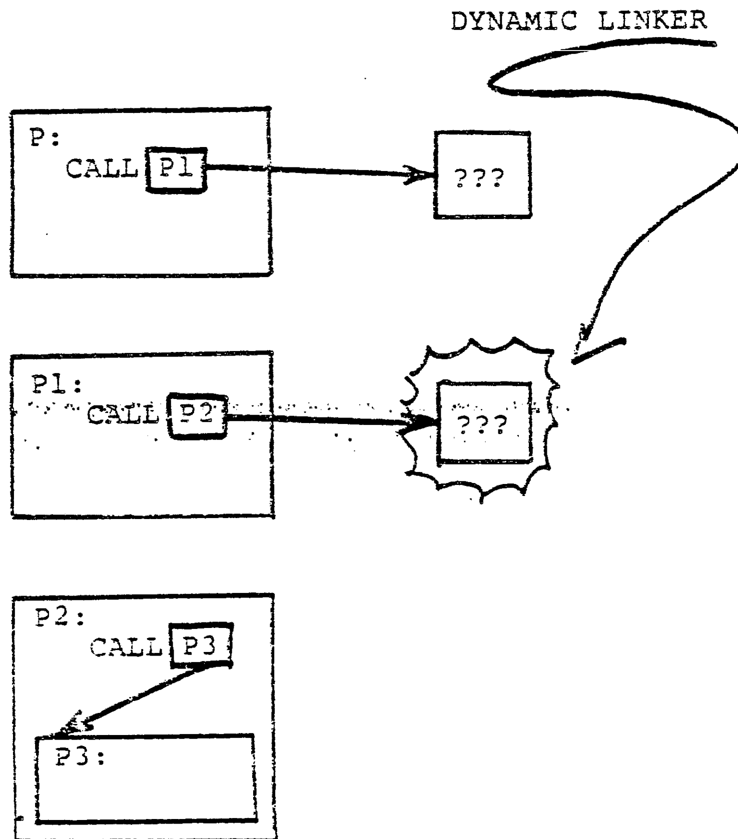
INTRODUCTION

I ADVANTAGES

- I PROGRAMS ARE EXECUTEABLE WITHOUT ALL EXTERNAL REFERENCES BEING PRESENT - ONE NEED ONLY WRITE AND DEBUG A SMALL PORTION AT A TIME

- I SUBSTITUTING SUBROUTINES CAN BE DONE AT RUNTIME WITH LITTLE EFFORT AND COST TO THE OPERATING SYSTEM

- I EXTERNAL REFERENCES THAT ARE AVOIDED BECAUSE OF TRANSFERS WITHIN A PROGRAM NEVER HAVE TO BE RESOLVED



Not To Be Reproduced

- EXTERNAL REFERENCES

  - ALL STANDARD MULTICS COMPILERS GENERATE THE SAME TYPE OF LINK THAT RELATES TO BOTH PRELINKING (BINDING) AND DYNAMIC LINKING

    - IS A REPOSITORY FOR THE EXTERNAL ADDRESS

    - CONTAINS INFORMATION TELLING ABOUT THE NAME OF THE EXTERNAL REFERENCE, INTERNAL LOCATIONS, WHETHER OR NOT TO CREATE IT IF NOT FOUND, ETC.

- THE LINK

  - INITIALLY CREATED IN THE LINKAGE SECTION OF THE OBJECT SEGMENT

  - ONE FOR EACH EXTERNAL REFERENCE

  - IS TWO WORDS LONG

    - MAY BE UNSNAPPED

      - THE END OF THE FIRST WORD CONTAINS THE "FAULT-TRAP 2" CODE

      - THE REST OF THE LINK CONTAINS ADDRESSES OF INFORMATION FOR THE LINKERS

    - MAY BE SNAPPED

      - THE END OF THE FIRST WORD CONTAINS THE "ITS" CODE - INDICATES A VALID MULTICS POINTER

- THE FIRST HALF OF THE FIRST WORD CONTAINS THE SEGMENT NUMBER OF THE EXTERNAL REFERENCE

- THE FIRST HALF OF THE SECOND WORD CONTAINS THE OFFSET WITHIN THE SEGMENT OF THE EXTERNAL REFERENCE

- THE LINKAGE SECTION

  - CONTAINS ALL THE LINKS A PROGRAM NEEDS

  - IS COPIED OUT OF THE OBJECT SEGMENT BEFORE THE OBJECT IS EXECUTED

  - IS MERGED INTO THE COMBINED LINKAGE AREA

<table>
<tr><td colspan="2">000000</td><td>DEFS</td><td>TRAP</td></tr>
<tr><td colspan="4">POINTER TO SYMBOL SECTION</td></tr>
<tr><td colspan="4">POINTER TO LINKS' ORIGIN</td></tr>
<tr><td colspan="4">UNUSED</td></tr>
<tr><td></td><td>LENGTH</td><td>SEGNO</td><td>LENGTH</td></tr>
<tr><td colspan="4">LINK</td></tr>
<tr><td colspan="4">LINK</td></tr>
</table>

⋮

- MANAGEMENT OF EXTERNAL REFERENCES

  - WITH FEW EXCEPTIONS, COMPUTERS EXPECT ADDRESSES TO BE IN NUMERIC (BINARY) FORM

  - THE MULTICS HARDWARE CURRENTLY FOLLOWS THIS ARCHITECTURE

  - THE DESIGNERS OF MULTICS DECIDED THAT IT WAS TIME TO MOVE AWAY FROM SOFTWARE ALSO USING NUMBERS

    - PROGRAMMERS STOPPED PROGRAMMING IN BINARY MACHINE LANGUAGE YEARS AGO, USING A MNEMONIC ASSEMBLY LANGUAGE

    - HUMANS REMEMBER AND MANIPULATE WORDS MORE EFFICIENTLY THAN NUMBERS

    - THE HARDWARE LOCATIONS OF DATA MAY CHANGE, BUT THE NAME WILL REMAIN THE SAME

    - SOFTWARE TECHNOLOGY NOW ALLOWS SYMBOLIC NAMES TO BE AUTOMATICALLY MANAGED CHEAPLY

  - MULTICS SUPPORTS SYMBOLIC (VIRTUAL) ADDRESSES FOR POTENTIALLY ALL THE DATA ON THE SYSTEM

    - SPECIFIED BY A TWO COMPONENT NAME IN THE FORM OF

      alpha$beta

      - alpha IDENTIFIES A SEGMENT

      - beta IDENTIFIES A LOCATION WITHIN THE SEGMENT

MUTLICS OPERATING SYSTEM SUPPORT

- MULTICS WORRIES ABOUT THE HARDWARE NUMBER ASSOCIATED WITH alpha

- MULTICS WORRIES ABOUT THE HARDWARE NUMBER ASSOCIATED WITH beta

- THE PROGRAMMER IS CONCERNED ONLY WITH REMEMBERING AND MANAGING THE ENTITY alpha$beta

- THIS SCHEME IS IMPLEMENTED EVEN AT THE MULTICS ASSEMBLY LANGUAGE LEVEL

- MULTICS MANAGES SEVERAL TABLES TO MAINTAIN THE ASSOCIATION BETWEEN A SYMBOLIC NAME AND ITS HARDWARE NUMBERS
  - dseg
  - kst
  - RNT
  - DEFINITION SECTION IN OBJECT SEGMENTS

- THERE IS AN INTERPLAY AMONG THE TABLES
  - dseg AND kst CAN BE CONSIDERED AS A NECESSARY PAIR
    - dseg TELLS THE HARDWARE WHERE SEGMENTS ARE
    - kst TELLS THE SOFTWARE WHO SEGMENTS ARE
    - THE RNT LISTS ALIASES FOR SEGMENTS LISTED IN THE kst

THE FAULT

- PROLOGUE

  - THE INSTANT A PROGRAM BEGINS TO RUN, IT HAS NOT YET MADE ANY CALLS

  - ITS LINKAGE SECTION WILL CONTAIN ONLY UNSNAPPED LINKS BECAUSE LINKS ARE NOT SNAPPED UNTIL NEEDED

  - AN ATTEMPT TO REFERENCE THROUGH ANY OF THESE LINKS WILL CAUSE A HARDWARE FAULT

THE LINKER - PHASE I

PR0	270	7030
PR1	110	322
PR2	244	4420
PR3	121	0
PR4	260	13200
PR5	77	3736
PR6	244	4420
PR7	244	0

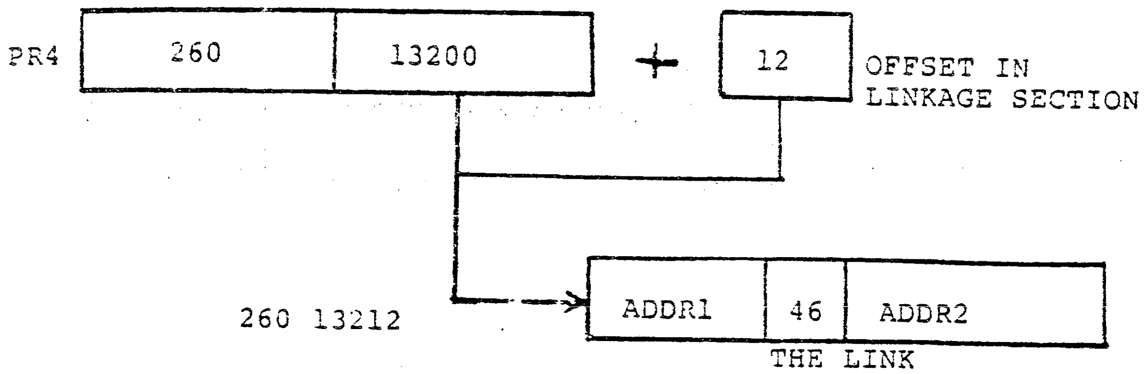
<table>
<tr><td colspan="2">000000</td><td>DEFS</td><td>TRAP</td></tr>
<tr><td colspan="4">POINTER TO SYMBOL SECTION</td></tr>
<tr><td colspan="4">POINTER TO LINKS' ORIGIN</td></tr>
<tr><td colspan="4">UNUSED</td></tr>
<tr><td></td><td>LENGTH</td><td>SEGNO</td><td>LENGTH</td></tr>
<tr><td colspan="4">LINK</td></tr>
<tr><td colspan="4">LINK</td></tr>
</table>

⋮

- THE FAULT HANDLER
  - A REFERENCE THROUGH A LINK IS DONE WITH HARDWARE INDIRECTION
    - ALL THE LINK DOES IS TELL THE HARDWARE WHERE TO GO FOR THE NEXT REFERENCE
    - USES AN ITS-PAIR
    - ITS STANDS FOR "INDIRECT-TO-SEGMENT"

  4/68

  - IF THE HARDWARE FINDS THE BIT PATTERN 100110 AT THE END OF THE FIRST WORD OF A LINK, IT FALLS INTO A FAULT TRAP 2.
  - THE WHOLE PROCESSING UNIT IS HALTED AND THE MACHINE IS FORCED TO EXECUTE THE PROGRAM fim fault intercept module
    - FROM THIS POINT ON, CONTROL IS IN THE HANDS OF THE SUPERVISOR
    - THIS IS RING ZERO
    - THERE IS NO WAY FOR THE USER TO INTERCEPT THIS FAULT
  - fim ASCERTAINS THAT THE FAULT WAS FAULT TRAP 2 AND CALLS THE PROGRAM link_snap_
    - link_snap_ IS THE DYNAMIC LINKER
  - link_snap FIRST VERIFYS THAT THIS IS A VALID UNSNAPPED LINK


PR4
260
13200
+
12
OFFSET IN
LINKAGE SECTION
260 13212
ADDR1
46
ADDR2
THE LINK

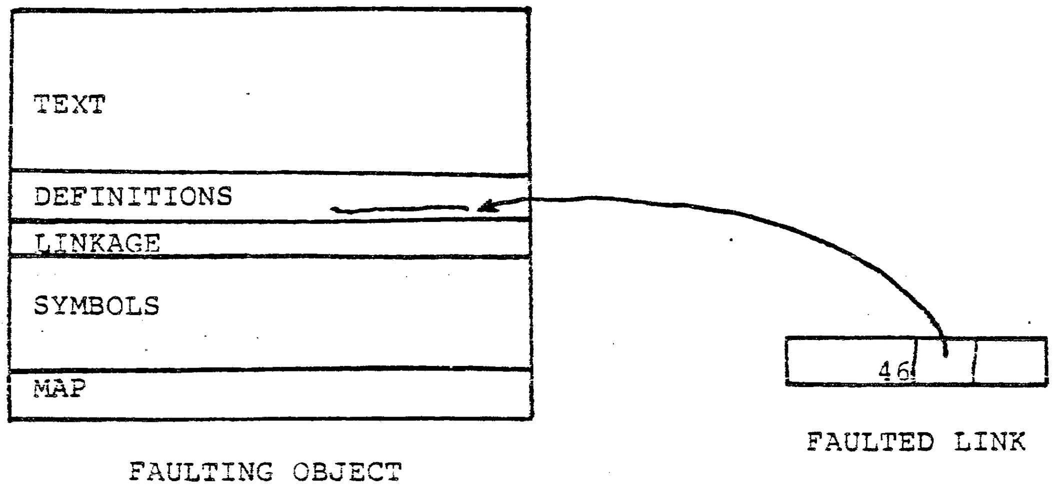
THE LINKER - PHASE I

- SEARCH FOR THE NAME

  - link_snap LOOKS AT THE LINK THAT CAUSED THE FAULT AND EXTRACTS FROM IT POINTERS BACK TO THE OBJECT SEGMENT

  - IN THE OBJECT, AS PART OF THE DEFINITION SECTION, ARE "OUTWARD-REFERENCING-SYMBOLS" THAT NAME THE SEGMENT AND LOCATION WITHIN IT THAT WE WANT

  - link_snap OBTAINS THE TWO SYMBOLIC NAMES THAT MAKE UP THE VIRTUAL ENTRY DESCRIBING THE FAULTED LINK FROM THIS LIST OF SYMBOLS



FAULTING OBJECT

- OBTAINING THE SEGMENT NUMBER

  - A SEGMENT NUMBER DESCRIBES TO HARDWARE THE LOCATION OF A REAL SEGMENT; THE DYNAMIC LINKER HAS TO FIND THAT REAL SEGMENT

  - THE SEARCH RULES

    - EACH PROCESS HAS A LIST OF PLACES TO SEARCH JUST IN CASE LINKAGE FAULTS OCCUR (WHICH IS INEVITABLE)

    - THIS IS THE USER MODIFIABLE ATTRIBUTE OF A PROCESS CALLED "THE SEARCH RULES"

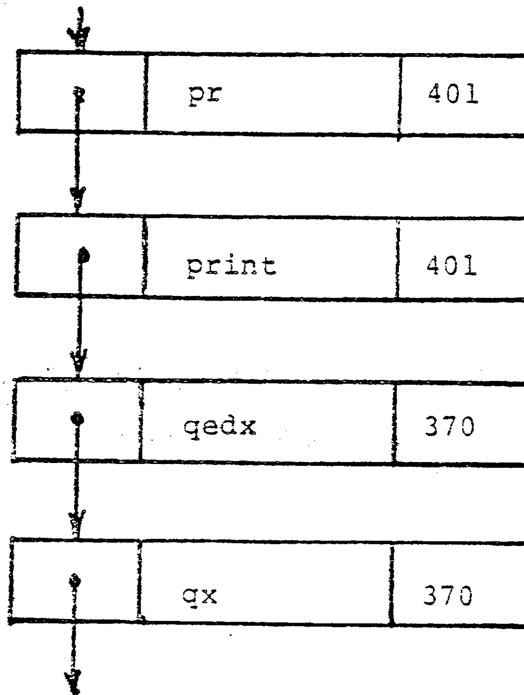
    - A SPECIAL CASE: initiated segments

      - THE LINKER GOES TO THE RNT

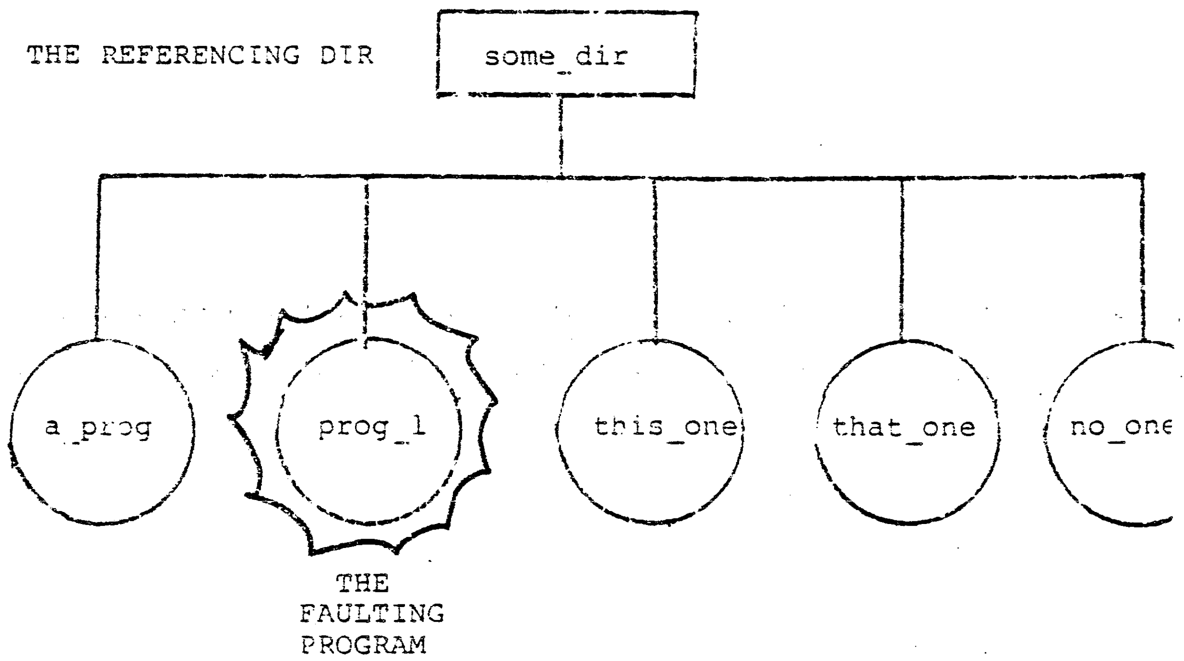
      - IT LOOKS UP THE NAME OF THE SEGMENT IT OBTAINED DURING PHASE I (THE RNT IS A BUNCH OF LINKED LISTS)

      - IF THE NAME IS FOUND, THEN THE NUMBER ASSOCIATED WITH IT IS ASSUMED TO BE THE DESIRED SEGMENT NUMBER

      - IF THE NAME WAS NOT FOUND, THEN WE MOVE ON TO THE NEXT SEARCH RULE

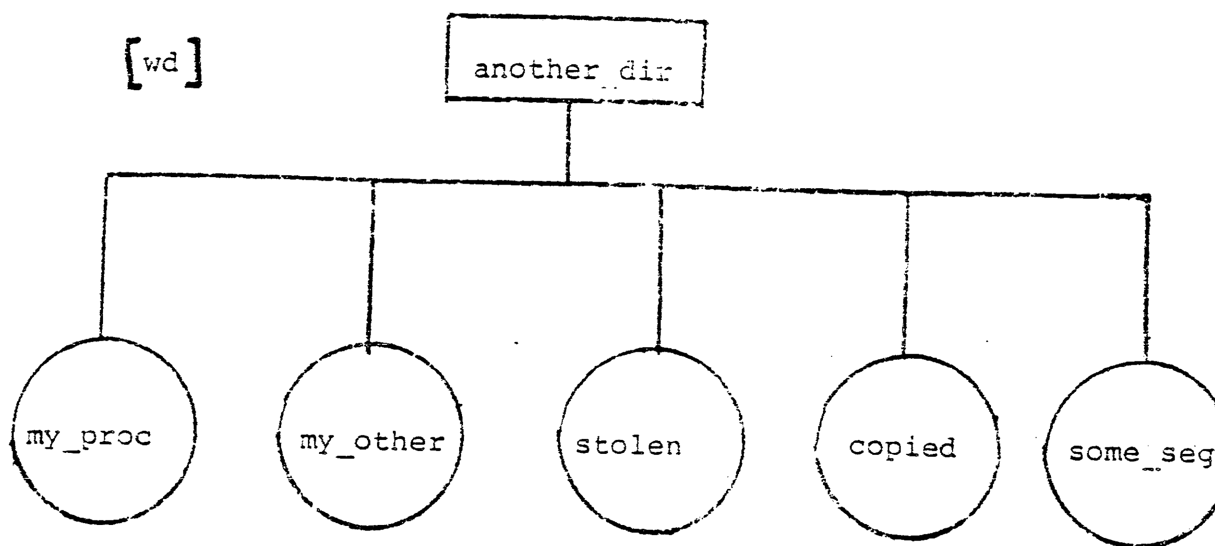

pr
401
print
401
qedx
370
qx
370

- A HUNCH: referencing_dir
  - THE SECOND RULE IN THE SEARCH RULES IS referencing_dir
  - THIS MEANS THE LINKER WILL LOOK IN THE DIRECTORY OF THE PROGRAM THAT CAUSED THE LINKAGE FAULT FOR THE SEGMENT
  - ALTHOUGH THIS IS NOTHING MORE THAN THE LIST COMMAND, IT IS NOT AN INEXPENSIVE OPERATION
  - ITS PURPOSE IS TO CONTAIN THE GLOBAL NATURE OF THE DYNAMIC LINKER IN ITS SEARCH FOR THE SEGMENT
  - THIS TENDS TO ISOLATE ALL THE PROGRAMS IN A DIRECTORY INTO A SUBSYSTEM
  - THE DANGER OF THIS WILL BE STATED LATER


THE REFERENCING DIR
some_dir
a_prog
prog_1
this_one
that_one
THE
FAULTING
PROGRAM

THE LINKER - PHASE II

- ONE MORE TRY: working_dir
  - THE LINKER FIGURES THAT THE USER HAS THE PROGRAM AND WILL LIST THE CONTENTS OF THE WORKING DIRECTORY
  - THIS ALSO TENDS TO CONTAIN THE WORKING DIRECTORY INTO A SUBSYSTEM OF SORTS
  - IT ALSO LETS THE USER ACCUMULATE A LIBRARY OF INTERWOVEN PROGRAMS

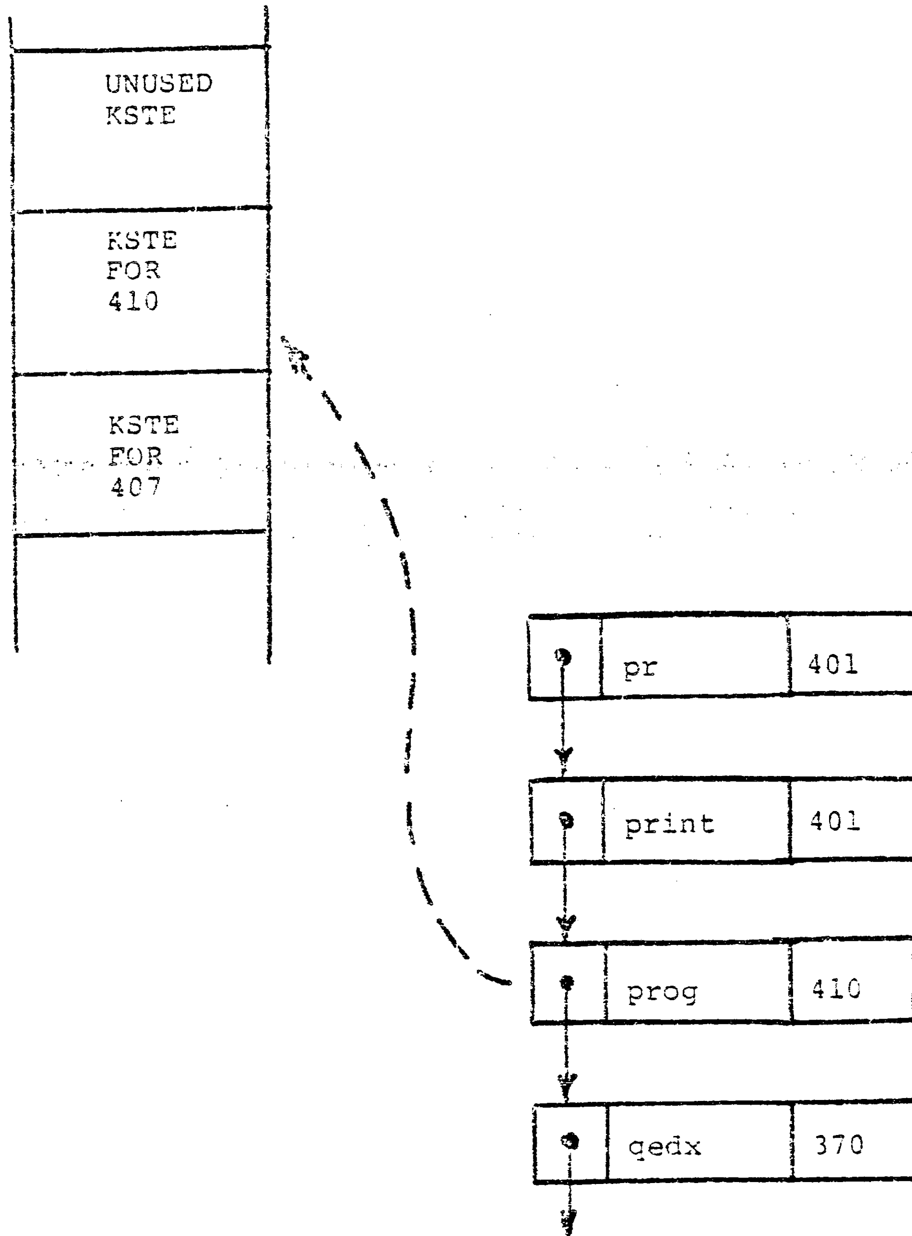

[wd]
another_dir
my_proc
my_other
stolen
copied
some_seg

THE LINKER - PHASE II

- SYSTEM LIBRARIES
  - HERE BEGINS THE BIG SEARCH
  - AGAIN, A LIST FOR EACH DIRECTORY UNTIL THE LINKER FINDS THE SEGMENT IN QUESTION
  - THIS CAN EASILY BECOME THE MOST TIME CONSUMING JOB OF THE DYNAMIC LINKER

- IF THE SEGMENT WAS NOT FOUND IN EITHER THE RNT OR THE DIRECTORIES, THEN THE "linkage_error" CONDITION IS SIGNALLED

- IF THE SEGMENT WAS NOT FOUND IN THE RNT, BUT WAS FOUND IN A DIRECTORY, ADD IT TO THE kst AND RNT
  - IF NOT IN THE kst, THEN THE PROGRAM IS NOT IN THE ADDRESS SPACE OF THE PROCESS AND CAN'T BE USED
  - AFTER ADDING THE PROGRAM INTO THE kst THE LINKER ALSO PLACES THE SEGMENT NAME INTO THE RNT
  - THE SEGMENT NAME WE OBTAINED IN PHASE ONE IS NOW A REFERENCE NAME
  - SUBSEQUENT SEARCHES FOR THIS NAME BY FUTURE PROGRAMS IS THIS PROCESS WILL FIND A MATCH IN THE RNT, AND HAVE AN INEXPENSIVE LINKAGE FAULT


UNUSED
KSTE
KSTE
FOR
410
KSTE
FOR
407
pr
401
print
401
prog
410
qedx
370

- OBTAINING THE OFFSET VALUE

  - LOCATE THE DEFINITION SECTION OF THE JUST FOUND SEGMENT

    - OBTAIN THE BIT COUNT FROM THE DIRECTORY

    - DIVIDE IT BY 36

    - SUBTRACT 1

    - THIS IS THE LOCATION OF THE OBJECT MAP POINTER; USE IT TO LOCATE THE OBJECT MAP

    - WITHIN THE OBJECT MAP FIND THE ADDRESS OF THE DEFINITION SECTION

(This page intentionally left blank)

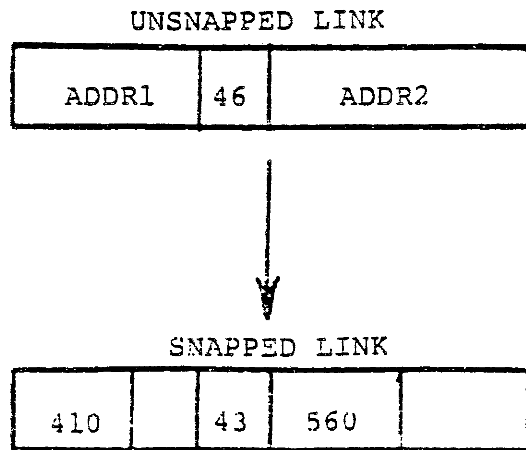
THE LINKER - PHASE II

- WALK THROUGH THE DEFINITION SECTION
  - THIS IS ANOTHER LINKED LIST
  - LOOK FOR A CHARACTER ON THE CHAIN THAT MATCHES THE SECOND NAME EXTRACTED FROM PHASE ONE.
  - ASSOCIATED WITH THIS NAME IS A BINARY OFFSET THAT THE HARDWARE CAN USE
- THE LINKER NOW HAS ALL THE INFORMATION NECESSARY TO SNAP THE LINK

SNAPPING THE LINK

- BACK IN THE FAULTING OBJECT SEGMENT WAS AN UNSNAPPED LINK
- THE LINKER WILL OVERWRITE THIS INFORMATION WITH THE NEWLY FOUND SEGMENT NUMBER AND OFFSET
- THE LINKER WILL PLACE BINARY 100011 IN THE LAST SIX BITS OF THE FIRST WORD OF THE LINK
- THE LINK IS NOW A STANDARD POINTER - IT IS SNAPPED
- ANY FURTHER REFERENCES THROUGH THIS LINK WILL NOT RESULT IN A FAULT

I THIS SNAPPING DOES NOT EFFECT ANY OTHER LINK WITHIN THE LINKAGE SECTION.



Not To Be Reproduced

THE LINKER - PHASE II

- LOADING THE OBJECT SECTIONS

  - THE NEWLY REFERENCED SEGMENT MAY HAVE A LINKAGE AND STATIC SECTION OF ITS OWN THAT MUST BE LOADED INTO MEMORYY

  - THE DYNAMIC LINKER PERFORMS THE COPYING OF THE STATIC AND LINKAGE TEMPLATES INTO THE PROPER AREAS

- IF THE LOT WORD CORRESPONDING TO THE SEGMENT IS EMPTY (I.E. LOT (SEGMENT_NUMBER) =0) THEN THE LINKER WILL COPY IT

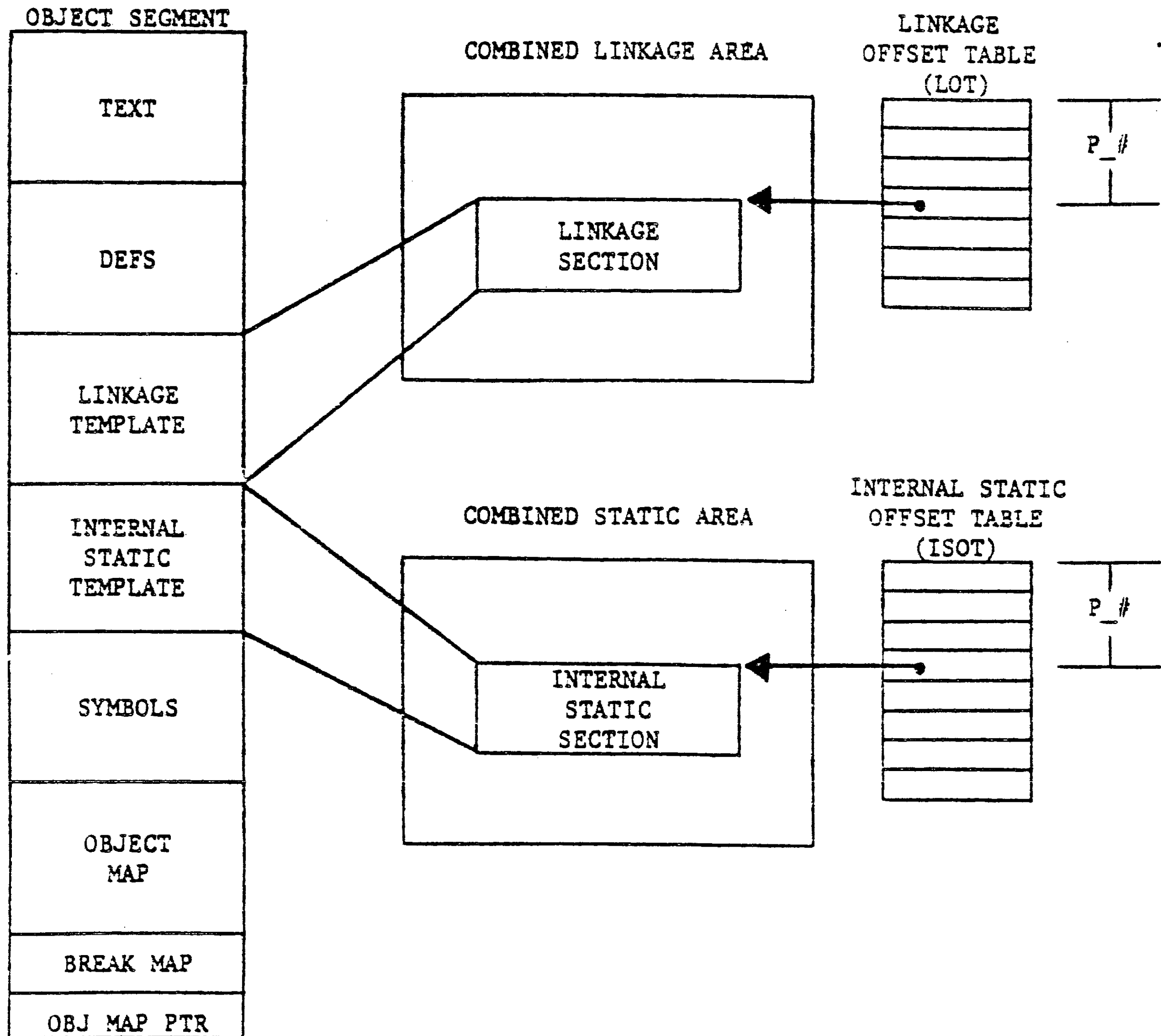
  - GET SOME ROOM IN THE COMBINED LINKAGE AREA FOR THE LINKAGE SECTION

  - COPY THE LINKAGE TEMPLATE FROM THE OBJECT INTO THE SPOT IN THE COMBINED LINKAGE AREA

  - PLACE THE ADDRESS OF THE LINKAGE SECTION INTO THE LOT ENTRY

- PERFORM THE SAME WITH THE STATIC SECTION, USING THE ISOT AND COMBINED STATIC AREA

  - NOTE: THE STATIC MAY BE COMBINED WITH THE LINKAGE INFORMATION, IN WHICH CASE THE STATIC WAS LOADED WITH THE LINKAGE SECTION


OBJECT SEGMENT
TEXT
DEFS
LINKAGE TEMPLATE
INTERNAL STATIC TEMPLATE
SYMBOLS
OBJECT MAP
BREAK MAP
OBJ MAP PTR
COMBINED LINKAGE AREA
LINKAGE SECTION
LINKAGE OFFSET TABLE (LOT)
P_#
COMBINED STATIC AREA
INTERNAL STATIC SECTION
INTERNAL STATIC OFFSET TABLE (ISOT)
P_#

THE LINKER - PHASE II

- INSTRUCTION RETRY

  - RETURN TO THE SCENE OF THE CRIME

    - WHEN link_snap IS FINISHED, IT RETURNS TO fim

    - fim THEN CAUSES THE INSTRUCTION THAT GENERATED THE LINKAGE FAULT TO BE REEXECUTED

    - WITH THE LINK NOW SNAPPED, A FAULT WILL NOT OCCUR AND THE INSTRUCTION WILL FIND THE THING IT WAS LOOKING FOR

(This page intentionally left blank)

BY-PRODUCTS OF DYNAMIC LINKING

- INITIATION

  - EVERY SEGMENT THAT A PROCESS WANTS TO USE MUST BE REGISTERED WITH BOTH THE dseg AND kst

    descriptor seg

  - IF A SEGMENT IS NOT REGISTERED - KNOWN - TO A PROCESS AND IT IS THE OBJECT OF A LINKAGE FAULT, THEN THE DYNAMIC LINKER WILL MAKE IT KNOWN

    - THIS INVOLVES GOING TO THE kst, FINDING OUT THE NEXT FREE NUMBER TO USE AND ASSIGNING IT TO THE NEW SEGMENT

    - BECAUSE THIS IS PRETTY MUCH AN UNPREDICTABLE OPERATION AS FAR AS AVAILABLE NUMBERS ARE CONCERNED, MULTICS DOES NOT GUARANTEE THE CONSISTENCY OF SEGMENT NUMBERS ACROSS PROCESS BOUNDARIES

- HIDDEN DANGERS

  - THE SEARCH FOR A SEGMENT TO FULFILL THE LINKAGE FAULT CAN CREATE DANGERS FOR PROGRAMMERS WHO ARE NOT AWARE OF THE NATURE OF THE SEARCH

  - CONSIDER THE FOLLOWING SCENARIO

    - A PROGRAMMER HAS WRITTEN A SET OF PROGRAMS

      THE FIRST PROGRAM IS CALLED driver

      IT CALLS calculate_total

      IT THEN CALLS ioa_ TO PRINT THE TOTAL OUT

      LATER (AND ALTHOUGH UNLIKELY, POSSIBLE) driver CALLS A PROGRAMMER PROVIDED PROGRAM NAMED formline_

BY-PRODUCTS OF DYNAMIC LINKING

- THERE IS A SIGNIFICANT CHANCE THAT THE PROGRAMMER SUPPLIED formline_ WILL NOT EXECUTE

- THE MULTICS SYSTEM SUBROUTINE, ioa_, ALSO CALLS A PROGRAM NAMED formline_

  IF ioa_ TOOK A LINKAGE FAULT WHILE CALLING formline_, IT WOULD HAVE FOUND THE SYSTEM'S COPY USING THE referencing_dir RULE, PLACED S NAME IN THE RNT, AND driver WOULD HAVE FOUND IT WHEN IT CALLED formline_.

- NOTE, THEN THAT IT IS POTENTIALLY DANGEROUS TO CALL PROGRAMS OUTSIDE THE DIRECTORY OF EXECUTION IF THE NAMES OF SEGMENTS CAN BE DUPLICATED ELSEWHERE

- BINDING ALSO HELPS

BINDING

- BINDING ITSELF IS A LINKING PROCESS, BUT ITS EFFECTS CAN BE FELT SYSTEM WIDE

- THIS EXPLANATION WILL CONCERN ITSELF WITH ONLY THE LINKING ASPECTS OF BINDING

- ONE OF THE ADVANTAGES OF DYNAMIC LINKING IS THAT UNUSED EXTERNAL REFERENCES WERE NOT LINKED, SAVING TIME

- IF A SET OF PROGRAMS MAKE MANY CALLS TO EACH OTHER AND IT IS ALMOST UNAVOIDABLE THAT ALL LINKS WILL BE SNAPPED IN THE COURSE OF THEIR EXECUTION, THEN PRELINKING WILL BE CHEAPER

BY-PRODUCTS OF DYNAMIC LINKING

- I EACH LINK WILL BE SNAPPED ONCE IN ITS LIFE, AS OPPOSED TO MANY TIMES WITH DYNAMIC LINKING

- I THE PROGRAMMER MUST GIVE UP THE ABILITY TO MAKE CHANGES TO OBJECT EASILY; THEREFORE BINDING SHOULD BE DONE ONLY AFTER THE PROGRAMS ARE FULLY DEBUGGED

I THE BINDER'S TASK

- I BREAK APART ALL THE SECTIONS OF ALL THE PROGRAMS TO BE BOUND

- I GROUP ALL LIKE SECTIONS TOGETHER (TEXT WITH TEXT, ETC)

- I COMBINE ALL THE LINKAGE SECTIONS TOGETHER, AND ELIMINATE ALL LINK DUPLICATIONS

- I ELIMINATE SOME ENTRYPOINTS INTO THE PROGRAMS, TRIMMING DOWN THE DEFINITION SECTION

- I GENERATE ONE OBJECT MAP AND OBJECT MAP POINTER

I A BOUND SEGMENT MAY ACTUALLY HAVE LINKS LEFT OVER THAT WERE NOT RESOLVED AT BINDING TIME; THEY WILL BE HANDLED BY THE DYNAMIC LINKER WHEN NEEDED

TOPIC VII

The Multics Programming Environment 7-1
Destruction of the Programming Environment. . 7-1
Error Recovery Techniques 7-8
The Multics Programming Environment 7-1
Destruction of the Programming Environment. . 7-1
Error Recovery Techniques 7-8

OBJECTIVES:

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

1. Discuss some of the ways in which the Multics process environment can be disrupted.

2. Locate and correct user programming errors which cause a process to terminate abnormally.

3. Apply preventive techniques during program development to minimize the number of potentially dangerous programming errors.

DESTRUCTION OF THE PROGRAMMING ENVIRONMENT

- SOURCE SEGMENT
    - WHEN INITIATED BY THE COMPILER, NOT GIVEN A REFERENCE NAME
    - USUALLY NOT MADE KNOWN EXCEPT BY COMPILER
  - SEGMENT USUALLY NOT KNOWN
  - DESTRUCTION UNLIKELY

- OBJECT SEGMENT
  - SEGMENT READ-EXECUTE ONLY (EXCEPT WHEN DEBUGGER IS SETTING BREAKPOINTS)
  - DESTRUCTION UNLIKELY BECAUSE OF HARDWARE PROTECTION

DESTRUCTION OF THE PROGRAMMING ENVIRONMENT

- stack_n SEGMENT
  - IN [pd]
  - READ-WRITE
  - INCLUDES
    - PROGRAM ACTIVATION HISTORY (STACK)
    - AUTOMATIC VARIABLES
    - STACK HEADER INFORMATION
      - INCLUDES INITIAL LOT & ISOT ALLOCATIONS
  - DESTRUCTION POSSIBLE THROUGH MISUSE OF AUTOMATIC VARIABLES OR BUILT-IN FUNCTIONS
    - SUBSCRIPTRANGE
    - STRINGRANGE
    - USE OF UNINITIALIZED POINTER TO BASED VARIABLE
  - SYMPTOMS
    - IF STACK HEADER OVERWRITTEN, FATAL PROCESS ERROR USUALLY OCCURS

DESTRUCTION OF THE PROGRAMMING ENVIRONMENT

- STRINGRANGE OFTEN RESULTS IN STORAGE CONDITION (out_of_bounds ON USER'S STACK)

- SUBSCRIPTRANGE CAUSES AUTOMATIC DATA AND/OR PROGRAM ACTIVATION INFORMATION TO BE OVERWRITTEN, LEADING TO IMPROPER RESULTS AND PROGRAM OPERATION

DESTRUCTION OF THE PROGRAMMING ENVIRONMENT

- [unique].area.linker SEGMENT
  - IN [pd]
  - READ-WRITE AREA
  - INCLUDES
    - COMBINED LINKAGE AREA
      - LINKAGE SECTIONS
      - LOT
      - ISOT
      - RNT
    - COMBINED STATIC AREA
      - INTERNAL STATIC SECTIONS (VARIABLES)
    - USER FREE AREA
      - EXTERNAL STATIC AND COMMON VARIABLES - PER PROCESS
      - EXTERNAL VARIABLE CONTROL INFORMATION
      - CONTROLLED VARIABLES
      - BASED VARIABLES - NO AREA, IN AN I/O BUFFER
      - COBOL VARIABLES
    - ASSIGNED LINKAGE AREA
      - BASED STORAGE-ALLOCATED THROUGH hcs_$assign_linkage

DESTRUCTION OF THE PROGRAMMING ENVIRONMENT

- [unique].area.linker SEGMENT (continued)

  - DESTRUCTION POSSIBLE THROUGH

    - SUBSCRIPTRANGE

    - STRINGRANGE

    - USE OF UNINITIALIZED POINTERS

    - MISUSE OF AREA

      - ~~FREEING SAME BASED VARIABLE TWICE~~

  - SYMPTOMS

    - IF LINKAGE SECTIONS OVERWRITTEN, IMPROPER PROGRAM OPERATION

    - IF LOT OVERWRITTEN, IMPROPER OPERATION OF ALL PROGRAMS

    - IF ISOT OVERWRITTEN, IMPROPER INTERNAL STATIC DATA; SUBSEQUENT DESTRUCTION OF OTHER DATA

    - IF RNT OVERWRITTEN, UNABLE TO FIND PREVIOUSLY-REFERENCED PROGRAMS

    - IF VARIABLES (OF ANY STORAGE CLASS) ARE OVERWRITTEN, IMPROPER VARIABLE VALUES

    - IF EXTERNAL VARIABLE CONTROL INFORMATION OVERWRITTEN, IMPROPER COMMUNICATION OF SHARED VARIABLES BETWEEN PROGRAMS; IMPROPER DATA VALUES

DESTRUCTION OF THE PROGRAMMING ENVIRONMENT

- IF AREA CONTROL INFORMATION OVERWRITTEN, bad_area_format CONDITION

DESTRUCTION OF THE PROGRAMMING ENVIRONMENT

DIRECTORIES, dseg, kst

- NO DIRECT ACCESS TO USER FROM USER RING (RING 4)

- DESTRUCTION UNLIKELY, SEGMENTS PROTECTED BY HARDWARE

ERROR RECOVERY TECHNIQUES

- MOST ERRORS WHICH DESTROY THE PROGRAMMING ENVIRONMENT

  - ARE CAUSED BY IMPROPER SUBSCRIPTS, BAD SUBSTRING OPERANDS, OR POINTERS USED IMPROPERLY

  - RECOVERY FROM SUBSCRIPTRANGE AND STRINGRANGE

    - RECOMPILE PROGRAMS CAUSING THESE ERRORS AND ENABLE CHECKING FOR THESE CONDITIONS

      - PL/1: INSERT A LINE CONTAINING

        (size, stringsize, stringrange, subscriptrange):

        AT THE BEGINNING OF EACH SOURCE SEGMENT, AND RECOMPILE WITH -table OPTION

      - COBOL: USE -runtime_check AND -table CONTROL ARGUMENTS IN cobol COMMAND

      - FORTRAN: USE -subscriptrange AND -table CONTROL ARGUMENTS IN fortran COMMAND

    - RUN PROGRAMS

    - IF CONDITIONS ARE SIGNALLED, USE probe TO FIND CAUSE

    - FIX PROBLEMS, AND RECOMPILE AS ABOVE UNTIL NO MORE CONDITIONS ARE SIGNALLED

    - IF NO MORE CONDITIONS ARE SIGNALLED, BUT PROGRAMMING ENVIRONMENT ERRORS STILL PERSIST

      - RECOMPILE WITHOUT THE CONDITION CHECKING, BUT WITH -table CONTROL ARGUMENT

      - PROCEED AS GIVEN BELOW UNDER "FURTHER ERROR RECOVERY"

ERROR RECOVERY TECHNIQUES

- IF ALL ERRORS CORRECTED, RECOMPILE WITHOUT CONDITION CHECKING OR -table

- FURTHER ERROR RECOVERY

  - bad_area_format CONDITION IN [unique].area.linker SEGMENT

    - CAUSED BY OVERWRITING AREA CONTROL INFORMATION

      - STORED AT BEGINNING OF AREA

      - STORED BETWEEN SPACE ALLOCATIONS

    - RECOVERY TECHNIQUES (ASSUMES STRINGRANGE AND SUBSCRIPTRANGE TESTS HAVE ALREADY BEEN DONE)

      - USE area_status COMMAND TO FIND FAULTY LOCATION IN AREA

      - USE dump_segment COMMAND TO PRINT AREA AROUND THAT LOCATION; RECOGNIZABLE DATA MAY LEAD TO THE CAUSE

      - USE create_area AND set_user_storage COMMANDS TO SEPARATE USER FREE AREA FROM OTHER GROUPED AREAS

      - IF ERROR OCCURS NOW IN USER-SPECIFIED AREA SEGMENT, THEN PROBLEM IS IN A USER PROGRAM (NO SYSTEM PROGRAMS EXCEPT EXTERNAL VARIABLE MANAGER USE THIS AREA)

      - USE probe TO EXAMINE ALL POINTER-QUALIFIED REFERENCES TO BE SURE POINTER IS SET PROPERLY

      - AFTER BASED VARIABLES HAVE BEEN FREED, NULL THEIR POINTER TO PREVENT SUBSEQUENT REFERENCE TO FREED SPACE

- FURTHER ERROR RECOVERY (continued)

  - FATAL PROCESS ERRORS (REPRODUCIBLE)

    - CAUSED BY OVERWRITING

      - STACK HEADER

      - LINKAGE OR INTERNAL STATIC SECTIONS OF CRITICAL PROGRAMS (iox_, listen_, command_processor_, print_ready_msg_, etc.)

    - RECOVERY TECHNIQUES

      - ATTEMPT TO ISOLATE POINT OF PROCESS FAILURE TO A SINGLE PROGRAM STATEMENT

      - USE probe TO

        - SET BREAKS AT KEY POINTS IN THE EXECUTION OF THE PROGRAM

        - CONTINUE EXECUTION AS EACH BREAK IS REACHED UNTIL FATAL ERROR OCCURS

        - WHEN FATAL ERROR OCCURS, POINT OF FAILURE LIES AFTER LAST BREAKPOINT WHICH WAS REACHED

        - SET BREAKS AFTER THIS POINT TO FURTHER ISOLATE POINT OF FAILURE TO A SINGLE STATEMENT

      - FAILING STATEMENT MAY BE

        - CAUSE OF ERROR

        - USING INCORRECT DATA AS RESULT OF A PREVIOUS ERROR

      - USE probe TO TRACK ORIGINAL CAUSE OF ERROR

      - USE -watch CONTROL ARGUMENT OF trace COMMAND TO ISOLATE THE SUBROUTINE WHICH IS DAMAGING A PARTICULAR DATA ITEM

- FURTHER ERROR RECOVERY (continued)

  - FATAL PROCESS ERRORS (INTERMITTENT)

    - CAN BE CAUSED BY

      - UNINITIALIZED DATA VALUES
      - ANOTHER PROGRAM DESTROYING YOUR PROGRAM'S DATA

    - RECOVERY TECHNIQUES

      - IN A NEW PROCESS, RUN JUST THE FAILING PROGRAM
      - IF PROGRAM OPERATES CORRECTLY, ANOTHER PROGRAM MAY BE SOURCE OF ERROR
      - IF PROGRAM FAILS (ESPECIALLY FAILS INTERMITTENTLY OR IN DIFFERENT WAYS), USE probe TO LOOK FOR UNINITIALIZED VARIABLES

TOPIC VIII

Other Useful Debugging Tools. 8-1
list_external_variables 8-1
list_external_variables 8-1
reset_external_variables. 8-2
reset_external_variables. 8-2
delete_external_variables 8-3
delete_external_variables 8-3
print_bind_map. 8-4
print_bind_map. 8-4
print_link_info 8-5
print_link_info, pli. 8-5
resolve_linkage_error 8-7
reslve_linkage_error, rle 8-7
trace_stack 8-8
trace_stack, ts 8-8

OBJECTIVES:

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

1. Manipulate external variables with the following commands:

   list_external_variables (lev)

   reset_external_variables (rev)

   delete_external_variables (dev)

2. Find and correct problems related to linking with the following commands:

   print_bind_map (pbm)

   print_link_info (pli)

   resolve_linkage_error (rle)

3. Use the trace_stack (ts) command in conjunction with trace and probe to determine the state of the process when an error occurs.

Name: list_external_variables

The list_external_variables command prints information about variables managed by the system for the user, including FORTRAN common and PL/I external static variables whose names do not contain dollar signs. The default information is the location and size of each specified variable.

Usage

list_external_variables names {-control_args}

where:

1. names
   are names of external variables, separated by spaces.

2. control_args
   can be chosen from the following:

   -unlabeled_common, -uc
      is the name for unlabeled (or blank) common.

   -long, -lg
      prints how and when the variables were allocated.

   -all, -a
      prints information for each variable the system is managing.

   -no_header, -nhe
      suppresses the header.

Name: reset_external_variables

The reset_external_variables command reinitializes system-managed variables to the values they had when they were allocated.

Usage

reset_external_variables names {-control_arg}

where:

1. names
   are the names of the external variables, separated by spaces, to be reinitialized.

2. control_arg
   is -unlabeled_common (or -uc) to indicate unlabeled (or block) common.

   Note

   A variable cannot be reset if the segment containing the initialization information is terminated after the variable is allocated.

Name: delete_external_variables

The delete_external_variables command deletes from the user's name space specified variables managed by the system for the user. All links to those variables are unsnapped and their storage is freed.

Usage

delete_external_variables names {-control_arg}

where:

1. names
   are the names of the external variables, separated by spaces, to be deleted.

2. control_arg
   Is -unlabeled_common (or -uc) to indicate unlabeled (or blank) common.

Name: print_bind_map

The print_bind_map command displays all or part of the bind map of an object segment generated by version number 4 or subsequent versions of the binder.

Usage

```
print_bind_map path {components} {-control_args}
```

where:

1. path
   is the pathname of a bound object segment.

2. components
   are the optional names of one or more components of this bound object and/or the bindfile name. Only the lines corresponding to these components are displayed. A component name must contain one or more nonnumeric characters. If it is purely numerical, it is assumed to be an octal offset within the bound segment and the lines corresponding to the component residing at that offset are displayed. A numerical component name can be specified by preceding it with the -name control argument (see below). If no component names are specified, the entire bind map is displayed.

3. control_args
   may be chosen from the following list:

   -long, -lg
   prints the components' relocation values (also printed in the default brief mode), compilation times, and source languages.

   -name STR, -nm STR
   is used to indicate that STR is really a component name, even though it appears to be an octal offset.

   -no_header, -nhe
   omits all headers, printing only lines concerning the components themselves.

Name: print_link_info, pli

The print_link_info command prints selected items of information for the specified object segments.

Usage

```
print_link_info paths {-control_args}
```

where:

1. paths
   are the pathnames of object segments.

2. control_args
   can be chosen from the following list. (See "Note" below.)

   -length, -ln
   print only the lengths of the sections in pathi.

   -entry, -et
   print only a listing of the pathi external definitions, giving their symbolic names and their relative addresses within the segment.

   -link, -lk
   print only an alphabetically sorted listing of all the external symbols referenced by pathi.

   -long
   prints more information when the header is printed. Additional information includes a listing of source programs used to generate the object segment, the contents of the "comment" field of the symbol header (often containing compiler options), and any unusual values in the symbol header.

   -header, -he
   prints the header (The header is not printed by default, if the -length, -entry, or -link control argument is specified.)

   -no_header
   suppresses printing of the header.

Note

Control arguments can appear anywhere on the command line and apply to all pathnames.

Example

```
! print_link_info program -long -length

                  program 07/30/76  1554.2 edt Fri


Object Segment >udd>Work>Wilson>program
Created on 07/30/76  0010.1 edt Fri
by Wilson.Work.a
using Experimental PL/I Compiler of Thursday, July 26, 1976 at 21:38


Translator:         PL/I
Comment:            map table optimize
Source:
 07/30/76  0010.1 edt Fri  >user_dir_dir>work>Wilson>s>s>program.pl1
 12/15/75  1338.1 edt Mon  >library_dir_dir>include>linkdcl.incl.pl1
 06/30/75  1657.7 edt Mon  >library_dir_dir>include>object_info.incl.pl1
 10/06/72  1206.8 edt Fri  >library_dir_dir>include>source_map.incl.pl1
 05/18/72  1512.4 edt Thu  >library_dir_dir>include>symbol_block.incl.pl1
 01/17/73  1551.4 edt Wed  >library_dir_dir>include>pl1_symbol_block.incl.pl1
Attributes:         relocatable,procedure,standard

        Object    Text    Defs    Link     Symb   Static
Start        0       0    3450    3620     3656     3630
Length   11110    3450     150      36     5215        0


<ready>
```

Also printed is:

Severity, if it is nonzero.
Entrybound, if it is nonzero.
Text Boundary, if it is not 2.
Static Boundary, if it is not 2.

Name : resolve_linkage_error, rle

The resolve_linkage_error command is invoked to satisfy the linkage fault after a process encounters a linkage error. The program locates the virtual entry specified as an argument and patches the linkage information of the process so that when the start command is issued the process continues as if the original linkage fault had located the specified virtual entry.

Usage

```
resolve_linkage_error  virtual_entry
```

where virtual_entry is a virtual entry specifier.

Notes

For an explanation of virtual entries, see the description of the cv_entry_ subroutine.

Examples

```
!     myprog
      Error:  Linkage error by >udd>m>vv>myprog|123
      referencing subroutine$entry
      Segment not found.
      r 1234 2.834 123.673 980 level 2, 26

!     rle mysub$mysub_entry
      r 1234 0.802 23.441 75 level 2, 26

!     start
      ... myprog is running
```

Name: trace_stack, ts

The trace_stack command prints a detailed explanation of the current process stack history in reverse order (most recent frame first). For each stack frame, all available information about the procedure that established the frame (including, if possible, the source statement last executed), the arguments to that (the owning) procedure, and the condition handlers established in the frame are printed. For a description of stack frames, see "Multics Stack Segments" in Section!IV of the MPM Subsystem Writers' Guide.

The trace_stack command is most useful after a fault or other error condition. If the command is invoked after such an error, the machine registers at the time of the fault are also printed, as well as an explanation of the fault. The source line in which it occurred can be given if the object segment is compiled with the -table option.

Usage

```
trace_stack {-control_args}
```

where control_args can be selected from the following:

-brief, -bf
: suppresses listing of arguments and handlers. This control argument cannot be specified if -long is also specified as a control argument.

-long, -lg
: prints octal dump of each stack frame.

-depth N, -dh N
: dumps only N frames.

Output Format

When trace_stack is invoked, it first searches backward through the stack for a stack frame containing saved machine conditions as the result of a signalled condition. If such a frame is found, tracing proceeds backward from that point; otherwise, a comment is printed and tracing begins with the stack frame preceding trace_stack.

If a machine-conditions frame is found, trace_stack repeats the system error message describing the fault. Unless the -brief control argument is specified, trace_stack also prints the source line and faulting

instruction and a listing of the machine registers at the time the error occurred.

The command then performs a backward trace of the stack, for N frames if the -depth N argument was specified, or else until the beginning of the stack is reached.

For each stack frame, trace_stack prints the offset of the frame, the condition name if an error occurred in the frame, and the identification of the procedure that established the frame. If the procedure is a component of a bound segment, the bound segment name and the offset of the procedure within the bound segment are also printed.

The trace_stack command then attempts to locate and print the source line associated with the last instruction executed in the procedure that owns the frame (that is, either a call forward or a line that encountered an error). The source line can be printed only if the procedure has a symbol table (that is, if it was compiled with the -table option) and if the source for the procedure is available in the user's working directory. If the source line cannot be printed, trace_stack prints a comment explaining why.

Next, trace_stack prints the machine instruction last executed by the procedure that owns the current frame. If the machine instruction is a call to a PL/I operator, trace_stack also prints the name of the operator. If the instruction is a procedure call, trace_stack suppresses the octal printout of the machine instruction and prints the name of the procedure being called.

Unless the -brief control argument is specified, trace_stack lists the arguments supplied to the procedure that owns the current frame and also lists any enabled condition, default, and clean-up handlers established in the frame.

If the -long control argument is specified, trace_stack then prints an octal dump of the stack frame, with eight words per line.

Example

After a fault that reenters the user environment and reaches command level, the user invokes the trace_stack command.

For example, after quitting out of the list command, the following process history might appear:

```
        !  list

Segments=8, Records=3

rew  0  mailbox
r w
QUIT

!  trace_stack
   quit in ipc_$block|156
   (>system_library_1>bound_command_loop_|156)
     No symbol table for ipc_
      156   400010116100     cmpq    pr4|10

Machine registers at time of fault

pr0 (ap)   263|4656            pl1_operators_$operator_table|162
                                 (external symbol in separate nonstandard
                                  text section)
pr1 (ab)   103|264             scs|264
pr2 (bp)   14|12200            as_linkage|12200
pr3 (bb)   113|0               tc_data|0
pr4 (lp)   253|2250            !BBBJGjFkPBWcNZ.area.linker|2250
                                 (internal static|0 for ipc_)
pr5 (lb)   244|3614            stack_4|3614
pr6 (sp)   244|3500            stack_4|3500   ( -> "kcpMbLH +0000000")
pr7 (sb)   244|0               stack_4|0

x0      73   x1       0   x2       0   x3 600000
x4       0   x5      32   x6    3033   x7      4
a 000000000000  q 000000000004  e 0
Timer reg - 1746005, Ring alarm reg - 0

SCU Data:

  4030    400270250011 000000000021 400270000000 000000672000
          000156000200 000154000700 002250370000 600044370120

Connect Fault (21)
At: 270|156 ipc_|156  (bound_command_loop_|156)
On: cpu a (#0)
Indicators:  ^bar
```

```
APU Status:  xsf, sd-on, pt-on, fabs
CU Status:   rfi, its, fif
Instructions:
  4036    002250 3700 00      epp4      2250
  4037   6 00044 3701 20      epp4      pr6|44,*

Time stored: 08/02/77  1635.5 edt Tue (104541674361226602)
Ring:           4

Backward trace of stack from 244|3500


3500 quit ipc_$block|156 (bound_command_loop_|156)
  No symbol table for ipc_
   156   400010116100      cmpq    pr4|10
          ARG  1: 253|5704 !BBBJGjFkPBWcNZ.area.linker|5704
          ARG  2: 244|3152 stack_4|3152
          ARG  3: 0


2720       tty_$tty_get_line|2442 (bound_iox_|11546)
  No symbol table for tty_
  call_ext_out to ipc_$block
          ARG  1: 253|4320 !BBBJGjFkPBWcNZ.area.linker|4320
                             (internal static|154 for find_iocb)
          ARG  2: 244|2660 stack_4|2660 ( -> "fc stuff")
          ARG  3: 128
          ARG  4: 0
          ARG  5: 0


2400       listen_$listen_|461 (bound_command_loop_|1325)
  No symbol table for listen_
  call_ext_out to iox_$get_line
          ARG  1: ""
          on "cleanup"  call listen_|256 (bound_command_loop_|1122)


2100       process_overseer_$process_overseer_|473 (bound_command_loop_|214
  No symbol table for process_overseer_
  call_ext_out_desc to listen_$listen_
          Argument list header invalid.
          on "any_other"
            call standard_default_handler_$standard_default_handler_3
            (external symbol in separate nonstandard text section)


2000       user_init_admin_$user_init_admin_|36 (bound_command_loop_|21676)
  No symbol table for user_init_admin_
 21676   700036670120      tsp4     pr7|36,* alm_call
          No arguments.

End of trace.
```

trace stack, ts

r 1635 1.756 40.790 207 level 2, 9

Name: area_status

The area_status command is used to display certain information about an area.

Usage

area_status area_name {-control_args}

where:

1. area_name
   is a pathname specifying the segment containing the area to be looked at.

2. control_args
   can be chosen from the following:

   -trace
      displays a trace of all free and used blocks in the area.

   -offset N, -ofs N
      specifies that the area begins at offset N (octal) in the given segment.

   -long, -lg
      dumps the contents of each block in both octal and ASCII format.

Note

If the area has internal format errors, these are reported. The command does not report anything about (old) buddy system areas except that the area is in an obsolete format.

Name: cancel_cobol_program, ccp

The cancel_cobol_program command causes one or more programs in the current COBOL run unit to be cancelled. Cancelling ensures that the next time the program is invoked within the run unit, its data is in its initial state. Any files that have been opened by the program and are still open are closed and the COBOL data segment is truncated. Refer to the run_cobol command for information concerning the run unit and the COBOL runtime environment.

Usage

```
cancel_cobol_program names {-control_arg}
```

where:

1. names
   are the reference names of COBOL programs that are active in the current run unit. If the name specified in the PROG-ID statement of the program is different from its associated namei argument, namei must be in the form refname$PROG-ID.

2. control_arg
   may be -retain_data or -retd to leave the data segment associated with the program intact for debugging purposes. (See "Notes" below.)

Notes

The results of the cancel_cobol_program command and the execution of the CANCEL statement from within a COBOL program are similar. The only difference is that if a namei argument is not actually a component of the current run unit, an error message is issued and no action is taken; for the CANCEL statement, no warning is given in such a case.

To preserve program data for debugging purposes, the -retain_data control argument should be used. The data associated with the cancelled program is in its last used state; it is not restored to its initial state until the next time the program is invoked in the run unit.

Refer to the following related commands:

cancel cobol program, ccp

display_cobol_run_unit, dcr
stop_cobol_run, scr
run_cobol, rc

Name: create_area

The create_area command creates an area and initializes it with user-specified area management control information.

Usage

create_area virtual_ptr {-control_args}

where:

1. virtual_ptr
   is a virtual pointer to the area to be created. The syntax of virtual pointers is described in the cv_ptr_ subroutine description. If the segment already exists, the specified portion is still initialized as an area.

2. control_args
   can be chosen from the following:

   -no_freeing
      allows the area management mechanism to use a faster allocation strategy that never frees.

   -dont_free
      is used during debugging to disable the free mechanism. This does not affect the allocation strategy.

   -zero_on_alloc
      instructs the area management mechanism to clear blocks at allocation time.

   -zero_on_free
      instructs the area management mechanism to clear blocks at free time.

   -extend
      causes the area to be extensible, i.e., span more than one segment. This feature should be used only for perprocess, temporary areas.

   -size N
      specifies the octal size, in words, of the area being created or of the first component, if extensible. If this control argument is omitted, the default size of the area is the maximum size allowable for a segment.

   -id STR
      specifies a string to be used in constructing the names of the components of extensible areas.

Name: create_data_segment, cds

The create_data_segment command translates a create_data_segment source program (CDS program) into an object segment. A listing segment is optionally created. These results are placed in the user's working directory. This command cannot be called recursively.

The source for create_data_segment programs is standard PL/I with the restriction that the program include a call to the create_data_segment_ subroutine. The create_data_segment_ subroutine creates a standard object segment from PL/I data structures passed to it as parameters. These data structures can be initialized with arbitrarily complex PL/I statements in the CDS program. (See the MPM Subroutines for a description of the create_data_segment_ subroutine.)

Usage

```
create_data_segment path {-control_arg}
```

where:

1. path
   is the pathname of a CDS segment that is to be translated into an object segment. If path does not have a cds suffix, one is asssumed. However, the cds suffix must be the last component of the name of the source segment.

2. control_arg
   can be -list (-ls) to produce a source listing of the CDS program used to generate the data segment followed by object segment information (as printed by the print_link_info command described in the MPM Subsystem Writers' Guide) about the actual object segment created.

Note

Since the create_data_segment command invokes the PL/I compiler to first compile the CDS segment, any errors that the compiler finds are reported by its standard technique. If any errors with a severity greater than 2 occur, the CDS run is aborted and an object segment is not created.

Name: cumulative_page_trace, cpt

The cumulative_page_trace command accumulates page trace data so that the total set of pages used during the invocation of a command or subsystem can be determined. The command accumulates data from one invocation of itself to the next. Output from the command is in tabular format showing all pages that have been referenced by the user's process. A trace in the format of that produced by the page_trace command can also be obtained.

The cumulative_page_trace command operates by sampling and reading the system trace array after invocation of a command and at repeated intervals. Control arguments are given to specify the detailed operation of the cumulative_page_trace command.

The command line used to invoke the cumulative_page_trace command includes the command or subsystem to be traced as well as optional control arguments.

Usage

cumulative_page_trace command_line {-control_args}

where:

1. command_line
   Is a character string to be interpreted by the command processor as a command line. If this character string contains blanks, it must be surrounded by quotes. All procedures invoked as a result of processing this command line are metered by the cumulative_page_trace command.

2. control_args
   may be chosen from the following:

   -count, -ct
   prints the accumulated results, giving the number of each page and the number of faults for each page.

   -flush
   clears primary memory before each invocation of the command line and after each interrupt. This helps the user determine the number of page faults but increases the cost.

   -interrupt N -int N
   interrupts execution every N virtual CPU milliseconds for page fault sampling.

-long, -lg
: produces output in long format, giving full pathnames.

-loop N
: calls the command to be metered N times.

-print, -pr
: prints the accumulated results, giving the number of each page referenced.

-print_linkage_faults
: prints all accumulated linkage faults and calls to the hcs_$make_ptr entry point.

-reset, -rs
: resets the table of accumulated data. If the table is not reset, data from the current use of cumulative_page_trace is added to that obtained earlier in the process.

-short, -sh
: formats output for a line length of 80.

-sleep N
: waits for N seconds after each call to the command being metered.

-timers
: includes all faults between signal and restart.

-total, -tt
: prints the total number of page faults, the total number of segment faults, and the number of pages referenced for each segment.

-trace_linkage_faults
: accumulates linkage faults information along with page and segment fault information.

-trace path
: writes the trace on the segment named path using an I/O switch named cpt.out; cumulative_page_trace attaches and detaches this switch.

Notes

At least one of three generic operations must be requested. They may all be combined and, if so, are performed in the following order: resetting the table of accumulated data, calling the command to be metered, applying the specified control arguments, and printing the results in the specified format.

The default mode of operation permits no interrupts for page fault sampling. If the command or subsystem to be metered will take more than several hundred page faults, linkage faults, or other system events that are indicated in the page trace array, it is recommended that interrupts be requested. If the user does not know a suitable value for the -interrupt control argument, the value recommended is 400 milliseconds. If this figure is too large, messages indicate that some page faults may have been missed; a smaller value can then be chosen. The cost of a smaller value is high and may cause additional side effects. If the command or subsystem to be metered includes the taking of CPUT interrupts, then the -timers control argument should be given. This control argument causes some of the page faults of the metering mechanism to be included as well.

Only one of the control arguments -print, -count, or -total may be given. Each of these control arguments produces printed output in a different format. If more than one format is desired, the command must be invoked once for each format.

Examples

The command line:

```
cpt "pl1 test" -interrupt 400 -trace trace_out
```

calls the pl1 command to compile the program named test, requesting an interrupt every 400 milliseconds to obtain page trace information. Trace information is placed in a segment named trace_out.

The command line:

```
cpt "list -pn >udd>Multics" -loop 2 -sleep 10
```

calls the list command twice, and sleeps for 10 seconds between calls.

The command line:

```
cpt -print
```

prints the accumulated results of previous metering.

Name: cv_ptr_

The cv_ptr_ subroutine converts a virtual pointer to a pointer value. A virtual pointer is a character string representation of a pointer value. The types of virtual pointers accepted are described under "Virtual Pointers" below.

Usage

```
declare cv_ptr_ entry (char(*), fixed bin(35)) returns (ptr);

ptr_value = cv_ptr_ (vptr, code);
```

where:

1. vptr is the virtual pointer to be converted. (Input) See "Virtual Pointers" below for more information.

2. code is a standard status code. (Output)

3. ptr_value is the pointer that results from the conversion. (Output)

Entry: cv_ptr_$terminate

This entry point is called to terminate the segment that has been initiated by a previous call to cv_ptr_.

Usage

```
declare cv_ptr_$terminate (ptr);

call cv_ptr_$terminate (ptr_value);
```

where ptr_value is the pointer returned by the previous call to cv_ptr_. (Input)

Notes

Pointers returned by the cv_ptr_ subroutine cannot be used as entry pointers in calls to cu_$gen_call or cu_$make_entry_value. The cv_ptr_ subroutine constructs the returned pointer to a segment in a way that avoids copying of the segment's linkage and internal static data into the combined linkage area. The cv_entry_ subroutine is used to convert virtual entries to an entry value.

The segment pointed to by the returned ptr_value is initiated with a null reference name. The cv_ptr_$terminate entry point should be called to terminate this null reference name.

Virtual Pointers

The cv_ptr_ subroutine converts virtual pointers that contain one or two components -- a segment identifier and an optional offset into the segment. Altogether, fourteen forms are accepted. They are shown in the table below.

In the table that follows, W is an octal word offset from the beginning of the segment. It may have a value from 0 to 777777 inclusive. B is a decimal bit offset within the word. It may have a value from 0 to 35 inclusive.

Virtual Pointer	Interpretation
path|W(B)	points to octal word W, decimal bit B of segment identified by absolute or relative pathname path.
path|W	same as path|W(0).
path|	same as path|0(0).
path	same as path|0(0).
path|entry_pt	points to word identified by entry point entry_pt in segment identified by path.
ref_name$entry_pt	points to word identified by entry point entry_pt in segment whose reference name is ref_name.
ref_name$W(B)	points to octal word W, decimal bit B of segment whose reference name is ref_name.
ref_name$W	same as ref_name$W(0).
ref_name$	same as ref_name$0(0).
segno|W(B)	points to octal word W, decimal bit B of segment whose octal segment number is segno.
segno|W	same as segno|W(0).
segno|	same as segno|0(0).
segno	same as segno|0(0).
segno|entry_pt	points to word identified by entry point entry_pt in segment whose octal segment number is segno.

A null pointer is represented by the virtual pointer 77777|1, by -1|1, or by -1.

Name: delete_external_variables

The delete_external_variables command deletes from the user's name space specified variables managed by the system for the user. All links to those variables are unsnapped and their storage is freed.

Usage

delete_external_variables names {-control_arg}

where:

1. names
   are the names of the external variables, separated by spaces, to be deleted.

2. control_arg
   is -unlabeled_common (or -uc) to indicate unlabeled (or blank) common.

display cobol run unit, dcr

The display_cobol_run_unit command displays the current state of a COBOL run unit. The minimal information displayed tells which programs compose the run unit. Optionally, more detailed information can be displayed concerning active files, data location, and other aspects of the run unit. Refer to the run_cobol command for information concerning the run unit and the COBOL runtime environment.

Usage

```
display_cobol_run_unit {-control_args}
```

where control_args may be chosen from the following list:

-long, -lg
: causes more detailed information about each COBOL program in the run unit to be displayed.

-files
: displays information about the current state of the files that have been referenced during the execution of the current run unit.

-all, -a
: prints information about all programs in the run unit, including those that have been cancelled.

Note

Refer to the following related commands:

run_cobol, rc
stop_cobol_run, scr
cancel_cobol_program, ccp

Name: display_pllio_err, dpe

The display_pllio_error command is designed to be invoked after the occurrence of an I/O error signal during a PL/I I/O operation. It describes the most recent file on which a PL/I I/O error was raised and displays diagnostic information associated with that type of error.

Usage

```
display_pllio_error
```

Example

The command line:

```
display_pllio_error
```

might respond with the following display:

```
Error on file afile
Title: vfile_ afile
Attributes: open input keyed record sequential
Last i/o operation attempted: write from
Attempted "write" operation conflicts with file "input" attribute.
Attempted "from" operation conflicts with file "input" attribute.
```

Name: dump_segment, ds

The dump_segment command prints, in octal or hexadecimal format, selected portions of a segment. It prints out either four or eight words per line and can optionally be instructed to print out an edited version of the ASCII, BCD, EBCDIC (in 8 or 9 bits), or 4-bit byte representation.

Usage

```
dump_segment path {first} {n_words} {-control_args}
```

where:

1. path
   is the pathname or (octal) segment number of the segment to be dumped. If path is a pathname, but looks like a number, the preceding argument should be the -name (or -nm) control argument (see below).

2. first
   is the (octal) offset of the first word to be dumped. If both first and n_words are omitted, the entire segment is dumped.

3. n_words
   is the (octal) number of words to be dumped. If first is supplied and n_words is omitted, 1 is assumed.

4. control_args
   can be chosen from the following:

   -4bit
   prints out a translation of the octal or hexadecimal dump based on the Multics unstructured 4-bit byte. The translation ignores the first bit of each 9-bit byte and uses each of the two groups of four bits remaining to generate a digit or a sign.

   -address, -add
   prints the address (relative to the base of the segment) with the data. This is the default.

   -bcd
   prints the BCD representation of the words in addition to the octal or hexadecimal dump. There are no nonprintable BCD characters, so periods can be taken literally.

   -block N, -bk N
   dumps words in blocks of N words separated by a blank

line. The offset, if being printed, is reset to initial value at the beginning of each block.

-character, -ch, -ascii
prints the ASCII representation of the words in addition to the octal or hexadecimal dump. Characters that cannot be printed are represented as periods.

-ebcdic9
prints the EBCDIC representation of each 9-bit byte in addition to the octal or hexadecimal dump. Characters that cannot be printed are represented by periods.

-ebcdic8
prints the EBCDIC representation of each eight bits in addition to the octal or hexadecimal dump. Characters that cannot be printed are represented by periods. If an odd number of words is requested to dump, the last four bits of the last word do not appear in the translation.

-header, -he
prints a header line containing the pathname (or segment number) of the segment being dumped as well as the date-time printed. The default is to print a header only if the entire segment is being dumped, i.e., neither the first nor the n_words arguments is specified.

-hex8
prints the dumped words in hexadecimal with nine hexadecimal digits per word rather than octal with 12 octal digits per word.

-hex9
prints the dumped words in hexadecimal with eight hexadecimal digits per word rather than 12 octal digits per word. Each pair of hexadecimal digits corresponds to the low-order eight bits of each 9-bit byte.

-long, -lg
prints eight words on a line. Four is the default. This control argument cannot be used with -character, -bcd, -4bit, -ebcdic8, -ebcdic9, or -short. (Its use with these control arguments, other than -short, results in a line longer than 132 characters.)

-name, -nm
indicates that the following argument is a pathname even though it may look like an octal segment number.

-no_address, -nad
does not print the address.

-no_header, -nhe
suppresses printing of the header line even though the entire segment is being dumped.

-no_offset, -nofs
does not print the offset. This is the default.

-offset N, -ofs N
prints the offset (relative to N words before the start of data being dumped) along with the data. If N is not given, 0 is assumed.

-short, -sh
compacts lines to fit on a terminal with a short line length. Single spaces are placed between fields, and only the two low-order digits of the address are printed, except when the high-order digits change. This shortens output lines to less than 80 characters.

Note

Only one of the control arguments: -ebcdic8, -ebcdic9, -character, -bcd, or -4bit can be specified.

Name: io_call, io

The io_call command performs an operation on a designated I/O switch.

Usage

```
io_call opname switchname {args}
```

where:

1. opname
   designates the operation to be performed.

2. switchname
   is the name of the I/O switch.

3. args
   may be one or more arguments, depending on the particular operation to be performed.

The opnames permitted, followed by their alternate forms where applicable, are:

attach	look_iocb
close	open
control	position
delete_record, delete	print_iocb
detach_iocb, detach	put_chars
destroy_iocb	read_key
find_iocb	read_length
get_chars	read_record, read
get_line	rewrite_record, rewrite
modes	seek_key
move_attach	write_record, write

Usage is explained below under a separate heading for each designated operation. The explanations are arranged functionally rather than alphabetically.

Unless otherwise specified, if a control block for the I/O switch does not already exist, an error message is printed on error_output and the operation is not performed. If the requested operation is not supported for the switch's attachment and/or opening, an error message is printed on error_output.

The explanations of the operations cover only the main points of interest and, in general, treat only the cases where the I/O switch is attached to a file or device. For full details see the descriptions of the iox_ subroutine and the I/O modules in the MPM Subroutines and Section V, "Input and Output Facilities," in the MPM Reference Guide.

Operation: attach

io_call attach switchname modulename {args}

where:

1. modulename
   is the name of the I/O module to be used in the attachment. If modulename contains less-than (<) or greater-than (>) characters, it is assumed to be a pathname, otherwise, it is a reference name.

2. args
   may be one or more arguments, depending on what is permitted by the particular I/O module.

This command attaches the I/O switch using the designated I/O module. The attach description is the concatenation of modulename and args separated by blanks. The attach description must conform to the requirements of the I/O module. If the I/O modulename is specified by a pathname, it is initiated with a reference name equal to the entryname. If the entryname or reference name does not contain a dollar sign ($), the attachment will be made by calling modulename$modulenameattach. If a dollar sign is specified, the entry point specified is called. See "Entry Point Names" in the MPM Reference Guide.

If a control block for the I/O switch does not already exist, one is created.

Operation: detach_iocb, detach

io_call detach switchname

This command detaches the I/O switch.

Operation: open

io_call open switchname mode

where mode is one of the following opening modes, which may be specified by its full name, or by an abbreviation:

stream_input, si
stream_output, so
stream_input_output, sio
sequential_input, sqi
sequential_output, sqo
sequential_input_output, sqio
sequential_update, squ
keyed_sequential_input, ksqi
keyed_sequential_output, ksqo
keyed_sequential_update, ksqu
direct_input, di
direct_output, do
direct_update, du

This command opens the I/O switch with the specified opening mode.

Operation: close

io_call close switchname

This command closes the I/O switch.

Operation: get_line

io_call get_line switchname {N} {-control_args}

where:

1. N
   is a decimal number greater than zero specifying the maximum number of characters to be read.

2. control_args
   can be selected from the following:

   -segment path {offset}, -sm path {offset}
   specifies that the line read from the I/O switch is to be stored in the segment specified by path, at the location specified by offset.

   -nnl
   specifies that the newline character, if present, is deleted from the end of the line.

-nl
: specifies that a newline character is added to the end of the line if one is not present.

-lines
: specifies that the offset, if given, is measured in lines rather than characters. This control argument only has meaning if the -segment control argument is also specified.

This command reads the next line from the file or device to which the I/O switch is attached. If N is given, and the line is longer than N, then only the first N characters are read.

If the -segment control argument is not specified, the line read is written onto the I/O switch user_output, with a newline character appended if one is not present and -nnl has not been specified.

If the -segment control argument is specified, the line is stored in the segment specified by path. If this segment does not exist, it is created. If offset is specified, the line is stored at that position relative to the start of the segment. This is normally measured in characters, unless -lines has been used. If offset is omitted, the line is appended to the end of the segment. The bit count of the segment is always updated to a point beyond the newly added data.

Operation: get_chars

io_call get_chars switchname N {-control_args}

where:

1. N
   is a decimal number greater than zero specifying the number of characters to read.

2. control_args
   can be selected from the same list as described under the get_line operation.

This command reads the next N characters from the file or device to which the I/O switch is attached. The disposition of the characters read is the same as described under the get_line operation; that is, they are written upon user_output if the -segment control argument is not specified, or stored in a segment if the -segment control argument is specified.

Operation: put_chars

```
io_call put_chars switchname {string} {-control_args}
```

where:

1. string
   may be any character string.

2. control_args
   can be selected from the following:

   -segment path {length}, -segment path {offset} {length},
   -sm path {length}, -sm path {offset} {length}
   specifies that the data for the output operation is to be found in the segment specified by pathname. The location and length of the data may be optionally described with offset and length parameters.

   -nnl
   specifies that a newline character is not to be appended to the end of the output string.

   -nl
   specifies that a newline character is to be added to the end of the output line if one is not present.

   -lines
   specifies that offsets and lengths are measured in lines instead of characters.

The string parameter and the -segment control argument are mutually exclusive. If a string is specified, the contents of the string are the data output to the I/O switch. If the -segment control argument is specified, the data is taken from the segment specified by path, at the offset and length given. If offset is omitted, the beginning of the segment is assumed. If length is omitted, the entire segment is output.

If the I/O switch is attached to a device, this command transmits the characters from the string or the segment to the device. If the I/O switch is attached to an unstructured file, the data is added to the end of the file. The -nl control argument is the default on a put_chars operation: a newline character is added unless one is already present, or the -nnl control argument is specified.

Operation: read_record, read

```
io_call read_record switchname N {-control_args}
```

where:

1. N
   is a decimal integer greater than zero specifying the size of the buffer to use.

2. control_args
   can be selected from the same list as described under the get_line operation.

This command reads the next record from the file to which the I/O switch is attached into a buffer of length N. If the -segment control argument is not specified, the record (or the part of it that fits into the buffer) is printed on user_output. If the -segment control argument is specified, the record is stored in a segment as explained under the get_chars operation.

Operation: write_record, write

```
io_call write_record switchname {string} {-control_args}
```

where:

1. string
   is any character string.

2. control_args
   may be selected from the same list as described under the put_chars operation.

This command adds a record to the file to which the I/O switch is attached. If the string parameter is specified, the record is equal to the string. If the -segment control argument is specified, the record will be extracted from the segment as described under the put_chars operation. In either case, the -nnl control argument is the default: a newline character is added only if the -nl control argument is specified. If the file is a sequential file, the record is added at the end of the file. If the file is an indexed file, the record's key must have been defined by a preceding seek_key operation.

Operation: rewrite_record, rewrite

io_call rewrite_record switchname {string} {-control_args}

where:

1. string
   is any character string.

2. control_args
   may be selected from the same list as described under the put_chars operation.

This command replaces the current record in the file to which the I/O switch is attached. The new record is either the string parameter, or is taken from a segment, as described under the write_record operation. The current record must have been defined by a preceding read_record, seek_key, or position operation as follows:

read_record
   current record is the last record read.

seek_key
   current record is record with the designated key.

position
   current record is the record preceding the record to which the file was positioned.

Operation: delete_record, delete

io_call delete_record switchname

This command deletes the current record in the file to which the I/O switch is attached. The current record is determined as in rewrite_record above.

Operation: position

io_call position switchname type

where type may be one of the following:

bof
: sets position to beginning of file

eof
: sets position to end of file

forward N, fwd N, f N
: sets position forward N records or lines (same as reverse N)

reverse N, rev N, r N
: sets position back N records (same as forward -N records)

other
: any other numeric argument or pair of numeric arguments may be specified, but its function depends on the I/O module being used and may not be implemented for all I/O modules.

This command positions the file to which the I/O switch is attached. If type is bof, the file is positioned to its beginning, so that the next record is the first record (structured files), or so that the next byte is the first byte (unstructured files). If type is eof, the file is positioned to its end; the next record (or next byte) is at the end-of-file position. If type is forward or reverse the file is positioned forwards or backwards over records (structured files) or lines (unstructured files). The number of records or lines skipped is determined by the absolute value of N.

In the case of unstructured files, the next byte position after the operation is at a byte immediately following a newline character (or at the first byte in the file or at the end of the file); and the number of newline characters moved over is the absolute value of N.

If the I/O switch is attached to a device, only forward skips (where type is forward) are allowed. The effect is to discard the next n lines input from the device.

Operation: seek_key

io_call seek_key switchname key

where key is a string of ASCII characters with 0!<u><</u>!length!<u><</u>!256.

This command positions the indexed file to which the I/O switch is attached to the record with the given key. The record's length is printed on user_output. Trailing blanks in the key are ignored.

If the file does not contain a record with the specified key, it becomes the key for insertion. A following write_record operation adds a record with this key.

Operation: read_key

io_call read_key switchname

This command prints, on user_output, the key and record length of the next record in the indexed file to which the I/O switch is attached. The file's position is not changed.

Operation: read_length

io_call read_length switchname

This command prints, on user_output, the length of the next record in the structured file to which the I/O switch is attached. The file's position is not changed.

Not To Be Reproduced

Operation: control

io_call control switchname order {args}

where:

1. order
   is one of the orders accepted by the I/O module used in the attachment of the I/O switch.

2. args
   are additional arguments dependent upon the order being issued and the I/O module being used.

This command applies only when the I/O switch is attached via an I/O module that supports the control I/O operation. The exact format of the command line depends on the order being issued and the I/O module being used. For more details, refer to "Control Operations from Command Level" in the appropriate I/O module in the MPM Subroutines. If the I/O module supports the control operation and the paragraph just referenced does not appear, it can be assumed that only control orders that do not require an info_structure can be performed with the io_call command, as a null info_ptr is used. (See the description of the iox_$control entry point and the I/O module's control operation, both in the MPM Subroutines.)

Operation: modes

io_call modes switchname {string} {-control_arg}

where:

1. string
   is a sequence of modes separated by commas. The string must not contain blanks.

2. control_arg
   may be -brief or -bf.

This command applies only when the I/O switch is attached via an I/O module that supports modes. The command sets only new modes specified in string, and then prints the old modes on user_output. Printing of the old modes is suppressed if the -brief control argument is used.

If the switch name is user_i/o, the command refers to the modes controlling the user's terminal. See the I/O module tty_ subroutine description in the MPM Subroutines for an explanation of applicable modes.

Operation: find_iocb

io_call find_iocb switchname

This command prints, on user_output, the location of the control block for the I/O switch. If it does not already exist, the control block is created.

Operation: look_iocb

io_call look_iocb switchname

This command prints, on user_output, the location of the control block for the I/O switch. If the I/O switch does not exist, an error is printed.

Operation: move_attach

io_call move_attach switchname switchname2

where switchname2 is the name of a second I/O switch.

This command moves the attachment of the first I/O switch (switchname) to the second I/O switch (switchname2). The original I/O switch is left in a detached state.

Operation: destroy_iocb

io_call destroy_iocb switchname

This command destroys the I/O switch by deleting its control block. The switch must be in a detached state before this command is used. Any pointers to the I/O switch become invalid.

Operation: print_iocb

io_call print_iocb switchname

This command prints, on user_output, all of the data in the control block for the I/O switch, including all pointers and entry variables.

Summary of Operations

```
Usage:  io attach switchname modulename {args}
Usage:  io detach switchname
Usage:  io open switchname mode
Usage:  io close switchname
Usage:  io get_line switchname {N} {-control_args}
Usage:  io get_chars switchname N {-control_args}
Usage:  io put_chars switchname {string} {-control_args}
Usage:  io read_record switchname N {-control_args}
Usage:  io write_record switchname {string} {-control_args}
Usage:  io rewrite_record switchname {string} {-control_args}
Usage:  io delete_record switchname
Usage:  io position switchname type
Usage:  io seek_key switchname key
Usage:  io read_key switchname
Usage:  io read_length switchname
Usage:  io control switchname order {args}
Usage:  io modes switchname {string} {-control_arg}
Usage:  io find_iocb switchname
Usage:  io look_iocb switchname
Usage:  io move_attach switchname switchname2
Usage:  io destroy_iocb switchname
Usage:  io print_iocb switchname
```

where:

1. switchname
   is the name of the I/O switch.

2. modulename
   is the name of I/O module used in the attachment.

3. args
   are any arguments accepted by the I/O module used in the attachment.

4. mode

   is one of the following modes:

   stream_input, si
   stream_output, so
   stream_input_output, sio
   sequential_input, sqi
   sequential_output, sqo
   sequential_input_output, sqio
   sequential_update, squ
   keyed_sequential_input, ksqi
   keyed_sequential_output, ksqo
   keyed_sequential_update, ksqu
   direct_input, di
   direct_output, do
   direct_update, du

5. N

   is a decimal number.

6. string

   is any character string.

7. type

   sets the file position. It can be:

   bof   forward N
   eof   reverse N
   other

8. key

   is a string of ASCII characters with 0!≤!length!≤!256.

9. order

   is one of the orders accepted by the I/O module used in the attachment of the I/O switch.

10. control_args

   can be chosen from the following:

   -segment path {length}, -sm path {length}
   -segment path {offset}, -sm path {offset}
   -segment path {offset} {length}, -sm path {offset} {length}
   -nnl
   -nl
   -lines
   -brief, -bf

Name: list_external_variables

The list_external_variables command prints information about variables managed by the system for the user, including FORTRAN common and PL/I external static variables whose names do not contain dollar signs. The default information is the location and size of each specified variable.

Usage

```
list_external_variables names {-control_args}
```

where:

1. names
   are names of external variables, separated by spaces.

2. control_args
   can be chosen from the following:

   -unlabeled_common, -uc
   is the name for unlabeled (or blank) common.

   -long, -lg
   prints how and when the variables were allocated.

   -all, -a
   prints information for each variable the system is managing.

   -no_header, -nhe
   suppresses the header.

Name: list_temp_segments

The list_temp_segments command lists the segments currently in the temporary segment pool associated with the user's process. This pool is managed by the get_temp_segments_ and release_temp_segments_ subroutines (described in the MPM Subroutines).

Usage

list_temp_segments {names} {-control_arg}

where:

1. names
   is a list of names identifying the programs whose temp segments are to be listed.

2. control_arg
   is -all (or -a) to list all temporary segments. If the command is issued with no control argument, it lists only those temporary segments currently assigned to some program.

Examples

To list all the segments currently in the pool, type:

```
!   list_temp_segments -all

         5 Segments, 2 Free

    !BBBCdfghgffkkkl.temp.0246   work
    !BBBCdffddfdffkl.temp.0247   work
    !BBBCddffdfffhhh.temp.0253   (free)
    !BBBCdgdgfhfgfsf.temp.0254   (free)
    !BBBCvdvfgvdgvvv.temp.0321   editor
```

To list the segments currently in use, type:

```
!   list_temp_segments

         3 Segments

    !BBBCdfghgffkkkl.temp.0246   work
    !BBBCdffddfdffkl.temp.0247   work
    !BBBCvdvfgvdgvvv.temp.0321   editor
```

To list segments used by the program named editor, type:

```
!  list_temp_segments editor

      1 segment

   !BBBCvdvfgvdgvvv.temp.0321   editor
```

Name: page_trace, pgt

The page_trace command prints a recent history of page faults and other system events within the calling process.

Usage

page_trace {N} {-control_args}

where:

1. N

   prints the last N system events (mostly page faults) recorded for the calling process. If N is not specified, then all the entries in the system trace list for the calling process are printed. Currently, there is room for approximately 350 entries in the system trace array.

2. control_args

   can be chosen from the following:

   -from STR, -fm STR
      searched the trace array for a user marker matching STR. If one is found, printing begins with it; otherwise, printing begins with the first element in the array.

   -long, -lg
      prints full pathnames where appropriate. The default is to print only entrynames.

   -no_header, -nhe
      suppresses the header that names each column. The default is to print the header.

   -output_switch swname, -os swname
      writes all output on the I/O switch named swname, which must already be attached and open for stream_output. The default is to write all output on the user_output I/O switch.

   -to STR
      stops printing if a user marker marching STR is found. The default is to print until the end of the array. If both -from and -to are specified, the from marker is assumed to occur before the to marker.

Output

The first column of output describes the type of trace entry. An empty column indicates that the entry is for a page fault. The second column of output is the real time, in milliseconds, since the previous entry's event occurred. The third column (printed for page faults only) is the ring number in which the page fault occurred. The fourth column of output contains the page number for entries, where appropriate. The fifth column gives the segment number for entries, where appropriate. The last column is the entryname (or pathname) of the segment for entries, where appropriate.

Notes

Since it is possible for segment numbers to be reused within a process, and since only segment numbers (not entrynames or pathnames) are kept in the trace array, the entrynames and pathnames associated with a trace entry may be for previous uses of the segment numbers, not the latest ones. In fact, the entry and pathnames printed are the current ones appropriate for the given segment number.

For completeness, events occurring while inside the supervisor are also listed in the trace. The interpretation of these events sometimes requires detailed knowledge of the system structure; in particular, they may depend on activities of other users. For many purposes, the user will find it appropriate to identify the points at which he enters and leaves the supervisor and ignore the events in between.

Typically, any single invocation of a program does not induce a page fault on every page touched by the program, since some pages may still be in primary memory from previous uses or use by another process. It may be necessary to obtain several traces to fully identify the extent of pages used.

A count value (N) and either the -from or -to control argument cannot be specified in the same invocation of the page_trace command.

Name: print_bind_map

The print_bind_map command displays all or part of the bind map of an object segment generated by version number 4 or subsequent versions of the binder.

Usage

```
print_bind_map path {components} {-control_args}
```

where:

1. path
   is the pathname of a bound object segment.

2. components
   are the optional names of one or more components of this bound object and/or the bindfile name. Only the lines corresponding to these components are displayed. A component name must contain one or more nonnumeric characters. If it is purely numerical, it is assumed to be an octal offset within the bound segment and the lines corresponding to the component residing at that offset are displayed. A numerical component name can be specified by preceding it with the -name control argument (see below). If no component names are specified, the entire bind map is displayed.

3. control_args
   may be chosen from the following list:

   -long, -lg
      prints the components' relocation values (also printed in the default brief mode), compilation times, and source languages.

   -name STR, -nm STR
      is used to indicate that STR is really a component name, even though it appears to be an octal offset.

   -no_header, -nhe
      omits all headers, printing only lines concerning the components themselves.

Name: print_link_info, pli

The print_link_info command prints selected items of information for the specified object segments.

Usage

```
print_link_info paths {-control_args}
```

where:

1. paths
   are the pathnames of object segments.

2. control_args
   can be chosen from the following list. (See "Note" below.)

   -length, -ln
   print only the lengths of the sections in path*i*.

   -entry, -et
   print only a listing of the path*i* external definitions, giving their symbolic names and their relative addresses within the segment.

   -link, -lk
   print only an alphabetically sorted listing of all the external symbols referenced by path*i*.

   -long
   prints more information when the header is printed. Additional information includes a listing of source programs used to generate the object segment, the contents of the "comment" field of the symbol header (often containing compiler options), and any unusual values in the symbol header.

   -header, -he
   prints the header (The header is not printed by default, if the -length, -entry, or -link control argument is specified.)

   -no_header
   suppresses printing of the header.

Note

Control arguments can appear anywhere on the command line and apply to all pathnames.

Example

```
!  print_link_info program -long -length

                    program 07/30/76  1554.2 edt Fri


Object Segment >udd>Work>Wilson>program
Created on 07/30/76  0010.1 edt Fri
by Wilson.Work.a
using Experimental PL/I Compiler of Thursday, July 26, 1976 at 21:38


Translator:         PL/I
Comment:            map table optimize
Source:
 07/30/76  0010.1 edt Fri  >user_dir_dir>work>Wilson>s>s>program.pl1
 12/15/75  1338.1 edt Mon  >library_dir_dir>include>linkdcl.incl.pl1
 06/30/75  1657.7 edt Mon  >library_dir_dir>include>object_info.incl.pl1
 10/06/72  1206.8 edt Fri  >library_dir_dir>include>source_map.incl.pl1
 05/18/72  1512.4 edt Thu  >library_dir_dir>include>symbol_block.incl.pl1
 01/17/73  1551.4 edt Wed  >library_dir_dir>include>pl1_symbol_block.incl.pl1
Attributes:          relocatable,procedure,standard

        Object    Text    Defs    Link      Symb   Static
Start        0       0    3450    3620      3656     3630
Length   11110    3450     150      36      5215        0


<ready>
```

Also printed is:

Severity, if it is nonzero.
Entrybound, if it is nonzero.
Text Boundary, if it is not 2.
Static Boundary, if it is not 2.

Name: print_linkage_usage, plu

The print_linkage_usage command lists the locations and size of linkage and static sections allocated for the current ring. This information is useful for debugging purposes or for analysis of how a process uses its linkage segments.

A linkage section is associated with every procedure segment and every data segment that has definitions.

Usage

```
print_linkage_usage
```

Note

For standard procedure segments, the information printed includes the name of the segment, its segment number, the offset of its linkage section, and the size (in words) of both its linkage section and its internal static storage.

Name: probe, pb

The probe command provides symbolic, interactive debugging facilities for programs compiled with PL/I, FORTRAN, or COBOL. Its features permit a user to interrupt a running program at a particular statement, examine and modify program variables in their initial state or during execution, examine the stack of block invocations, and list portions of the source program. External subroutines and functions may be invoked, with arguments as required, for execution under probe control. The probe command may be called recursively.

Usage

probe {procedure_name}

where procedure_name is an optional argument that gives the symbolic name of an entry to the procedure or subroutine that is to be examined with probe. It can take the form reference_name$offset_name. If no procedure_name argument is specified, the procedure owning the frame in which the last condition was raised is assumed, if one exists; otherwise, an error is reported.

Overview of Processing

The probe command is generally used to examine an active program at points where execution has been suspended by one of the following:

1. Breakpoint. Execution is temporarily halted at a point selected by the user and probe entered directly. Debugging requests associated with the breakpoint are automatically carried out and/or requests issued from the user's terminal. Program execution can be resumed at the point of interruption.

2. Error. An error such as zerodivide or subscriptrange can interrupt program execution. After an error message is printed, a new command level is established. The user can then call probe to examine the state of the program.

3. Quit signal. A run-away or looping program can be stopped by issuing a quit signal. A new command level is established and the user can call probe to determine the source of the problem.

In all of these cases, variables of all storage classes (including automatic) are accessible.

The probe command can also be used to examine a nonactive program -- one that has never been run or that has completed execution -- by specifying a procedure_name argument in the command line. In this case, the user can examine static variables and the program source. However, the most common use is to set breaks before actually running the program.

A program to be debugged with probe must have a standard symbol table that contains information about variables defined in the program and a statement map that gives the correspondence between source statements and object code. A symbol table and statement map are produced for the languages supported if the -table control argument is given at compilation. (A program may also be compiled with the -brief_table control argument, which produces only a statement map. The variables of a program compiled in this way cannot be examined with probe; however, the user may retrieve information about source statements and where the program was interrupted and also may set breakpoints at particular statements.)

Information about programs being debugged is stored by probe in a segment in the user's home directory called Person_id.probe where Person_id is the user's log-in name. This segment is created automatically when needed.

Probe Pointers

Three internal "pointers" are used by probe to keep track of the program's state. They are:

source pointer	indicates the current source-program statement
block pointer	indicates the current block
control pointer	indicates the current control point

These values are affected by certain probe requests. A user can, for example, position the source pointer to a particular statement, then list a portion of the source program beginning at that point.

The block pointer serves two purposes. It identifies the procedure, subprogram, or begin block whose variables are to be examined. Further, it specifies the stack frame associated with the block and is used to distinguish among different occurrences of an automatic variable in a recursively invoked procedure. The control pointer marks the point at which a program is suspended.

The initial values of these pointers are determined as described below. If a procedure_name argument is given in the command line and

if the designated program is active, the control and source pointers are set to the last statement executed, and the block pointer is set to the most recent invocation of the procedure. If the designated program is not active, then the control and source pointers are set to the entry statement, and the block pointer to the outermost block (but with no active frame).

If no procedure_name argument is given and the default rule applies (i.e., a condition has been raised), then the procedure in which the condition was raised is used. The source and control pointers are set to the statement where the condition was raised, and the block pointer to the block containing that statement.

Similarly, when probe is entered because of a breakpoint encountered during the execution of a program, the source and control pointers are set to the statement at which the break has been set; and the block pointer to the block containing that statement.

Breakpoints

A breakpoint causes a temporary interruption of program execution, during which debugging operations can be performed. Using probe requests, a user can set a breakpoint before or after any statement and can associate a list of probe requests with the break. A break set after a statement may, in some cases, not be executed due to the nature of the code generated for that statement. When the break is encountered during execution, probe is entered and the list of requests interpreted automatically. These requests might, for example, display the value of a variable or alter its value (effectively allowing source level patching of the program), tell what line was just executed, or cause probe to read a list of requests from the terminal to permit the user to interactively examine the state of his program. When the request list associated with the break is exhausted, the execution of the program is resumed from the point at which it was interrupted.

The implementation of a breakpoint by probe consists of patching a call to the probe command into the appropriate location in the object segment of the program. As a result, there need not be an active invocation of probe for a break to occur; also, breakpoints may be set in a program before it is run, while the program is suspended by another break, or before a program interrupted by a quit signal or error condition has been restarted.

Probe Requests

A probe request consists of a keyword (or its abbreviation) that specifies the desired function and any arguments required by the particular request.

A series of requests may be given in the form of a request list. Here, individual requests are separated by semicolons or newline characters.

A single request or a parenthesized request list may be preceded by a conditional predicate whose value determines if and when the requests it modifies will be executed.

The following pages present the format and function of each probe request. Requests are grouped according to function. Required arguments are indicated for each. The syntax and semantics of generic arguments such as expressions, procedures, labels, and variables are defined under separate headings following the request descriptions.

The following descriptions first give the name of the request and its abbreviated form (if any). This line is followed by the general format line(s) of the request.

BASIC REQUESTS

1. value, v

    value expression
    value cross-section

The request "value expression" causes the value of the given expression to be displayed. Allowable expressions are variables, builtin functions such as addr and octal, and the value returned by an external function. The evaluation of expressions is described later (following the descriptions of all the requests) under "Evaluation of Expressions."

Examples:

```
value var
value p -> a.b(j).c
value addr (i)
value octal (ptr)
value function (2)
```

The request "value cross-section" is used to display values contained in a cross-section of an array. A cross-section is specified by giving the upper and lower bounds of one or more subscripts, as in:

```
value array (1:5, 1)
```

The notation 1:5 indicates the range one through five for the first subscript. The example above prints array(1,1), array(2,1), ..., array(5,1). More than one dimension can be iterated; for instance, array(1:2,1:2) prints, in order, array(1,1), array(1,2), array(2,1), array(2,2).

2. let, l

```
let variable = expression
let cross-section = expression
```

This request sets the specified variable or array elements to the value of the expression. If the variable and the expression are of different data types, conversion is performed according to the rules of PL/I. Array cross-sections are expressed as shown in the value request above. One array cross-section may not be assigned to another.

Examples:

```
let var = 2
let array (2,3) = i + 1
let p -> a.b(1:2).c = 10b
let ptr = null
```

Because of compiler optimization, the change may not take immediate effect in the program, though the value request shows the value to be altered.

3. call, cl

call procedure (arg1, ..., argn)

This request calls the procedure named with the arguments given. If the procedure expects arguments of a certain type, those given are converted to the expected type; otherwise, they are passed without conversion. The value request (see above) can be used to invoke a function, with the same sort of argument conversion taking place. If the procedure has no arguments, a null argument list, "()", must be given.

Examples:

```
call sub ("abc", p -> p2 -> bv, 25o, addr(j))
call sub_noargs ()
value function ("010"b)
```

4. goto, g

goto label

This request transfers control from probe to the statement specified and initiates program execution at that point.

Examples:

goto label_var	transfer to value of label variable
goto action (3)	transfer to label constant
goto 29	transfer to statement on line 29 of current program
goto $110	transfer to line with label 110 in the FORTRAN program
goto $c,1	transfer to the statement following the current statement

Because of compiler optimization, unpredictable results may occur when using this request.

5. quit, q

quit

This request causes a return to command level.

6. continue, c

continue

This request restarts a program that has been suspended by a break. If this request is issued in any other context, probe returns to its caller (generally command level).

SOURCE REQUESTS

1. source, sc

source n

This request displays one or more statements beginning with the current statement (i.e., the source pointer). If n is not specified, one line is printed; otherwise, n lines are printed. Only executable statements for which code has been generated can be listed; however, if a range of statements is requested, intervening text, such as comments and nonexecutable statements (for example, declarations), is included in the output.

2. position, ps

position label
position ±n

This request sets the source pointer to the statement indicated by label or to an executable statement relative to the current statement as indicated by the value of n and displays it if the user is in long mode. If +n is given, the pointer is set forward n statements; if -n is given, the pointer is set back n statements. If no label or offset is given, the statement designated by the control pointer is assumed.

Examples:

position here	set the source ptr to the statement labeled here
position action (3)	to the statement labeled action (3)
position 2-14	to the statement on line 14 of include file 2 of the program
position +2	move forward two statements in the source
position -5	move back five statements

The position request can also be used to search for an executable statement that contains a specified string, using the form:

position "string"

The search begins with the statement following the current statement and continues through the program, if necessary, until the current statement is again reached. If a match is found, the source pointer is set to that statement. If the specified string contains a quotation mark, it must be doubled when given in the request line. Because statements are reordered by the compiler, the search may not necessarily find statements in the same order as the source listing of the program would indicate.

Examples:

position "write (6,10)"	locate the statement in the program
position "str = ""a"	locate str = "a
position "q+2"; source	locate and print the statement

SYMBOL REQUESTS

1. stack, sk

   stack i, n all

This request traces the stack backward beginning at the ith frame and continuing for n frames. If i is not given, then the trace begins with the most recent frame and continues for n frames. If no limits are given, the entire stack is traced. The trace lists all active procedures and block invocations (including quick blocks) beginning with the most recent. For each block, a frame or level number is given, as is the name of any conditions raised in the frame.

Examples:

stack	trace the whole stack
stack 2	trace the two most recent frames
stack 3, 2	trace the third and second frames

Normally, system or subsystem support procedures are not included in the stack trace. These may be included by specifying "all".

Examples:

stack all	trace the whole stack including all support stack frames

stack 5,3 all	trace the fifth, fourth, and third frames including all support stack frames

2. use, u

use block

This request selects the block to be used for subsequent probe requests. It may be specified by the name of an entry, a label, or a stack frame number (level i). If no block is specified, then the block originally used (when probe was entered) is assumed. The block pointer is set to the specified block so that variables in that block can be referenced. In addition, the source pointer is set to the last statement executed in the block. In this way, the point at which the block exited can be found through use of the source request. Acceptable block specifications include:

procedure_name
label
level i
-n

In this context, procedure_name is the name of a procedure or subprogram entry point whose frame is desired; its usage is essentially the same as if used on the command line. A label denotes the block that contains the statement identified by the label or line number; for instance, the label on a begin statement denotes that begin block. If the label's block is not active, the source pointer is set to the statement specified. The block specification level i uses the block with level number i from a stack trace; -n uses the nth previous instance of the current block, allowing one to move back to a previous recursion level. If more frames are requested than actually exist, the last one found is used.

Examples:

use sub	use the block that procedure sub occupies
use label	use the block that contains the statement labeled label
use level 2	use the second frame in the stack trace
use -1	use the previous instance of the current block
use -999	use the last (oldest) instance

When a level is specified, the last trace mode (support procedures included or excluded) specified is used to find the level requested.

3. symbol, sb

   symbol identifier

This request displays the attributes of the variable specified and the name of the block in which its declaration is found. If the size or dimensions of the variable are not constant, an attempt is made to evaluate the size or extent expression; if the value is not available, an asterisk (*) is used instead.

4. where, wh

   where source
   where block
   where control

This request displays the current value of one or all of the pointers. Source and control give the statement number of the corresponding statement. Block gives the name of the block currently being used; if the block is active, its level number is also given. If neither source, block, or control appears, the information for all three is given.

Examples:

where	give the value of all three pointers
where source	give the value of the source pointer

BREAK REQUESTS

1. before, b

   before label: request
   before label: (request list)

This request sets a breakpoint before the statement specified by label and causes the given request(s) to be associated with the break. If no label is given, the current statement is assumed. If no requests are given, a halt is assumed (see the halt request described below).

When the running program arrives at the statement specified, probe is entered before the statement is executed, and associated requests are processed automatically. When all requests are done, execution of the program resumes at the statement before which the break was set. A breakpoint set before a statement takes effect whether the statement is arrived at in sequence or as the result of a branch or call from some other location.

Examples:

before: (value var; value var2)	set a break before the current statement to display the value of the variables var and var2
before quick: value x	set a break before the statement labeled quick
before	set a break containing the halt request before the current statement

The request list may extend across line boundaries if necessary.

2. after, a

```
after label:  request
after label:  (request list)
```

This request is the same as the before request except that the break is set after the designated statement. This means that the request list is interpreted after the statement has been executed. If the statement branches to another location in the program, the breakpoint does not take effect; also, in some cases, the break may not be executed due to the nature of the code generated for the statement.

Notice the distinction between two breakpoints in sequence. The one that is <u>after</u> statement x is not effective when control is passed to statement x+1 from elsewhere. The break <u>before</u> statement x+1 <u>does</u> take place.

3. halt, h

halt

This request causes probe to stop processing its current input and to read requests from the terminal. A new invocation of probe is created with new pointers set to the values at the time the halt request was executed. As part of a break request list, it enables the user to enter requests while a program is suspended by the break. A running program can be halted in this way. A subsequent continue request causes probe to resume what it was doing before it stopped; for example, finish a break request list and resume execution of the program.

Examples:

before 29: halt	causes the program to halt at statement 29 and allows the user to enter probe requests (the continue request can be used to restart the program)
after: (value a; halt; value b)	causes the value of a to be printed before the program halts; later, after the user enters a continue request, the value of b is printed, and the execution of the program is resumed

4. reset, r

reset
reset at/after/before label
reset procedure
reset *

This request deletes breaks set by the before and after requests. When no argument is supplied, reset deletes the current break. With a label argument, breaks set before and/or after a statement are deleted; with a procedure or asterisk (*) argument, all the breaks in a specified segment or all breaks in all segments, respectively, can be deleted.

Examples:

reset	delete the current break
reset at 34	delete breaks set before and after the first statement on line 34
reset after 34	delete the break set after line 34
reset sub	delete all breaks in sub
reset *	delete all known breaks

5. status, st

```
status
status at/after/before label
status procedure
status *
```

This request gives information about breaks that have been set by the user. The scope of the requests is similar to reset except that status without arguments specifies all breaks in the current program (the program containing the statement designated by the source pointer).

Examples:

status	list the breaks set in the current program
status before label	give the break set before the statement at label
status sub	list the breaks set in sub
status *	list the procedures that have breaks set in them

6. pause, pa

```
pause
```

This request is equivalent to "halt; reset" in a break request list. It causes the procedure to execute a break once and then reset it. If the statement after which the break is set transfers elsewhere, the break does not occur and remains set until encountered sometime in the future or explicitly reset at some other point.

7. step, s

step

This request enables the user to step through his program one statement at a time. It sets a break consisting of a pause request after the next statement to be executed (as indicated by the control pointer) and resumes the execution of the program as with a continue request.

MISCELLANEOUS REQUESTS

1. mode

mode brief
mode long

This request turns the brief message mode on or off. In brief mode, most messages generated by probe are shortened and others are suppressed altogether. The default is long.

2. execute, e

execute "string"

This request passes one or more Multics command lines, represented above by "string", to the command processor for execution.

3. acknowledge

.

This request causes probe to identify itself by printing "probe" on the terminal. It may be used, for example, to determine if a called procedure has returned.

CONDITIONAL PREDICATES

1. if

```
if conditional expression:  request
if conditional expression:  (request list)
```

The request or request list is executed if the conditional expression is true. The expression must be of the form:

```
expression operator expression
```

where operator can be <=, <, =, ^=, >, or >=.

Example:

```
if a < b: let p = addr (a)
```

This predicate is most useful in a break request list where it can be used to cause a conditional halt. For example,

```
before: if z ^= "10"b: halt
```

causes the program to stop only when z ^= "10"b.

2. while, wl

```
while conditional expression:  request
while conditional expression:  (request list)
```

The request or request list is executed repeatedly as long as the conditional expression is true.

Example:

```
while p ^= null: (value p -> r.val; let p = p -> r.next)
```

Evaluation of Expressions

Allowable expressions include simple scalar variables, constants, and probe builtin functions. The sum and difference of computational (arithmetic and string) values can also be used.

Variables can be simple identifiers, subscripted references, structure qualified references, and locator qualified references. Subscripts are also expressions. Locators must be offsets, pointer variables, or constants.

Examples:

```
running_total
salaries (p -> i - 2)
a.b(2).c(3)
a.b.c(2,3)
x.y -> var
```

Constants can be arithmetic, string, bit, and pointer. Arithmetic constants can be either decimal or binary, fixed or floating point, real or complex. Also, octal numbers are permitted as abbreviations for binary integers (e.g., 12o = 10).

Examples:

```
-123
10b
45.37
4.73e10
2.1-0.3i
12345670o
```

Character and bit strings without repetition factors are allowed. Character strings can include newline characters. Octal strings can be used in place of bit strings (e.g., "123"o = "001010011"b).

Examples:

```
"abc"
"quote""instring"
"1010"b
"01234567"o
```

A pointer constant is of the form:

segment_number|word_offset(bit_offset)

where the segment_number and word_offset must be in octal. The bit_offset is optional but if given must be in decimal. The pointer constant can be used as a locator.

Examples:

214|5764
232|7413(9)

Four builtin functions are provided by probe: addr, null, octal, and substr. The function of addr and null is the same as in PL/I: addr takes one argument and returns a pointer to its argument; null, taking no arguments, returns a null pointer. The function octal acts very much like the PL/I unspec builtin function in that it treats its argument as a bit string of the same length as the raw data value and can be used in a similar manner as a pseudo-variable. However, when used in the value request, the value is displayed in octal. Data items not occupying a multiple of three bits are padded on the right. The substr builtin function may be used as a function or pseudo-variable. It takes two or three arguments. The first argument must be a character or bit string or a reference to the octal builtin function; the second and optional third arguments give the offset and length of the desired substring as with the PL/I substr builtin function.

Note

These builtin functions cannot be used if a program variable of the same name appears in the block being referenced. (For example, if x and octal are arrays in the same block, then octal (x(2)) becomes a reference to the variable octal, not the probe builtin).

Examples:

For the following examples, assume that p is declared as an aligned pointer, i as fixed binary initial(-2), and cs as character(8) initial ("abcdefgh").

value addr (i)	displays the address of i
let p = null	sets the pointer, p, to null
value octal (i)	displays the storage containing i in octal, giving: 777777777776
value substr (cs, 2, 3)	displays "bcd"
let substr (cs, 4, 1)=""	sets cs to "abc fgh"

Label References

A label identifies a source program statement and can be a label variable or constant, a line number in source-listing format, or one of the following special statement designators:

$c	designates the "current statement"
$b	designates the statement on which the most recent break occurred
$number	designates a FORTRAN label

An optional offset of the form ",s" is also allowed.

Examples:

label	statement at label
label_var	statement to which label_var is set
17	statement on line 17 of program
3-14,2	statement 2 on line 14 of include file 3
$b	statement at which last break occurred
$c,1	statement after current statement
$100	FORTRAN statement labeled 100

Generally, a label can also be the name of a procedure, entry, or subroutine statement.

Procedure References

A procedure_name is an identifier representing an entry variable or constant. External reference names, representing entry points not declared in the current block, can be used.

Evaluation of Variable References

When a variable is referenced in a request, probe first attempts to evaluate it by checking for an applicable declaration in the current block and, if necessary, in its parents. If no declaration is found, the list of builtin functions is searched. Finally, when the context allows a procedure_name, a search is made following the user's search rules.

The block in which a variable reference is resolved can be altered by the use request that sets the current block. For example, if "value var" displays the value of var in the current block, then "use -1; value var" displays the value of var at the previous level of recursion. An optional block specification is available for referencing variables in other blocks:

```
variable [block]
```

where block is the same as in the use request. The use of blocks in this manner does not alter the block pointer.

Examples:

var[-1]	looks for the previous value of var
abc[other_block]	looks in "other_block" for abc
xyz[39]	looks in the block that contains line 39 for xyz
n.m[level 4]	looks in the block at level 4 for n.m
q(2)[sub]	looks in the procedure sub for q(2)

A block specification can be used to qualify a variable reference in any context the variable could be used. However, a block specification on a label or entry constant is ignored unless the relative (-n) format is used and the label or entry is itself used in a block specification. In such a case, it is taken to mean the nth previous instance of the block designated by the label or entry; that is, "var[sub[-2]]" references var in the second previous invocation (third on the stack) of sub.

Sample Debugging Sessions

Two extensive examples are given on the following pages to illustrate both how probe requests are used and how to get useful debugging information out of them. the first example was devised principally to demonstrate the application of probe requests. A listing of the source of the program, test, is given on the next page. The program has been compiled with the -table control argument (line 1). The sample output follows with an exclamation point (!) denoting lines typed by the user. Unless otherwise indicated, line numbers referenced in the following paragraphs are from the sample output.

The user first calls his program (line 5); noticing that it seems to be looping, he stops it by issuing the quit signal (line 6). After the user invokes probe (line 10), it responds by telling him that the internal function fun was executing line 38 when interrupted. Since the source pointer was automatically set to that line, the source request (line 12) causes the current source statement to be displayed. A statement causing an error could be displayed in a similar manner.

```
 1 test: procedure;
 2
 3      declare
 4
 5           (i, j) fixed binary,
 6           1 s structure based (p),
 7             2 num fixed binary,
 8             2 b (n refer (s.num)) float binary,
 9           p pointer, n fixed binary,
10           sysprint file;
11
12
13      n = 5;
14      allocate s set (p);
15
16      do i = 1 to s.num;
17           s.b(i) = fun (i, 1);
18      end;
19      put skip list (s.b);
20
21      do j = s.num to 1 by -1;
22           s.b(j) = fun (-j, -1);
23      end;
24      put skip list(s.b);
25
26      return;
27
28
29      fun: procedure (b, i) returns (float binary);
30
31           declare
```

```
32                  (b, i) fixed binary;
33
34             if b = 0
35                  then return (1);
36                  else do;
37                       b = b - i;
38                       return (2**b + fun (b, i));
39                  end;
40
41        end fun;
42
43
44  end test;
```

```
 1        ! pl1 test -table
            PL/I
            r 1248 3.211 28.336 280
 4
 5        ! test
 6   !(quit)
            QUIT
 8          r 1250 5.371 6.702 52  level 2, 10
 9
10        ! probe
11          Condition quit raised at line 38 of fun.
12        ! source
13                        return (2**b + fun (b, i));
14        ! stack
15          11    command_processor_
16          10    release_stack
17           9    unclaimed_signal
18           8    real_sdh_
19           7    return_to_ring_0_
20           6    fun                                  quit
21           5    test
22           4    command_processor_
23           3    listen_
24           2    process_overseer_
25           1    user_init_admin_
26        ! use level 5
27        ! source
28                     s.b(i) = fun (i, 1);
29        ! value s.num
30               5
31        ! position "i = 1"; source
32                do i = 1 to s.num;
33        ! after: value i
34          Break set after line 16 of test.
35        ! quit
36          r 1252 1.375 16.394 354  level 2, 10
37
38        ! release
39          r 1252 .126 .922 19
40
41        ! test
42               1
43               1
44               1
45               1
46   !(quit)
47          QUIT
48          r 1252 3.069 .650 25  level 2, 12
49
50        ! release
51          r 1253 .092 .937 20
52
53        ! probe test
```

```
 54      ! status
           Break after line 16.
 56      ! status after 16
 57        Break after line 16:  value i
 58      ! reset at 16
 59        Break reset after line 16 of test.
 60      ! position 34
 61      ! source
 62                  if b = 0
 63                         then return (1);
 64      ! before: halt
 65        Break set before line 34 of test.
 66      ! quit
 67        r 1255 .781 12.356 333
 68
 69      ! test
 70        Stopped before line 34 of fun.
 71      ! value b
 72              1
 73      ! where
 74        Current line is line 34 of test.
 75        Using level 6: fun.
 76        Control at line 34 of fun.
 77      ! value i
 78              1
 79      ! c
 80        Stopped before line 34 of fun.
 81      ! stack 5
 82           8    break
 83           7    fun
 84           6    fun
 85           5    test
 86           4    command_processor_
 87      ! value b
 88              0
 89      ! value b[-1]
 90              0
 91      ! value i
 92              1
 93      ! symbol i
 94        fixed binary(17,0) aligned parameter
 95        Declared in fun.
 96      ! use test
 97      ! value i
 98              0
 99      ! reset
100        Break reset before line 34 of test.
101      ! quit
102        r 1307 4.870 64.788 1544
```

The stack command is then used (line 14) to see in what order the procedures were called. The output shows that procedure test was

called from command level, and then called fun. While fun was executing, a quit signal was issued and established a new command level.

The use request (line 26) sets the block pointer to the outermost block of procedure test, and the source pointer to the last statement executed in that block -- the statement which invoked the function fun.

The source request (line 27) is issued to display the current statement (as set above) to determine from which line of the program (17 or 27) fun was actually invoked.

Since the block pointer has also been set, the user can check the value of "s.num" with the value request (line 28) and ascertain that it is as desired. Since there is no new declaration of "s.num" within the procedure fun, the declaration made in the parent block, test, is known and the value of "s.num" could be displayed without changing the block pointer as would be necessary if there were a conflicting declaration.

The user decides that it is worthwhile to trace the value of i. Rather than recompiling his program with a "put statement" added in a strategic location, probe allows him to set a break containing a value request to accomplish the same thing. The user wants to set the break after the do statement on line 16 of the program and searches for it with the position request (line 31). The source request is used to verify that the correct line was found. The after request is used to actually set the break (line 33). The quit request (line 35) then causes probe to return command level.

To abort the suspended program test, the user invokes the Multics release command (line 38). If he had done this just after issuing the quit signal, he could not have used probe to examine automatic variables inside the program or to determine where the program had been interrupted.

The program is restarted (line 41) but now, after each execution of line 16, the break occurs and probe displays the value of i. Clearly, it is not being incremented as it should. Since this approach is not producing any useful information, the user aborts the program and tries to delete the break. The status request is used to tell what breaks have been set in the procedure test (line 54), and then (line 56) to see the probe request associated with that break. The break is then deleted with the reset request (line 58). If there had also been a "Break before 16", then the request "reset at 16" would have deleted both.

The user next decides to examine fun, so he sets a break that will halt every time fun is invoked (lines 60 through 64). Looking at the listing, he sees that the first statement in fun is on line 34, so he sets the source pointer to that statement with the position request and sets a break to halt the program. To accomplish the same thing, "before 34: halt" could have been used.

The program is called (line 69) and then halts when the break before line 34 is reached. The user displays b and i (lines 71 and 77), getting the values he expected. The where request is also used (line 73) to check on the current state of things. The continue request (line 79) restarts fun, which calls itself recursively and stops again. The stack request (line 81, showing the last five frames) verifies that fact. The user displays the b in the current instance of fun (line 87, at level 7) and in the previous one (line 89, at level 6). Mistakenly expecting the b's at different levels to be different, he gets suspicious. The variable i has the value expected (line 91), but the symbol command (line 93) shows that it is the wrong instance of i -- the parameter to fun, not the loop index. To get the correct instance, he must look in the frame belonging to the procedure test (line 96) and display that i (line 97). This i has been set to 0. The user then realizes his error. The function is modifying its argument (the loop index i) on line 37 (line 94). When the user has finished debugging the program, the reset request (line 99) is used to delete the currently active break (the one that just occurred), and the program is aborted with the quit request (line 101).

The preceding example was constructed to give a user a feeling for applying probe requests. The following example is taken from an actual debugging session using probe and illustrates several additional techniques available to the user.

The program of interest is a subroutine, sort_strings, that is supposed to sort a character array of arbitrary dimension; the array is passed as an argument to the subroutine. Since very large strings are being compared, it would be time consuming to exchange the strings themselves. Therefore, an array of pointers to the strings (actually, the indices of the strings in the original array) is first sorted by a simple bubble sort, and the strings moved afterwards into the correct order. There are (at least) two bugs in the program as it appears in the listing. The next two paragraphs further describe the algorithm intended.

A bubble sort involves making repeated passes over an input array, comparing adjacent pairs of values, and interchanging them as necessary. This moves the larger (smaller) values toward the end of the array. The sort only covers that portion of the array that is out of order (i.e., up to the element where the final exchange took place

on the previous pass -- all elements following this point are clearly correctly arranged). The example below illustrates how a bubble sort works in one case. (The hyphen delimits the end of the search.)

Original		First Pass		Second Pass		Third Pass
d		a		a		-
a		c		b		a
c	->	b	->	-	->	b
b		-		c		c
e		d		d		d
-		e		e		e

In the sort_strings subroutine (see source listing below), "k" determines the last element of the array needing to be sorted. Sorting continues until no exchanges occurred during the last pass (i.e., until the test, k <= 1, fails). The "order" array contains the indices that are actually sorted.

The reordering method used is to scan for unordered items and then move the entire chain (a replaces b; b replaces c; and c replaces a) containing the element. For example:

Initial Ordering		Desired Ordering		Movements	
1	e	3	a	temp <- 1 (e)	temp <- 2 (d)
2	d	4	b	1 <- 3 (a)	2 <- 4 (b)
3	a	5	c	3 <- 5 (c)	4 <- temp
4	b	2	d	5 <- temp	
5	c	1	e		

All elements that have been moved into the correct location are flagged as having been moved by setting their order values to -1.

Source listings of the program and subroutine, named testss and sort_strings respectively, are given below.

```
 1  testss: procedure;
 2
 3       /* test caller for sort_strings */
 4
 5       declare
 6
 7            i fixed binary, sysprint file,
 8            sort_strings entry (character(256) varying dimension(*

 9            array (6) character(256) varying initial
```

```
     10                 (  "probe", "hello", "xray", "nice", "def", "abc"
\c);
     11
     12
     13        call sort_strings (array);
     14        do i = 1 to 6;
     15             put list (array (i));
     16             put skip;
     17        end;
     18
     19   end testss;


      1   sort_strings: procedure (strings);
      2
      3        declare
      4
      5             strings character(256) varying dimension(*),
      6             order fixed binary dimension (hbound (strings, 1)),
      7             temp character(256) varying,
      8             (i, k, l, t) fixed binary;
      9
     10
     11        /* initialize order array */
     12
     13        do i = 1 to hbound (order, 1);
     14             order (i) = i;
     15        end;
     16
     17        /* perform bubble sort */
     18
     19        k, l = hbound (strings, 1);
     20        do while (k <= l);
     21             do i = 2 to k;
     22                  l = i - 1;
     23                  if strings (order (l)) > strings (order (i)) then
\c do;
     24                     t = order (l); order (l) = order (i); order (i
\c) = t;
     25                     k = l;
     26                  end;
     27             end;
     28        end;
     29
     30        /* move strings into above ordering */
     31
     32        do i = 1 to hbound (strings, 1);
     33             if order (i) ^= -1 then do;
     34                  temp = strings (i);
     35
     36                  /* follow chain 'til reach start again */
     37
     38                  do k = i repeat l while (k ^= -1);
     39                       l = order (k);
     40                       strings (k) = strings (l);
```

```
41                     order (k) = -1;
42                end;
43                strings (l) = temp;
44           end;
45      end;
46
47
48  end sort_strings;
```

The debugging session begins below. Again, an exclamation point (!) indicates lines typed by the user.

```
 1        !  testss
 2   !(quit)
 3           QUIT
 4           r 736 6.068 0.132 9 level 2, 10
 5
 6        !  probe
 7           Condition quit raised at line 21 of sort_strings.
 8        !  source
 9                     do i = 2 to k;
10        !  value k
11               1
12        !  value l
13               1
```

First the program testss, used to test the sort_strings subroutine, is called from command level (line 1). When no output is produced, the program is aborted by issuing a quit signal, and probe is invoked to determine where the program was looping (line 6).

When probe is entered, it responds by giving the procedure and line where execution was interrupted. The source pointer is set by default to that line, so that the source request (line 8) may be used to display the text of the statement. The output does not indicate whether the infinite loop is occurring in the inner (do i = 2 to k) or outer (do while (k <= l)) loop. The value of k (line 11) is 1, which implies that the inner loop is not being entered; the value of l (line 13) is also 1 explaining why the outer loop never terminates.

An examination of the program shows that k and l could take on these values if elements 1 and 2 are exchanged on a pass with k = 2; on subsequent passes, no exchanges are made (as the inner loop is not entered), and the termination condition is never met. What is needed is to force l to be less than k on all passes unless an exchange actually occurs. This can be done by setting l = -1 before attempting the inner loop.

```
14          !  before: let l = -1
               Break set before line 21 of sort_strings.
16          !  quit
17             r 737 1.217 3.562 97 level 2, 10
18
19          !  start
20
21             def
22             hello
23             probe
24             abc
25             xray
26             r 737 0.359 0.182 0
```

The probe command can be used to modify the value of variables either interactively or as part of a break request list. In the latter case, the change is made every time the program is executed. A breakpoint is set before the current statement (line 21 of the program -- the inner loop) to set the value of l to -1 with the before request (line 14). The quit request (line 16) causes a return to command level, and the Multics start command (line 19) restarts the program from where it was interrupted. This time output is generated. However, the strings are not being sorted correctly.

```
27          !  probe sort_strings
28          !  position "i = 1";source
                    do i = 1 to hbound (order, 1);
30          !  position "i = 1";source
31                  do i = 1 to hbound (strings , 1);
32          !  before
33             Break set before line 32 of sort_strings.
34          !  quit
35             r 738 0.218 0.002 14
36
37          !  testss
38             Stopped before line 32 of sort_strings.
39          !  symbol order
40             fixed binary(17,0) aligned automatic dimension(6)
41             Declared in sort_strings.
42          !  value order(1:6)
43                 6
44                 5
45                 2
46                 4
47                 1
48                 3
```

One way to determine whether it is the sorting or ordering section of the program that is functioning incorrectly, is to stop the program before the ordering section and look at its input, the array

"order." The position request (line 28) is an attempt to locate the desired statement, but the source request (line 28), used to check that the correct line has been found, shows that the wrong one was found. The process is repeated (line 30), and the source pointer set to the correct line. A break is set (line 32) to cause the program to "halt" at that statement and enter probe. The driving program is begun once again (line 37), and sort_strings halts at the desired location. The symbol request (line 39) is used to check that the correct dimensions are being received for the array order. The value request (line 42) is used to display order(1), ..., order(6). It can be seen that these are the correct values ("abc", in position 6, is to be moved to position 1, etc.).

```
49        !  position 39; source
50                                     l = order (k)
51        !  after: (value k; value l)
52           Break set after line 39 of sort_strings.
53        !  continue
54                1
55                6
56                6
57                3
58                3
59                2
60                2
61                5
62                5
63                1
64                1
65                -1
66                4
67                4
68                4
69                -1
70           nice
71           def
72           hello
73           probe
74           abc
75           xray
76           r 740 0.602 0.000 0
```

It appears that the sorting code is working properly (with the patch in it). Therefore, the reordering of the array is failing for some other reason. The user then begins to trace the exchanges that are made. A break is set (lines 49 and 51) to display the values of k (the element to which the string is to be moved) and l (the element from which the string is to be moved) as the program is running. As stated previously, the effect of recompiling the program with a put statement added can be duplicated in this manner. The break is set after the line where both values have been determined for the

exchange. The continue request (line 53) restarts the program from where it was suspended by the break.

The output shows that extra exchanges are taking place. When k = 5, the next element on the chain is the first element (l = 1), and the fifth element should therefore be replaced by the copy of the first value stored in "temp." It should not be replaced by the current first element (the old element 6, "abc"). Nor should the program continue to move the undefined element -1 into element 1.

```
77      ! probe sort_strings
78      ! reset at 39
79        Break reset after line 39 of sort_strings.
80      ! before 39: if order(k) = i: (
81      !      let strings(k) = temp
82      !      let order(k) = -1
83      !      goto 42
84      ! )
85        Break set before line 39 of sort_strings.
86      ! quit
87        r 742 0.280 0.966 56
```

For the program to work properly, the movement through the chain must stop when the next element is the first (i.e., when order (k) = i). The saved value of the first (temp) should then be copied into the current element (strings(k)), and the search for additional unreordered elements continued. If the user were to recompile the program, the following code should achieve the desired effect.

```
if order (i) ^= -1 then do;
     temp = strings (i);
     do k = i repeat l while (order (k) ^= i);
           l = order (k);
           strings (k) = strings (l);
           order (k) = -1;
     end;
     strings (k) = temp;
     order (k) = -1;
end;
```

This approach may be checked before recompilation by making a slightly more elaborate patch than the one made previously. The probe command may be used to place a check for the correct terminating condition as the first thing in the loop on k and, if the condition is met, cause strings(k) to be set and the loop exited. First the break (containing the two value requests) previously set after the statement (line 78) is reset. Then a break, containing several requests and

extending across line boundaries, is set (lines 80 through 84) before the statement on line 39 of the program.

```
 88          ! testss
 89             Stopped before line 32 of sort_strings.
 90          ! reset
 91             Break reset before line 32 of sort_strings.
 92          ! continue
 93             abc
 94             def
 95             hello
 96             nice
 97             probe
 98             xray
 99             r 743 0.357 1.582 42
100
101          ! probe sort_strings
102          ! status
103             Break before line 39.
104             Break before line 21.
105          ! status at 21
106             Break before line 21:  let l = -1
107          ! quit
108             r 744 0.184 1.146 72
```

The program is run once again (line 88), and the break set between the two sections is encountered again. As it is no longer of any use, the reset request (line 90), assuming the default of the last break encountered, is used to delete the break. The continue request (line 92) resumes the execution of the program. This time it works!

The probe command is invoked once again. This time the status request is used to recall the breaks set, and, hence, the changes to be made to the program. Two forms of the status request are used. Just "status" (line 102) gives a list of all breaks set in the program; "status at line_number" (line 105) gives the text of the associated break request list. The user can now edit and recompile the program and expect it to work correctly. The remaining breaks need not be reset, because a recompilation has the same effect.

Terminology

active - a procedure is said to be active if its execution is ongoing or suspended by an error, quit signal, breakpoint, or call. An active procedure should be distinguished from one that has never been run, has completed execution, or has been interrupted and aborted by a Multics release command.

automatic storage - a storage class for which space is allocated dynamically in a stack frame upon block invocation. As a result, variables of this class only have storage assigned to them, and hence a legitimate address and value, when the block in which they are declared has an active invocation. PL/I variables, by default, belong to this class. FORTRAN variables must appear in an "automatic" statement in order to belong to this class.

block - corresponds to a PL/I procedure or begin block or FORTRAN program or subroutine, and identifies a particular group of variable declarations.

breakpoint - a point at which program execution is temporarily interrupted and probe requests executed.

invocation - when a procedure is called recursively, it will appear on the stack two or more times, and will have storage allocated for it the same number of times. Each instance of the procedure on the stack is considered a separate and distinguishable invocation of the block. The values of automatic variables can be different in different invocations of the same block. The most recent invocation is the topmost in stack trace.

level number - an integer used by probe to uniquely designate each block invocation (i.e., each entry in a stack trace). Level one is the first (least recent) procedure invoked. Level number is not necessarily the same as either of the numbers given after the word "level" in a ready message. The first of this pair gives the count of command levels in effect and gives the value n+1, where n is the number of programs (or groups of programs) whose execution has been suspended, the second gives the number of stack frames in existence and since the probe stack includes quick blocks, this number is less than or equal to the level number of the last command level in the stack trace.

quick block - internal procedures and begin blocks that satisfy certain requirements (e.g., are not called recursively, do not contain on, signal, or revert statements, etc.) have their automatic storage allocated by the blocks that call them. Hence, they do not actually have their own stack frames, but share the one of the caller. Certain system commands, such as trace_stack, ignore these blocks. The probe command, however, includes them in a stack trace, and treats them as if they were the same as any other blocks. The quickness of a block may be determined from a program listing containing information about the storage requirement of the program (produced with the -symbols, -map, or -list control arguments). For example, procedure "quick" shares stack frame of external procedure "main".

stack - if a procedure A calls another procedure B, then the execution of A is suspended until B returns. If B in turn calls C, then this is an ordered list of procedure or subroutine calls indicating which program called which other program, and which will return to which. This ordered list is called the "stack". In probe, a trace of the stack may be displayed by use of the stack request. The list is given in top-down fashion with the most recently called procedure listed first:

```
3    C
2    B
1    A
```

The numbers are level numbers.

stack frame - when a block is invoked (that is, a procedure is called or a begin block is entered), storage is allocated for its automatic variables. The area allocated is called a stack frame and logically corresponds to each entry in the stack.

static storage - a storage class for which space is allocated once per process, effectively at the time the procedure is first referenced. As a result, variables of this class always have a legitimate address and value. Regular FORTRAN variables, and those in a common block, have static storage. PL/I variables must be explicitly declared.

support procedure - a system utility routine that provides runtime support for other procedures (e.g., the procedure that allocates storage as requested by a PL/I allocate statement).

Summary of Requests

Request	Abbreviation	Description
after	a	Set a break after a statement.
before	b	Set a break before a statement.
call	cl	Call an external procedure.
continue	c	Return from probe.
execute	e	Execute a Multics command.
goto	g	Transfer to a statement.
halt	h	Stop the program.
if	(none)	Execute commands if condition is true.
let	l	Assign a value to a variable.
mode	(none)	Turn brief message mode on or off.
pause	pa	Stop a program once.
position	ps	Examine a specified statement or locate a string in the program.
quit	q	Return to command level.
reset	r	Delete one or more breaks.
source	sc	Display source statements.
stack	sk	Trace the stack.
status	st	Display information about breaks.
step	s	Advance one statement and halt.
symbol	sb	Display the attributes of a variable.
use	u	Examine the block specified.
value	v	Display the value of a variable.
where	wh	Display the value of probe pointers.
while	wl	Execute commands while condition is true.

de on or off.

pause	pa	Stop a program once.
position	ps	Examine a specified statement or locate a string in the program.
quit	q	Return to command level.
reset	r	Delete one or more breaks.
source	sc	Display source statements.
stack	sk	Trace the stack.
status	st	Display information about breaks.
step	s	Advance one statement and halt.
symbol	sb	Display the attributes of a variable.
use	u	Examine the block specified.
value	v	Display the value of a variable.
where	wh	Display the value of probe pointers.
while	wl	Execute commands while condition is true.

Name: profile

The profile command is a debugging tool used in conjunction with the -profile (-pf) control argument of the pl1, fortran, and cobol commands. The profile command prints information about the execution of each statement in the PL/I, COBOL, or FORTRAN program.

The -profile control argument causes the compiler to generate an internal static table containing an entry for each statement in the source program; the table entry contains information about the statement as well as a counter that starts out at zero. The counter associated with a statement is increased by one each time the statement is executed. The profile command prints and resets these counters.

Usage

```
profile paths {-control_args}
```

where:

1. paths
   are the pathnames or reference names of programs whose counters are to be printed or reset.

2. control_args
   are selected from the following list. Control arguments apply to all programs whose names appear in the command line.

   -print, -pr
   prints the following information for each statement in the specified programs:

   1. line number

   2. statement number, if greater than 1

   3. number of times the statement has been executed

   4. cost of executing the statement measured in number of instructions executed online plus the number of PL/I operators invoked. Each instruction and each operator invocation count as only one unit.

   5. the names of all the PL/I operators used by this statement

   6. total cost for all statements is printed at the end

-brief, -bf
    omits from the statement list statements that have never been executed.

-long, -lg
    includes in the statement list statements that have never been executed.

-reset, -rs
    causes profile to reset to zero all counters associated with the specified program.

Note

If no control arguments are given, the default control arguments are -print and -brief.

Example

The PL/I program shown below counts the number of occurrences of one string in another string. It was compiled with the -profile control argument and executed once. Notice that line number and statement number (LINE and ST, respectively) of the statement in the then clause is the same as the line number and statement number of the if statement itself.

The source code for the program is:

```
 1 example:  proc(s1,s2);
 2
 3 declare   (s1,s2) char(*),
 4           (i,k) fixed bin,
 5           ioa_ options (variable);
 6
 7           k = 0;
 8           do i = 1 to length(s1) - length(s2);
 9                if substr(s1,i,length(s2)) = s2
10                then k = k + 1;
11           end;
12
13           call ioa_("^d",k);
14           end example;
```

After executing the program once and invoking the profile command without any control arguments, the output is:

LINE ST	COUNT	COST	PROGRAM
			example
7	1	1	
8	1	5	
8	8	24	
9	7	56	
9	1	1	
11	7	14	
13	1	13+1	(call_ext_out_desc)
14	1	0+1	(return)
TOTAL		114+2	

Name: reset_external_variables

The reset_external_variables command reinitializes system-managed variables to the values they had when they were allocated.

Usage

```
reset_external_variables names {-control_arg}
```

where:

1. names
   are the names of the external variables, separated by spaces, to be reinitialized.

2. control_arg
   is -unlabeled_common (or -uc) to indicate unlabeled (or block) common.

   Note

   A variable cannot be reset if the segment containing the initialization information is terminated after the variable is allocated.

reslve linkage error, rle

Name : resolve_linkage_error, rle

The resolve_linkage_error command is invoked to satisfy the linkage fault after a process encounters a linkage error. The program locates the virtual entry specified as an argument and patches the linkage information of the process so that when the start command is issued the process continues as if the original linkage fault had located the specified virtual entry.

Usage

```
resolve_linkage_error  virtual_entry
```

where virtual_entry is a virtual entry specifier.

Notes

For an explanation of virtual entries, see the description of the cv_entry_ subroutine.

Examples

```
!    myprog
     Error:  Linkage error by >udd>m>vv>myprog|123
     referencing subroutine$entry
     Segment not found.
     r 1234 2.834 123.673 980 level 2, 26

!    rle mysub$mysub_entry
     r 1234 0.802 23.441 75 level 2, 26

!    start
     ... myprog is running
```

Name: run_cobol, rc

The run_cobol command explicitly initiates execution of a COBOL run unit in a specified "main program". This command is not needed to execute COBOL object programs on Multics; it is used to simulate an environment in which traditional COBOL concepts may be easily defined. This command cannot be called recursively.

Usage

```
run_cobol name {-control_args}
```

1. name
   is the reference name or pathname of the "main program" in which execution is to be initiated. If a pathname is given, then the specified segment is initiated with a reference name identical to the entryname portion of the pathname. Otherwise, the search rules are used to locate the segment. If the name specified in the PROG-ID statement of the COBOL program (i.e., the entry point name) is different from the current reference name of the object segment, then the name specified here must be in the form A$B where A is the pathname or reference name of the segment and B is the PROG-ID as defined in the IDENTIFICATION DIVISION of the source program.

2. control_args
   can be chosen from the following:

   -cobol_switch N, -cs N
   sets one or more of the eight COBOL-defined "external switches" on, where N is a number from 1 to 8 (or a series of numbers separated by spaces) that corresponds to the numbered external switch. At the outset of the run unit, the default setting of these external switches is off. (The eight external switches are defined in the Multics COBOL Reference Manual, Order No.!AS44.)

   -no_stop_run, -nsr
   avoids establishment of a handler for the stop_run condition. (See "Notes" below.)

   -sort_dir path, -sd path
   specifies the directory to be used during execution of this run unit for temporary sort work files. If this control argument is not specified, the process directory is assumed.

   -sort_file_size N, -sfs N
   is the floating point representation of the estimated average size in characters of the files to be sorted

during execution of this run unit. This information is used to optimize sorting. If not specified 1e6 is assumed (i.e., one million characters).

Notes

This command enables the user to explicitly define and start execution of a COBOL run unit. A run unit is either explicitly started by the execution of the run_cobol command or implicitly started by the execution of a COBOL object program either by invocation from command level or from a call by another program written in COBOL or another language. A run unit is stopped either by the execution of the STOP RUN statement in a COBOL object program or by invocation of the stop_cobol_run command. For the duration of time after a run unit is started and before it is stopped, it is said to be active. All COBOL programs executed while a run unit is active are considered part of that run unit.

A run unit is a subset of a Multics process; it is stopped when the process is ended. Also, when all programs contained in a run unit are cancelled, the run unit is stopped (refer to the cancel_cobol_program command). Only one run unit may be active at any given time in a process; thus, the run_cobol command cannot be invoked recursively. Additionally, if a run unit has been started implicitly (as described above), the run_cobol command may not be used until that run unit has been stopped; i.e., the run_cobol command does not terminate a currently active run unit.

The explicit creation of a run unit with the run_cobol command performs the following functions:

1. Establishment of a "main program", i.e., a program from which control does not return to the caller. The EXIT PROGRAM statements, when encountered in such a program, have no effect, as required in the COBOL definition. An implicitly started run unit has no "main program". The EXIT PROGRAM statement in all programs contained in such a run unit always causes control to be returned to the caller, even if the caller is a system program, e.g., the command processor.

2. Setting of the COBOL external switches. These switches are set to off unless otherwise specified by the -cobol_switch control argument.

3. User control of the action taken when a STOP RUN statement is executed in a COBOL object program. The action normally taken for STOP RUN is cancellation of all programs in the run unit, closing any files left open. After this has been

done, the data associated with any of the programs is no longer available. Thus in a debugging environment, it may be useful to redefine the action taken for STOP RUN. When the run unit is explicitly initiated with the run_cobol command, the STOP RUN statement causes the signalling of the stop_run condition for which a handler is established that performs the normal action described above. If the -no_stop_run control argument is specified, the handler is not established, thus allowing the user to handle the signal himself using other Multics commands. If the user has not provided a handler himself for stop_run and specifies the -no_stop_run control argument, an unclaimed signal results.

The name given in the run_cobol command need not be a COBOL object program. It may be a program produced by any language compiler that provides a meaningful interface with COBOL programs (e.g., PL/I, FORTRAN).

Refer to the following related commands:

display_cobol_run_unit, dcr
stop_cobol_run, scr
cancel_cobol_program, ccp

Name: set_fortran_common, sfc

The set_fortran_common command allocates and initializes all FORTRAN common blocks referenced by the specified FORTRAN object segments. The maximum declared length of a common block (of all those found in the list of FORTRAN object segments) is used for the allocation and initialization. This command can therefore be used to guarantee that the correct common block storage is allocated and initialized prior to a FORTRAN run. (If the user left it to the dynamic linker, the first reference to the common block would cause it to be allocated and initialized as declared in the referencing program. This program might not include the necessary initialization information.) The set_fortran_common command can also be used to reinitialize the common blocks referenced by the specified object segments, although it will not reinitialize any local storage such as static or automatic variables.

Usage

```
set_fortran_common paths {-control_arg}
```

where:

1. paths
   are the pathnames of the FORTRAN object segments whose common blocks are to be allocated and (re)initialized.

2. control_arg
   can be -long (-lg) indicating that warning messages are to be printed. Normally, all warning messages are suppressed. Warnings are printed if the common block is already allocated with a smaller size.

Notes

A FORTRAN object segment is either a segment created by one of the Multics FORTRAN compilers or is a segment created by the binder and contains at least one component that was created by one of the Multics FORTRAN compilers.

Only common storage is affected by this command. Local variables are not (re)initialized.

Common blocks without data initialization information are set to binary zeros.

If the common block is already allocated, its contents are reinitialized and the prior contents are lost.

A warning is always printed if different initialization values are encountered in the set of specified object segments.

Name: set_system_storage

The set_system_storage command establishes an area as the storage region in which normal system allocations are performed.

Usage

```
set_system_storage {virtual_ptr -control_arg}
```

where:

1. virtual_ptr
   is a virtual pointer to an initialized area. The syntax of virtual pointers is described in the cv_ptr_ subroutine description. This argument must be specified only if the -system control argument is not supplied.

2. control_arg
   is -system to specify the area used for linkage sections. This control argument must be specified only if virtual_ptr is not specified.

Notes

To initialize or create an area, refer to the description of the create_area command.

The area must be set up as either zero_on_free or zero_on_alloc.

It is recommended that the area specified be extensible.

Examples

The command line:

```
set_system_storage free_$free_
```

places objects in the segment whose reference name is free_ at the offset whose entry point name is free_.

The command line:

```
set_system_storage my_seg$
```

uses the segment whose reference name is my_seg. The area is assumed to be at an offset of 0 in the segment. The segment must already exist with the reference name my_seg and must be initialized as an area.

The command line:

```
set_system_storage my_seg
```

uses the segment whose (relative) pathname is my_seg. The segment must already exist.

Name: set_user_storage

The set_user_storage command establishes an area as the storage region in which normal user allocations are performed. These allocations include FORTRAN common blocks and PL/I external variables whose names do not contain dollar signs.

Usage

set_user_storage {virtual_ptr -control_arg}

where:

1. virtual_ptr
   is a virtual pointer to an initialized area. The syntax of virtual pointers is described in the cv_ptr_ subroutine description. This argument must be specified only if the -system control argument is not specified.

2. control_arg
   is -system to specify the area used for linkage sections. This control argument must be specified only if virtual_ptr is not specified.

Notes

To initialize or create an area, refer to the description of the create_area command.

The area must be set up as either zero_on_free or zero_on_alloc.

It is recommended that the area specified be extensible.

Examples

The command line:

set_user_storage free_$free_

places objects in the segment whose reference name is free_ at the offset whose entry point name is free_.

The command line:

set_user_storage my_seg$

uses the segment whose reference name is my_seg. The area is assumed to be at an offset of 0 in the segment. The segment must already exist with the reference name my_seg and must be initialized as an area.

The command line:

set_user_storage my_seg

uses the segment whose (relative) pathname is my_seg. The segment must already exist.

Name: stop_cobol_run, scr

The stop_cobol_run command causes the termination of the current COBOL run unit. Refer to the run_cobol command for information concerning the run unit and the COBOL runtime environment.

Usage

```
stop_cobol_run {-control_arg}
```

where the control_arg may be -retain_data or -retd to leave the data segments associated with the programs composing the run unit intact for debugging purposes. (See "Notes" below.)

Notes

The results of the stop_cobol_run command and the execution of the STOP RUN statement from within a COBOL program are identical. Stopping the run unit consists of cleaning up all files that have been opened during the execution of the current run unit, and ensuring that the next time a program that was a component of this run unit is invoked, its data is in its initial state.

To maintain the value of all data referenced in the run unit in its last used state, the -retain_data control argument should be used.

Refer to the related commands:

display_cobol_run_unit, dcr
cancel_cobol_program, ccp
run_cobol, rc

Name: trace

The trace command is a debugging tool that lets the user monitor all calls to a specified set of external procedures. The trace command modifies the standard Multics procedure call mechanism so that whenever control enters or leaves one of the procedures specified by the user, a debugging procedure is invoked. The user can request the following:

1. Print the arguments at entry, exit, or both.

2. Stop (by calling the command processor) at entry, exit, or both.

3. Change the frequency with which tracing messages are printed (e.g., every 100 calls, after the 2000th call, only if the recursion depth is less than five, etc.).

4. Execute a Multics command line at entry, exit, or both.

5. Meter the time spent in the various procedures being monitored.

Use of the trace command is subject to the following restrictions:

1. Only external procedures compiled by PL/I or FORTRAN can be traced.

2. Ring 0 or gate entries cannot be traced.

3. Incorrect execution results if the traced procedure looks back a fixed number of stack frames, e.g., cu_$arg_ptr cannot be traced.

4. Only 100 procedures can be traced at one time. Up to 16 locations can be watched at one time.

5. The procedure being traced and the trace package itself must share the same combined linkage segment.

6. A procedure in a bound segment can only be traced if its entry point is externally available.

Usage

```
trace {-control_args} names
```

where:

1. names
   is a pathname or reference name. The reference name or entry portion of a pathname is used in the trace table. (See "Notes" below.)

2. control_args
   apply to the namei arguments that follow, and, if applicable, change the current value in the trace control template (TCT). (See "Notes" below.) Control arguments may be chosen from the following:

   -after N
      calls the command processor after calling the traced procedure every N times (initial value!=!0: do not call).

   -argument N, -ag N
      prints the arguments every Nth time the procedure is entered (initial value!=!0: do not print).

   -before N
      calls the command processor before calling the traced procedure every N times (initial value!=!0: do not call).

   -brief, -bf
      prints a short form of the monitoring information.

   -depth N, -dh N
      monitors to the maximum recursion depth of N (initial value!=!0: no limit).

   -every N, -ev N
      monitors every Nth call (initial value!=!1).

   -execute STR, -ex STR
      executes the Multics command line specified by the string STR whenever the procedure is monitored (initial value!=!"": no command).

   -first N, -ft N
      starts monitoring on the Nth call (initial value!=!1).

   -govern STR, -gv STR
      limits/does not limit the recursion level for a procedure, where STR can be the string on or off (initial value!=!off). See "Recursion Limiting" below.

   -in
      prints the arguments only on entry (initial value!=!yes).

-inout
    prints the arguments on both entry and exit (initial value!=!no).

-io_switch STR, -is STR
    changes the switch for output to the switch specified by STR. (See "Changing Output Switch" below.)

-last N, -lt N
    stops monitoring after the Nth call (initial value!=!9999999999).

-long, -lg
    prints the long form of the monitoring information. (For use after the -brief control argument to restore the long form.)

-meter STR, -mt STR
    meters/does not meter the time spent in the procedure, where STR can be the string on or off (initial value!=!off). See "Metering" below.

-out
    prints the arguments only on exit (initial value!=!no).

-off entryname
    stops monitoring the specified procedure. The procedure remains in the trace table and calls continue to be counted.

-on entryname
    resumes monitoring the specified procedure. This control argument is used after the -off control argument.

-remove entryname, -rm entryname
    removes the specified procedure from the trace table. Tracing can be removed at any time.

-reset entryname, -rs entryname
    sets the number of calls and recursion depth of the specified procedure to zero.

-return_value STR, -rv STR
    prints/does not print the return value on exit, where STR can be the string on or off (initial value!=!off). This control argument assumes the entry is a function.

-status *, -st *
    prints the procedures being monitored and their counters. (See "Notes" below.)

-status entryname, -st entryname
    prints the trace parameters and counters for the procedure specified by entryname. (See "Notes" below.)

-stop_proc path, -sp path
    changes the procedure that is called for stop requests from the command processor to the procedure specified by path. To reset the stop procedure, issue this control argument with no path argument.

-subtotal, -stt
    prints and does not clear the metering statistics.

-template, -tp
    lists the trace control template.

Notes

The procedure whose pathname is given in the command line is added to the trace table with the tracing parameters from the trace control template (TCT). If the procedure is already in the table, the counters are reset and the current parameters in TCT are used.

For control arguments that affect procedures being traced, the argument is an entryname or an asterisk (*). If an entryname is used, the control argument applies to that procedure. If an asterisk is used, the control argument is applied to all entries in the trace table. All control arguments that affect the TCT must have a number argument (indicated by N above).

Examples

The command line:

```
trace -ag 1 -inout test
```

prints the arguments for test on entry and exit.

The command line:

```
trace -ag 2 -in -depth 6 test
```

prints the arguments for test every second time test is entered up to a recursion depth of six, i.e., 2, 4, 6.

The command line:

```
trace -govern on test
```

prints the arguments of test each time test is called with a new maximum recursion depth. The trace procedure calls the command processor every time the recursion depth is a multiple of 10.

The command line:

```
trace -st * -tp
```

lists the procedures in the trace table and prints the values of the trace control template.

Message Format

The message printed when control enters a procedure can appear in any one of several formats, depending on the setting of the brief switch and the status of the calling procedure. If the calling procedure is unbound or occurs in a bound segment containing a bindmap, the message takes the form:

```
Call 4.1 of alpha from beta|127, ap = 204|10746.
```

This is the fourth call of procedure alpha, which is at recursion level 1. The call comes from location 127 in component beta, and the argument list is at 204|10746. If the procedure making the call is in a bound segment that does not contain a bindmap, the message takes the form:

```
Call 4.1 of alpha from bound_gamma|437 (beta), ap = 204|10746.
```

The name in parentheses may not always be available and may be omitted in some cases. If the user has requested the brief output mode, the message is shortened to:

```
Call 4.1 of alpha.
```

When tracing is requested for a procedure, the parameters for that entry are taken from the trace control template (TCT). If the user does not alter the values in the TCT, the initial default values are used (see below). The initial values in the TCT specify that every call should be monitored.

Trace Control Template

As mentioned earlier, the trace table entry holds a number of parameters for each procedure to be traced. The values of the parameters are determined by the contents of the TCT at the time the table entry is filled in. These parameters are used in conjunction with N (the number of calls to the traced procedure in this process) and R (the current recursion depth) to control when and how the procedure should be monitored. The execution count (N) is set to 0 when tracing is first started and is incremented by 1 every time the traced procedure is called. The recursion depth (R) is set to 0 when tracing is first started and is incremented by 1 every time control enters the traced procedure and is decremented by 1 every time control leaves the traced procedure.

Let:

- D = the maximum recursion depth to be monitored (-depth)
- F = the number of the first call to be monitored (-first)
- L = the number of the last call to be monitored (-last)
- E = how often monitoring should occur (-every)
- B = the number of times the procedure is called before trace stops at entry to the traced procedure (-before)
- A = the number of times the procedure is called before trace stops at exit from the traced procedure (-after)
- AG = the number of times the procedure is called before trace prints the arguments of the traced procedure (-argument)
- I = a bit that is "1"b if the tracing procedure should print the arguments of the traced procedure when control goes into the traced procedure (-in)
- O = a bit that is "1"b if the tracing procedure should print the arguments of the traced procedure when control goes out of the traced procedure (-out)

A call is monitored and the tracing procedure is called if, and only if:

```
F <= N <= L
R <= D
mod(N,E) = 0
```

If AG ^= 0, mod(N,abs(AG)) = 0, and I = "1"b, trace prints the values of the arguments (if any) being passed to the traced procedure. All of the arguments are listed when AG < 0. If AG < 0, the procedure is assumed to be a function and the value of the last argument is printed after the procedure returns.

If B ^= 0 and mod(N,B) = 0, the monitoring procedure prints "Stop" and calls the command processor (or a user-set procedure if the -stop_proc control argument was used). This call occurs before the procedure being traced has created its stack frame.

After control leaves the traced procedure, trace prints a line of the form:

Return N.R from alpha.

If AG ^= 0 and mod(N,abs(AG)) = 0, then all of the arguments of the traced procedure are printed if O = "1"b; otherwise, if AG < 0, the value of the last argument (assumed to be the value of the function) is printed.

Finally, trace calls the command processor. If the -stop_proc control argument was given, a procedure set by the user is called. This call occurs after the stack frame of the procedure being traced has been destroyed.

Metering

The trace command can be used to meter the execution of a specified set of procedures. If the metering feature is being used, trace does not call the debugging procedure when control enters a procedure being traced; instead, it determines the current time and the virtual CPU time used, and the number of page faults taken by the user's process before control enters and after control leaves the traced procedure. This information is used to compute the real time and CPU time used, and the number of page faults taken by the traced procedure on a local and global basis. The global CPU time is the time spent in the procedure including the time spent in any procedures that it calls. The local CPU time does not include the time spent in any traced procedure called by the procedure, but it does include time spent in called procedures that are not being traced. The local and global versions of real time and page faults are calculated in a similar manner. Metering is only done when the first, last, every, and depth tracing conditions are satisfied.

The control argument:

-meter on, -mt on

sets the metering switch in the TCT; any procedures added to the trace table or that have their table entries updated after this argument is used are metered.

The control argument:

-meter off, -mt off

turns off the metering switch in the TCT; any procedures currently being metered continue to be metered.

The control argument:

-total

causes trace to print the metering statistics of all procedures in the trace table. The output gives the number of calls (#CALLS), global CPU time (GCPU), global real time (GREAL), global page waits (GPWS), local CPU time (LCPU), local real time (LREAL), local page waits (LPWS), and the usage percentage (%USAGE) based on local CPU time, of all the procedures being metered. The metering statistics are set to 0 after they are printed.

The control argument:

-subtotal, -stt

prints the same information as the -total control argument, but does not clear the statistics.

Recursion Limiting

The control argument:

-govern on, -gv on

sets a bit in the TCT that causes recursion limiting to be in effect for any procedure subsequently added to the trace table. When the governing feature is used, the depth control parameter is ignored and trace prints the call message only when the recursion depth of the traced procedure reaches a new, maximum depth. Each call message has a recursion depth one greater than the previous call message. In addition, trace calls the command processor (or a user-defined procedure if the -stop_proc control argument was used) whenever the recursion depth is a multiple of 10. Return messages are not printed. This feature enables the user to find and limit uncontrolled recursion; it can be very useful in finding the procedure(s) responsible for fatal process error.

The control argument:

-govern off, -gv off

turns off the governing switch in the TCT; any procedure currently being governed continues to be governed.

Watch Facility

The trace command has an optional watch facility in which trace watches the contents of a set of previously specified memory cells. The cells are checked at every entry to and every exit from every traced procedure. As long as the values in the locations being watched remain the same, no action is taken and no tracing messages are printed. The tracing message is printed as soon as trace finds that any of the locations being watched has had its value changed. This can be found either at entry to or exit from the traced procedure. When any value changes, the tracing message is preceded by lines that give the new values of all of the locations that have changed, and the command processor (or a user-set procedure if the -stop_proc control argument was used) is called even if the A or B conditions are not met. When execution continues, the locations that have changed are watched with the new value being used in subsequent checks. This feature can be very useful in determining which of the user's procedures has incorrectly modified a word of storage.

The control argument:

-watch STR, -wt STR

causes all procedures being traced to watch for a change in the current contents of the memory word(s) specified by the string STR. This string, specifying the location, can consist of a single address specification or a series of address specifications separated by blanks and surrounded by quotes. If an address specification does not contain a vertical bar (|), it is taken to be an octal number giving a location in the stack; otherwise, it is taken to be a segment number and offset in octal in the standard form, e.g., segment_number|offset.

The control argument:

-watch off, -wt off

turns off the watch facility.

The watch facility differs from other trace facilities in that there is a single table of locations being watched that is used by all procedures being traced. When the -watch control argument is processed, the new location(s) specified replace any locations currently in the watch table. There is no provision made for removing a single location from the watch table; the user must reissue a watch request that omits the location to be removed from the table.

Command Execution

The command execution facility of trace allows the user to specify a Multics command line to be executed whenever the trace debugging procedure is called. The trace procedure calls the command processor with the specified string after printing the tracing message, but before the stop request causes the command processor to be called.

The control argument:

-execute string

sets the execution string parameter in the TCT. Since string is a single argument, it must be enclosed in quotes if it contains any spaces. The execution parameter in the TCT is turned off if string has zero length (-execute ""). The following line:

trace -ex time test

causes trace to execute the time command before and after test is called.

Changing Output Switch

All of the messages from the trace command that may be generated while actually monitoring procedures are normally written on the user_i/o switch so that trace can conveniently be used with procedures that change the attachment of the normal switch, user_output. The control argument:

-io_switch STR

causes trace to write further monitoring output on the switch specified by STR, which must already be attached and opened for stream_output.

Name: trace_stack, ts

The trace_stack command prints a detailed explanation of the current process stack history in reverse order (most recent frame first). For each stack frame, all available information about the procedure that established the frame (including, if possible, the source statement last executed), the arguments to that (the owning) procedure, and the condition handlers established in the frame are printed. For a description of stack frames, see "Multics Stack Segments" in Section!IV of the MPM Subsystem Writers' Guide.

The trace_stack command is most useful after a fault or other error condition. If the command is invoked after such an error, the machine registers at the time of the fault are also printed, as well as an explanation of the fault. The source line in which it occurred can be given if the object segment is compiled with the -table option.

Usage

trace_stack {-control_args}

where control_args can be selected from the following:

-brief, -bf
: suppresses listing of arguments and handlers. This control argument cannot be specified if -long is also specified as a control argument.

-long, -lg
: prints octal dump of each stack frame.

-depth N, -dh N
: dumps only N frames.

Output Format

When trace_stack is invoked, it first searches backward through the stack for a stack frame containing saved machine conditions as the result of a signalled condition. If such a frame is found, tracing proceeds backward from that point; otherwise, a comment is printed and tracing begins with the stack frame preceding trace_stack.

If a machine-conditions frame is found, trace_stack repeats the system error message describing the fault. Unless the -brief control argument is specified, trace_stack also prints the source line and faulting

instruction and a listing of the machine registers at the time the error occurred.

The command then performs a backward trace of the stack, for N frames if the -depth N argument was specified, or else until the beginning of the stack is reached.

For each stack frame, trace_stack prints the offset of the frame, the condition name if an error occurred in the frame, and the identification of the procedure that established the frame. If the procedure is a component of a bound segment, the bound segment name and the offset of the procedure within the bound segment are also printed.

The trace_stack command then attempts to locate and print the source line associated with the last instruction executed in the procedure that owns the frame (that is, either a call forward or a line that encountered an error). The source line can be printed only if the procedure has a symbol table (that is, if it was compiled with the -table option) and if the source for the procedure is available in the user's working directory. If the source line cannot be printed, trace_stack prints a comment explaining why.

Next, trace_stack prints the machine instruction last executed by the procedure that owns the current frame. If the machine instruction is a call to a PL/I operator, trace_stack also prints the name of the operator. If the instruction is a procedure call, trace_stack suppresses the octal printout of the machine instruction and prints the name of the procedure being called.

Unless the -brief control argument is specified, trace_stack lists the arguments supplied to the procedure that owns the current frame and also lists any enabled condition, default, and clean-up handlers established in the frame.

If the -long control argument is specified, trace_stack then prints an octal dump of the stack frame, with eight words per line.

Example

After a fault that reenters the user environment and reaches command level, the user invokes the trace_stack command.

For example, after quitting out of the list command, the following process history might appear:

```
        !  list

    Segments=8, Records=3

    rew  0  mailbox
    r w
    QUIT

 !  trace_stack
    quit in ipc_$block|156
    (>system_library_1>bound_command_loop_|156)
      No symbol table for ipc_
       156   400010116100     cmpq    pr4|10

    Machine registers at time of fault

    pr0 (ap)  263|4656           pl1_operators_$operator_table|162
                                   (external symbol in separate nonstand
card
                                   text section)
    pr1 (ab)  103|264            scs|264
    pr2 (bp)  14|12200           as_linkage|12200
    pr3 (bb)  113|0              tc_data|0
    pr4 (lp)  253|2250           !BBBJGjFkPBWcNZ.area.linker|2250
                                   (internal static|0 for ipc_)
    pr5 (lb)  244|3614           stack_4|3614
    pr6 (sp)  244|3500           stack_4|3500  ( -> "kcpMbLH +0000000")
    pr7 (sb)  244|0              stack_4|0

    x0      73   x1       0   x2       0   x3 600000
    x4       0   x5      32   x6    3033   x7      4
    a 000000000000  q 000000000004  e 0
    Timer reg - 1746005, Ring alarm reg - 0

    SCU Data:

      4030    400270250011 000000000021 400270000000 000000672000
              000156000200 000154000700 002250370000 600044370120

    Connect Fault (21)
    At: 270|156 ipc_|156  (bound_command_loop_|156)
    On: cpu a (#0)
    Indicators:  ^bar
```

```
.

APU Status:  xsf, sd-on, pt-on, fabs
CU Status:   rfi, its, fif
Instructions:
  4036    002250 3700 00      epp4      2250
  4037   6 00044 3701 20      epp4      pr6|44,*

Time stored: 08/02/77  1635.5 edt Tue (104541674361226602)
Ring:          4

Backward trace of stack from 244|3500


3500 quit ipc_$block|156 (bound_command_loop_|156)
  No symbol table for ipc_
   156   400010116100      cmpq    pr4|10
         ARG  1: 253|5704 !BBBJGjFkPBWcNZ.area.linker|5704
         ARG  2: 244|3152 stack_4|3152
         ARG  3: 0


2720      tty_$tty_get_line|2442 (bound_iox_|11546)
  No symbol table for tty_
  call_ext_out to ipc_$block
         ARG  1: 253|4320 !BBBJGjFkPBWcNZ.area.linker|4320
                            (internal static|154 for find_iocb)
         ARG  2: 244|2660 stack_4|2660 ( -> "fo stuff")
         ARG  3: 128
         ARG  4: 0
         ARG  5: 0


2400      listen_$listen_|461 (bound_command_loop_|1325)
  No symbol table for listen_
  call_ext_out to iox_$get_line
         ARG  1: ""
         on "cleanup"  call listen_|256 (bound_command_loop_|1122)


2100      process_overseer_$process_overseer_|473 (bound_command_loop
\c_|21433)
  No symbol table for process_overseer_
  call_ext_out_desc to listen_$listen_
         Argument list header invalid.
         on "any_other"
           call standard_default_handler_$standard_default_handler_3
           (external symbol in separate nonstandard text section)


2000      user_init_admin_$user_init_admin_|36 (bound_command_loop_|2
\c1676)
  No symbol table for user_init_admin_
 21676   700036670120     tsp4     pr7|36,* alm_call
         No arguments.

End of trace.
```

trace stack, ts

r 1635 1.756 40.790 207 level 2, 9

APPENDIX W

Workshops W-1
  Workshop One. W-1
  Workshop Two. W-10
  Workshop Three. W-13

WORKSHOP ONE

A probe Workshop

The best (perhaps the only) way of learning how to use the probe command is by using the command in actual debugging sessions. This workshop provides the experience of debugging a moderately complicated program. The program computes and prints out the elements of a Fibonacci series. An F series begins as

0 1 1 2 3 5 8 13 21 34 55 ...

An element of the series is calculated by adding the previous 2 elements (for a Fibonacci series of degree 2). In the series shown above, the first two elements (0 and 1) are given as initial values and the remaining elements are then computed.

Fibonacci series of higher degrees can also be defined by adding more elements to calculate the next in the series. For example, a series of degree 4 begins as

0 0 0 1 1 2 4 8 15 29 56 108 ...

with the next element of the series calculated by adding the previous 4 elements in the series.

The program shown below reads two parameters from the terminal: Fdeg gives the number of the highest degree Fibonacci series to be computed; count gives the number of elements to be included in each series. For input of

Fdeg=4, count=7;

the first seven elements (excluding the assumed initial values) of the Fibonacci series of degrees 2, 3, and 4 are printed.

Now, without further ado, here are the programs! There is one written in PL/I, and one written in FORTRAN. You can copy whichever of these programs you wish to debug from >udd>F19>Student_01>fib.pl1 (or .fortran) into your home directory. Note that the line numbers shown below are not actually a part of the source segment.

>udd>f50ed>wksps>F21>s1>fib.pl1

WORKSHOP ONE

```
 1   fib: proc;
 2
 3   dcl (sysin, sysprint) file,
 4        Sfirst bit(1) int static init("1"b),
 5       (Fdeg, count, i) fixed bin,
 6        msg char(256) varying;
 7
 8   dcl  linesize fixed bin,
 9        get_line_length_$stream entry (char(*), fixed bin(35))
10             returns(fixed bin);
11
12   dcl  cleanup condition;
13
14             /* Establish cleanup on unit to close files.      */
15             /* Open input/output files, get output file line  */
16             /* length to insure output lines fit on terminal. */
17
18             on cleanup close file(sysin), file(sysprint);
19             open file(sysin) stream input;
20             open file(sysprint) stream output;
21             linesize = get_line_length_$stream ("sysprint", 0);
22
23             /* Initialize indicator of how many series should */
24             /* be output (Fdeg) and how many items should be  */
25             /* output in each series (excluding assumed       */
26             /* initial elements of each series).              */
27
28             Fdeg = 2;
29             count = 10;
30
31
32             /* Output brief instructions to the user, but     */
33             /* only the first time fib invoked in each process*/
34
35             if Sfirst then do;
36                  msg =
37   "Enter Fdeg and/or count, followed by a "";"" character.";
38                  write file(sysprint) from(msg);
39                  msg =
40   "For example,
41             Fdeg = 2, count=10;";
42                  write file(sysprint) from(msg);
43                  msg =
44   "These are the default values.  To stop, enter
45             Fdeg = 1;
46   ";
47                  write file(sysprint) from(msg);
48                  Sfirst = "0"b;
49             end;
50             put file(sysprint)
51                  list ("Enter data, or just a "";"" char:  ");
52             get file(sysin) data (Fdeg, count);
53
```

```
54          /* Compute and output each Fibonacci series.          */
55          /* Then get next set of input values.                 */
56
57          do while (Fdeg < 1);
58              put file(sysprint) skip(2) data (count);
59              put file(sysprint) skip;
60              do i = 2 to Fdeg;
61                  call gen_fib (i, count);
62              end;
63              put file(sysprint)
64                  list ("New data, or just a "";"" char:  ");
65              get file(sysin) data (Fdeg, count);
66          end;
67
68
69          /* Close files and return.                            */
70
71          close file(sysin), file(sysprint);
72          return;
73
```

I simple
I difficult

WORKSHOP ONE

```
 74   gen_fib: proc (grouping, count);
 75
 76   dcl (grouping, count) fixed bin,
 77             /* Fibonacci series to be computed, and number    */
 78             /* of items to be computed in the series.          */
 79        N (grouping) fixed bin(71),
 80             /* Array of values summed to form series elements*/
 81        result (-grouping:count-1) char(28) varying,
 82             /* Array of output values, including assumed       */
 83             /* values which begin the series.                  */
 84        r_matrix (Nrows, Ncols) char(28) varying based(Pr_matrix),
 85             /* 2-dimensional overlay for the computed output   */
 86             /* values (excluding assumed values).              */
 87        Pr_matrix ptr;
 88
 89   dcl (Icol, Irow, Ncols, Nrows,
 90        Nused_cols_in_last_row) fixed bin,
 91        Sdoes_not_fit bit(1),
 92        SPACES char(30) int static options(constant) init(""),
 93        cycle fixed bin,
 94             /* index of series element being computed.         */
 95        formatted_total pic "zzz,zzz,zzz,zzz,zzz,zzz,zz9",
 96        output_total char(100) varying,
 97        total fixed bin(71);
 98
 99
100             /* Initialize the assumed values which begin       */
101             /* the series.  All are 0 but the last, which is 1*/
102
103             N(*) = 0;
104             N(grouping-1) = 1;
105
106
107             /* Put the assumed values in the output array.     */
108
109             do cycle = -grouping to -2;
110                  result(cycle) = "0";
111             end;
112             result(cycle) = "1";
113
114
115             /* Compute remaining values of series, and put in */
116             /* the output array.                               */
117
118             do cycle = 0 to count-1;
119                  total = sum(N);
120                  formatted_total = total;
121                  result(cycle) = ltrim(formatted_total);
122                  N(mod(cycle, grouping)) = total;
123             end;
124
```

```
125             /* The output will be printed with assumed values */
126             /* preceding computed values.  The computed values*/
127             /* will be printed in as many rows as possible to */
128             /* reduce the number of output lines.  However, if*/
129             /* the output fits in 2 or more rows, the number  */
130             /* of rows is chosen so that all columns but the  */
131             /* final one are full (have Nrows values).        */
132             /* Of course, in multi-column format, all data    */
133             /* must fit the terminal linesize.                */
134
135             Pr_matrix = addr(result(0));
136             Sdoes_not_fit = "1"b;
137             do Ncols = 20 to 1 by -1 while (Sdoes_not_fit);
138                  total = -2;
139                  Nrows = divide(count+Ncols-1, Ncols, 17, 0);
140                  Nused_cols_in_last_row = mod(count,Ncols);
141                  if Nused_cols_in_last_row = 0 then
142                       Nused_cols_in_last_row = Ncols;
143                  if Nused_cols_in_last_row >= Ncols then do;
144                       do Icol = 1 to Nused_cols_in_last_row;
145                            total = total +
146                                      length(r_matrix(Icol, Nrows))+2;
147                       end;
148                       do Icol = Icol to Ncols;
149                            total = total +
150                                      length(r_matrix(Icol, Nrows-1))+2;
151                       end;
152                       if total <= linesize then Sdoes_not_fit = "0"b;
153                  end;
154             end;
155             Ncols = Ncols + 1;
156
```

a (ouple

WORKSHOP ONE

```
157         /* Output the values, starting with the assumed   */
158         /* values, then the computed (output in columns). */
159         /* Computed values are output in right-justified  */
160         /* columns.  Each row (line) is formatted and     */
161         /* then output.                                   */
162
163         put file(sysprint) edit ("Fdeg =", grouping,
164              ":") (a, f(3), a);
165         put file(sysprint) edit ("(assumed beginning of series)",
166              (result(cycle) do cycle=-grouping to -1))
167              (skip, a, skip, (grouping)(a, x(1)));
168         put file(sysprint)
169             edit ("(remainder of series)")(skip, a);
170         put file(sysprint) skip;
171         do Irow = 1 to Nrows;
172              msg = "";
173              do Icol = 1 to Ncols-1;
174                   msg = msg || substr(SPACES, 1,
175                         length(r_matrix(Icol,Nrows)) -
176                         length(r_matrix(Icol,Irow)));
177                   msg = msg || r_matrix(Icol, Irow);
178                   msg = msg || "  ";
179              end;
180              if Irow*Ncols + Icol <= count then do;
181                   msg = msg || substr(SPACES, 1,
182                         length(result(hbound(result,1))) -
183                         length(r_matrix(Icol, Irow)));
184                   msg = msg || r_matrix(Icol, Irow).
185              end;
186              write file(sysprint) from(msg);
187         end;
188         put skip(2) file(sysprint);
189         end gen_fib;
190
191     end fib;
```

WORKSHOP ONE

A FORTRAN VERSION

```
 1              logical sfirst /.true./
 2              save sfirst
 3
 4      c       Output instructions to the user, but only the first
 5      c        time 'fib' is invoked in each process.
 6
 7              if (.not.sfirst) goto 10
 8              print, "Enter first degree and count"
 9              print, "For example, First degree = 2, count = 10"
10              print, "To stop, enter First degree = 1"
11              sfirst = .false.
12
13      c       Prompt for First degree and count.
14          10  print, "First degree, count?"
15              read, ifdeg, icount
16
17      c       Stop when First degree is 1.
18              if (ifdeg - 1) 15,15,25
19
20          15  print, "Count =",icount
21
22      c       Compute and output each Fibonacci series.
23      c        Then get next set of input values.
24
25              do 22 i = 2,ifdeg
26          22  call gen_fib (i,icount)
27              goto 10
28
29          25  stop
30              end
31
32
33
34              subroutine gen_fib (igrouping, icount)
35
36      c       This subroutine actually computes the Fibonacci series.
37      c        'iresult' will be filled with the proper values,
38      c        while 'jresult' is a convenient equivalent view of
39      c        the solution which will be used for printing purposes.
40      c        The 'n' array holds the most recent terms to be added
41      c        together to obtain the next term in the series.
42
43              double precision n(10), total
44              dimension iresult(30)
45              dimension jresult(10,3)
46              equivalence (iresult(1),jresult(1,1))
47
48      c       Tell him which degree of Fibonacci series this is
49
50              print
51              print,"Degree =",igrouping
```

```
52              print
53
54     c        Initialize the assumed values which begin the
55     c         series. All are 0 but the last, which is
56     c         1 - also put the assumed values into the
57     c         output array
58
59              do 10 i = 1,igrouping-1
60              iresult(i) = 0
61         10   n(i) = 0
62              iresult(igrouping) = 1
63              n(igrouping) = 1
64
65     c        Compute remaining values of series, and put in
66     c         the output array
67
68              do 20 icycle = igrouping+1, icount
69              total = 0
70              do 15 i = 1, igrouping
71         15   total = total + n(i)
72              iresult(icycle) = total
73         20   n(mod(icycle-1,igrouping)) = total
74
75     c        The output will be printed with assumed values
76     c         preceding computed values. The computed values
77     c         will be printed along with the assumed values
78     c         in three columns. Hence, there will always be
79     c         'irow' rows with three values, and the last
80     c         row may have 1 2 or 3 values
81
82              icol = mod(icount,3)
83              irow = icount / 3
84
85              do 22 j=1,irow
86         22   print,(jresult(i,j),i=1,3)
87              if (icol) 30,30,25
88         25   print,(jresult(irow+1,i),i=1,icol)
89         30   return
90              end
```

WORKSHOP ONE

The following dialogue shows the correct operation of the PL/I version of the fib program. The dialogue is slightly different for the FORTRAN and COBOL versions, but the concept is basically the same for all three programs. The programs shown above may have errors which prevent them from generating these results. Use probe to find the errors. Change the source to correct the errors, recompile the program and continue testing it until it prints the results shown below.

```
 1   !  . pl1 fib -table
 2      PL/I
 3      r 2247 5.091 51.312 227
 4
 5   !  fib
 6      Enter Fdeg and/or count, followed by a ";" character.
 7      For example,
 8                Fdeg = 2, count=10;
 9      These are the default values.  To stop, enter
10                Fdeg = 1;
11
12      Enter data, or just a ";" char:   !  Fdeg = 4, count=9;
13
14
15      count=        9;
16      Fdeg =  2:
17      (assumed beginning of series)
18      0 1
19      (remainder of series)
20      1  2  3  5  8  13  21  34  55
21
22
23      Fdeg =  3:
24      (assumed beginning of series)
25      0 0 1
26      (remainder of series)
27      1  2  4  7  13  24  44  81  149
28
29
30      Fdeg =  4:
31      (assumed beginning of series)
32      0 0 0 1
33      (remainder of series)
34      1  2  4  8  15  29  56  108  208
35
36
37      New data, or just a ";" char:   !  Fdeg=1;
38      r 2248 0.249 0.228 19
```

WORKSHOP TWO

A trace Workshop

1. Use 'trace' to monitor the value of the arguments on return from the system program 'expand_pathname_' (trace -ag 1 -out expand_pathname_). Do you know what that program is used for? Issue the trace command to list the status of the expand_pathname_ trace entry (trace -status expand_pathname_). Do you know what those counters mean? Now issue a print command and observe what happens. Issue the command 'print >ldd>include>its.incl.pl1' and see what happens. Now try the command 'ds baloney'. Finally, try the command 'pr <>foo'. What happens? What do you think the fourth argument of expand_pathname_ is used for??

2. Now print the status of the expand_pathname_ (trace -status expand_pathname_). Also, list the control template for trace (trace -tp). Remove the trace entry for expand_pathname_ and reset the control template to its initial form (trace -remove expand_pathname_ -ag 0).

>fsoed>wksps>F21>S1>

3. Execute the following recursive program (see >udd>F19>sl>R2.pl1 and >udd>F19>sl>R2):

```
R2:  proc;
dcl  (sysin, sysprint) file;
dcl  (n, i) fixed bin;
dcl   R2$Seq entry (fixed bin);

          open file (sysprint) stream output env (interactive);
          put file (sysprint) skip list ("Enter value...");
          get list (n);
          call R2$Seq (n);
          return;

Seq:      entry (i);
          put file (sysprint) list (i);
          if i > 1 then do;
               call R2$Seq (i-1);
               put file (sysprint) list (i);
          end;
     end;
```

Execute the program with a value of 5. You should get 5 4 3 2 1 2 3 4 5. Type the command to trace this program, printing the argument values at input to every second call(trace -in -ag 2 R2$Seq). Now run R2 again using the value 5 and observe what happens. List the trace status of R2$Seq and then turn on the governing facility (trace -st R2$Seq -govern on R2$Seq). The governing facility is used to help trap runaway recursive procedures. Run R2 once again, using the value 5. List the status of R2$Seq(trace -st R2$Seq). Note the maximum recursion

level. Now lets see if we can blow it out. Run R2 again, this time with an input of 12. What happened? Since our procedure is not really a runaway program, type the 'start' command to continue. Did you realize that we were at command level? Why?(Hint: the govern facility stops on depth levels which are a multiple of 10 to give you a chance to find out what's happening).

4. Now, stop tracing R2$Seq, and reset the template. You may first want to issue the command 'trace -st * -tp' to see the current state of affairs. Next, copy the following three simple pl1 programs, which are found in the directory >udd>F19>sl:

```
init: proc;
dcl 1 S external static,
    2 sentinel fixed bin,
    2 array (5) float;
dcl  ioa_ entry options (variable);
dcl  addr builtin;

          sentinel = 0;
          call ioa_ ("sentinel located at ^p", addr (sentinel));
     end;


load: proc;
dcl  sysprint file;
dcl 1 S external static,
    2 sentinel fixed bin,
    2 array (5) float;
dcl  i fixed bin;

          open file (sysprint) stream output env (interactive);
          do i = 1 to 5;
               array (i) = 3* (i-2);
          end;
          put file (sysprint) skip (3) list (array);
     end;
```

```
print_stat: proc;
dcl  sysprint file;
dcl 1 S external static,
    2 sentinel fixed bin,
    2 array (5) float;

          open file (sysprint) stream output env (interactive);
          put skip (2) list ("*sentinel location clobbered!!*");
          put skip (1) data (S);
     end;
```

Compile each of these programs. Now run init. It should tell you that the external static member variable 'sentinel' is located at some segment number|offset. We want to use the watch facility of trace to find out whether any program is clobbering that location. Hence, issue the command to have trace watch that location (trace -watch seg_no|offset). Next, let's monitor the 'load' program, and if anything goes wrong, let's cause the print_stat program to be called as the 'stop_proc' instead of the command processor. Issue the command to do this(trace -stop_proc print_stat load). Now run the load program and see what happens. So far so good! Now modify the load program so that it inadvertently changes the value of sentinel by changing the 'do' statement to 'do i = 0 to 5'. Recompile and run the load program. What happens? The watch facility should have stopped your load program since the value of its watch location changed, and it should have called the appropriate 'stop_proc'. Did it? The watch facility is very useful when trying to track down the one procedure in a group of procedures that is going a bit awry, or has modified some externally accessible error cell,etc.

WORKSHOP THREE

On the Programming Environment: A Quiz

1. Object segments are an essential part of the Multics programming environment. Name the 8 sections into which an object segment is divided. Describe the contents of each section in general terms. Are all the sections always present in every object segment? If not, which are optional.

2. The system maintains information about the user ring programming environment in 2 important segments. Can you name these 2 segments? Briefly describe what kinds of data the system keeps in each segment. What directory are the segments located in? How are these segments protected from accidental damage?

3. One of the most powerful features of the Multics programming environment is the Dynamic Linking mechanism. The programming environment uses this mechanism to find an object segment which is called by another program.

   Briefly name the important steps taken by the dynamic linker to find an object segment.

   At what point during the compilation or execution of the calling program does dynamic linking take place?

   How often does it take place?

   If one program calls an object segment and then a second program (a second object segment) calls that same target object segment, are the same steps followed in both cases to dynamically link to the called object segment? If not, how does dynamic linking differ during the second call?

4. A fatal process error occurs when the system decides that the programming environment can no longer operate correctly. When this occurs, the system takes control of the user's terminal, prints a brief error message, and creates a new user process.

   Under what circumstances might the system decide that the programming environment can no longer operate?

   What 2 programming errors are the most common causes of fatal process errors?

   Briefly describe a procedure for finding the cause of a fatal process error.

5. In chapter four 11 different classes of data (storage classes) were discussed which can be used in PL/I, FORTRAN and/or COBOL programs. For each class of data, describe:

   o Where the data is stored. Give the logical name of a segment, table, or area; also give the pathname of the segment containing the data class.

   o The major characteristics of the storage class. (When the data is allocated, when freed, when initialized, can it be shared between programs, etc?)

   For example, one class of storage is:

   based storage, in an area: stored in a program-supplied area, such as the system free area (segment system_free_n_ in the process dir). Storage is known only to 1 program, is explicitly allocated and freed, is initialized by allocate or locate statements, and has a location maintained by a pointer or offset qualifier.

5. (More space for the answer)

```
{ō ō}
  V
   \      /--------------------------------------\
    \____/   Happiness is Multing the day away    \
         \                                        /
          \--------------------------------------/
```

Not To Be Reproduced