

R-17

BASIC HAL/S PROGRAMMING

06/23/97

Craig Schulenberg

SPACE SHUTTLE SYSTEMS

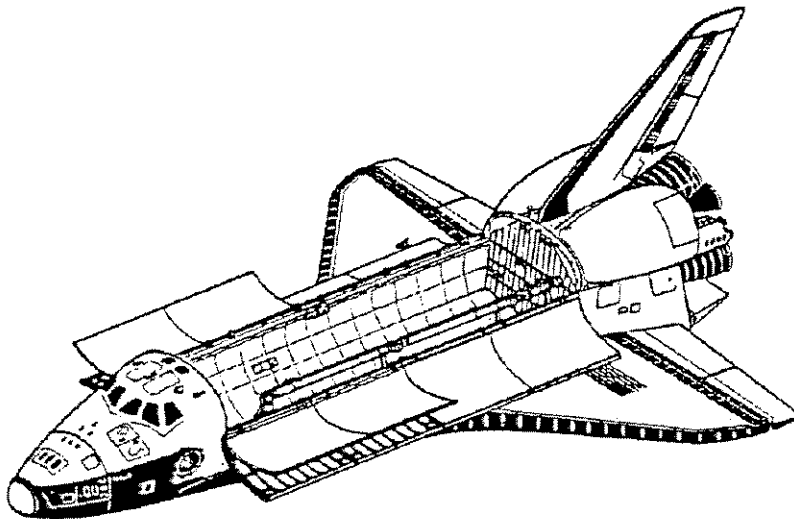
LOCKHEED MARTIN

Space Information Systems

1812 Space Park Drive

P.O. Box 58487

Houston, TX 77258





BASIC HAL/S PROGRAMMING CLASS INFORMATION

Course Number: OTP080

Course Length: Three two-hour classes

Intended Audience: HAL/S Programmers

Prerequisites: Some programming skills, but not necessarily HAL/S

Objectives: To provide an overview of the HAL/S language

Content:

Day 1)
Program structure, name scoping, data types, declares, arrays, initialization, replace macros, subscripting, integer/scalar conversions, expressions, assignments, conditionals and IF statements, statement groups (e.g. DO, DO FOR), procedures and functions

Day 2)
Bit strings (initialization, subscripting, expressions, assignments, comparisons), structures, raveling and unraveling, explicit conversions

Day 3)
Data storage and access (e.g. #D, #P), RIGID, exclusive and reentrant procedures and functions, CSECTs, stacks and stack frames, NAME variables, real time statements

BLOCKS

COMPOOL }
PROGRAM }

ALWAYS SEPARATE COMPILATION UNITS

PROCEDURE }
FUNCTION }

MAY BE SEPARATE COMPILATION UNITS (COMSUBS) --
OR NESTED IN A PROGRAM, OR FUNCTION

{ TASK }
{ UPDATE }

ALWAYS INTERNAL BLOCKS -- WILL BE DISCUSSED
LATER

COMPOOLS CONTAIN ONLY DATA (POSSIBLY REPLACE STATEMENTS).

PROGRAMS ARE "SCHEDULED" AS PROCESSES VIA THE OPERATING SYSTEM.

PROCEDURES ARE "CALLED".

FUNCTIONS ARE "INVOKED"

FORMATS

C1: COMPOOL;

DECLARES

CLOSE C1;

P1: PROGRAM;

DECLARES

NESTED BLOCKS

CODE

CLOSE P1;

I1: PROCEDURE (A, B) ASSIGN(C);

PARAMETER
DECLARES

DECLARES

NESTED
BLOCKS

CODE

CLOSE I1;

INPUT PARAMETERS

ASSIGN PARAMETER

FORMATS (CONT.)

I2: FUNCTION (A) MATRIX;

PARAMETER
DECLARES

INPUT
PARAMETER

DECLARES

NESTED
BLOCKS

CODE

RETURN MAT;
CLOSE I2;

NOTE: FUNCTIONS HAVE NO ASSIGN PARAMETERS, BUT MUST HAVE A RETURN STATEMENT.

FORMATS (CON'T.)

COMPOOL

USED VIA REFERENCES TO DATA

PROGRAM

SCHEDULE P1 PRIORITY(100);

PROCEDURE

CALL I1(X, Y) ASSIGN(Z);

 ↑ ↑
 INPUT ASSIGN
 ARGUMENTS ARGUMENT

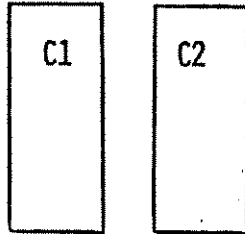
FUNCTION

 FUNCTION NAME
 |
*MATB = I2(S);

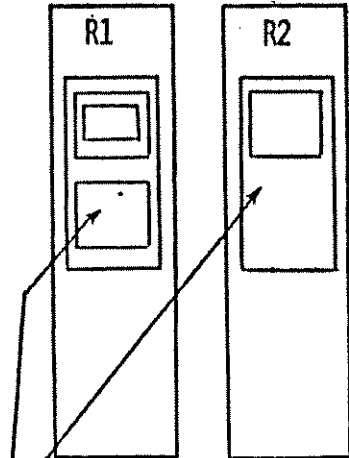
 ↑
 INPUT
 ARGUMENT

PROGRAM COMPLEX

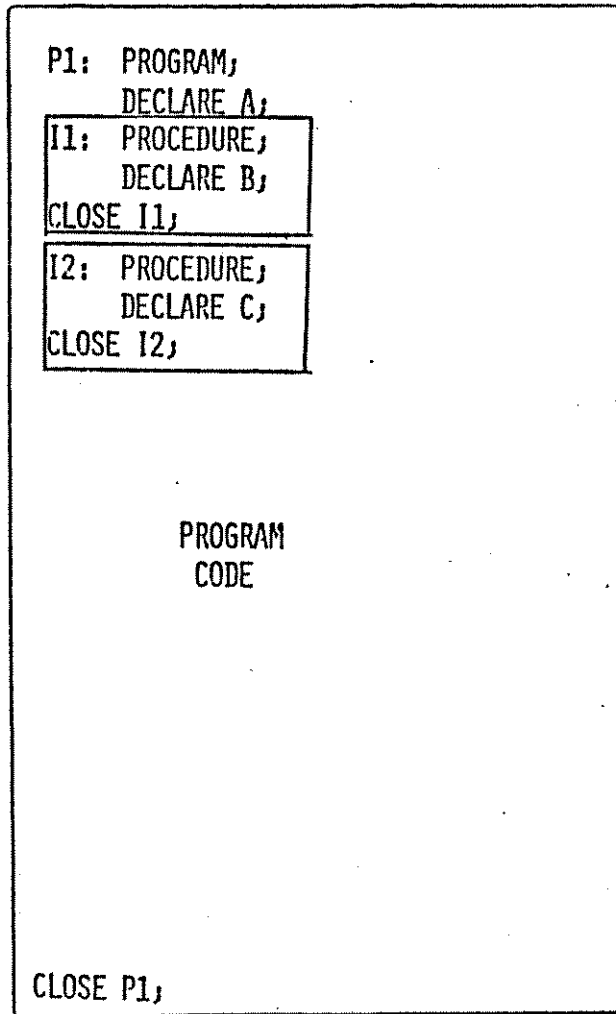
COMPOOLS



COMSUBS



NESTED PROCEDURES/FUNCTIONS



TEMPLATES

COMPOOL:

```
C1: COMPOOL,  
    DECLARE A, B,  
    DECLARE C MATRIX;  
    CLOSE C1;
```

COMPOOL TEMPLATE:

```
C1: EXTERNAL COMPOOL,  
    DECLARE A, B,  
    DECLARE C MATRIX;  
    CLOSE C1;  
D  VERSION 3
```

COMPILER

```
graph TD; A["C1: COMPOOL,  
    DECLARE A, B,  
    DECLARE C MATRIX;  
    CLOSE C1;"] --> B[COMPILER]; B --> C["C1: EXTERNAL COMPOOL,  
    DECLARE A, B,  
    DECLARE C MATRIX;  
    CLOSE C1;  
D  VERSION 3"];
```

A COMPOOL TEMPLATE IS TEXT THAT IS VIRTUALLY IDENTICAL TO THE COMPOOL SOURCE ITSELF -- EXCEPT THAT KEYWORD "EXTERNAL" IS ADDED. IN ADDITION, THE COMPILER GENERATES A VERSION NUMBER ON AN APPENDED "D" (DIRECTIVE) CARD.

TEMPLATES (CON'T.)

PROGRAM:

```
P1: PROGRAM,  
  DECLARE R,  
  ...  
  I1: PROCEDURE,  
  ...  
  CLOSE I1;  
  ...  
  PROGRAM  
  CODE  
  CLOSE P1;
```

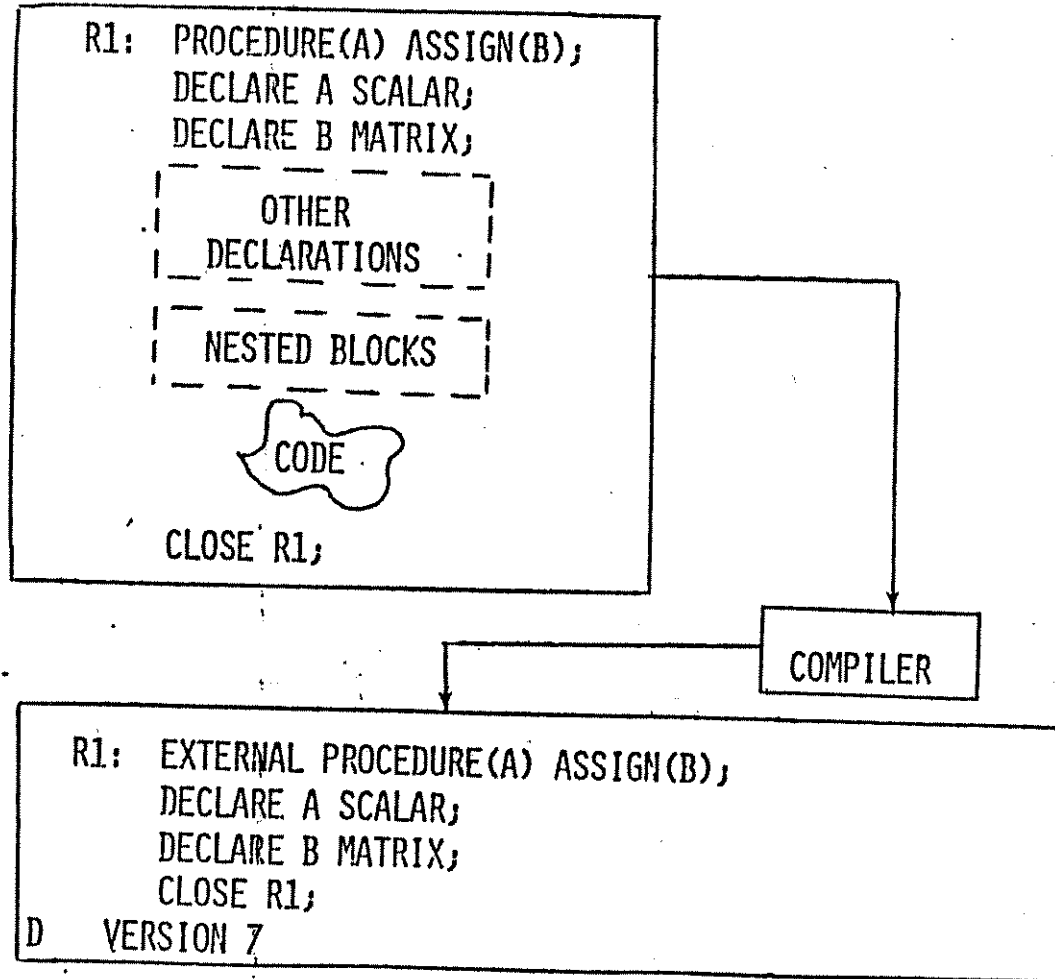
COMPILER

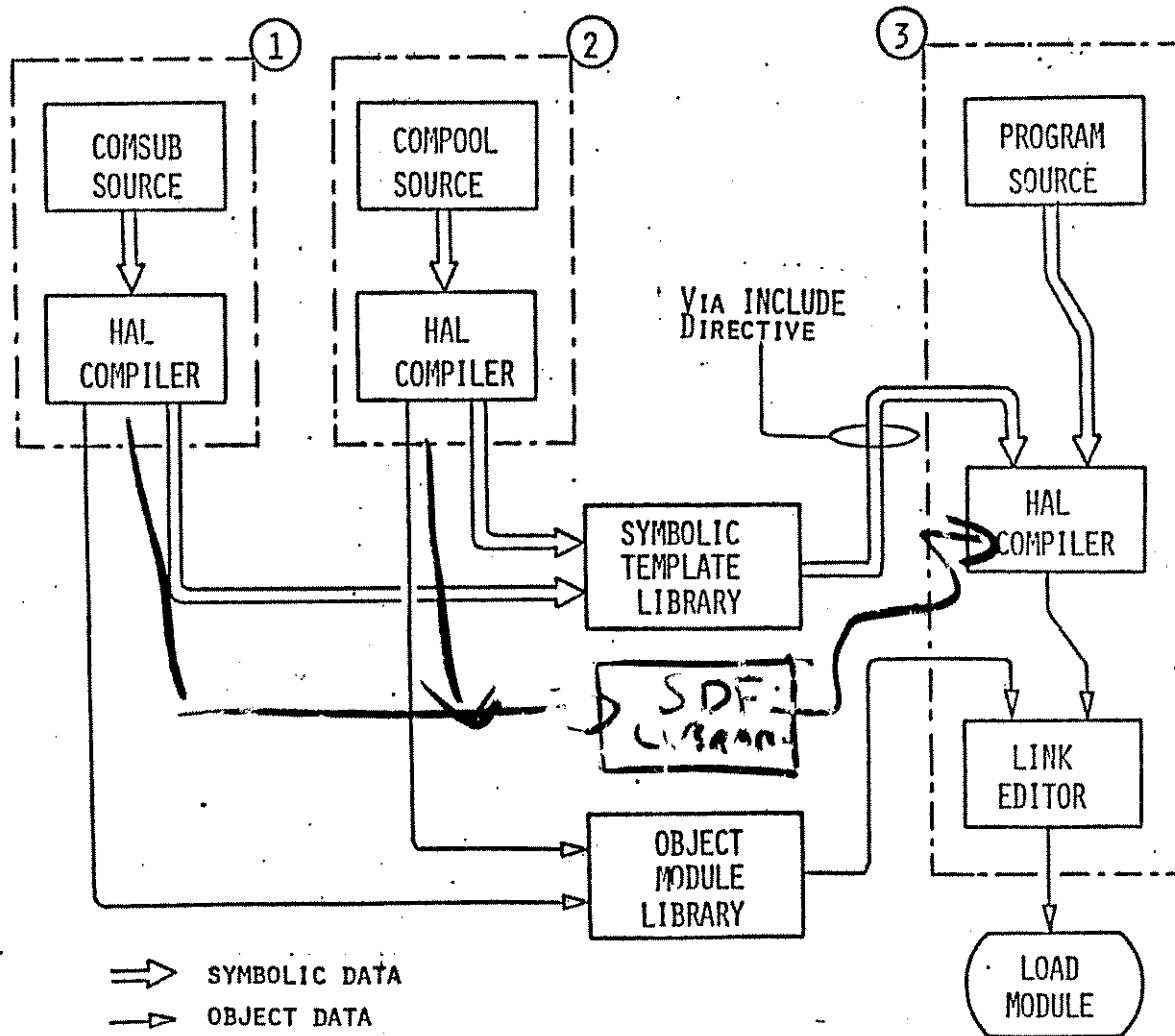
PROGRAM TEMPLATE:

```
P1: EXTERNAL PROGRAM,  
  CLOSE P1,  
D VERSION 1
```

TEMPLATES (CON'T.)

PROCEDURE (COMSUB):





HAL COMPILATION SYSTEM

TEMPLATES (CON'T.)

TEMPLATES ARE "INCLUDED" VIA A COMPILER "D" (DIRECTIVE) CARD,
E.G.,

```
COL 1  
  ↓  
  D  INCLUDE TEMPLATE C1 {NOLIST}
```

RULE: FOR ALL COMPOOLS, COMSUBS, AND PROGRAMS REFERENCED BY A
COMPILATION UNIT, A TEMPLATE MUST BE INCLUDED.

RULE: ALL INCLUDED TEMPLATES MUST PRECEDE THE FIRST LINE
OF THE COMPILATION UNIT BEING COMPILED.

SAMPLE PROGRAM

```
COL 1
  ↓
  →D INCLUDE TEMPLATE C1 NOLIST
  →D INCLUDE TEMPLATE R1 NOLIST
  →D INCLUDE TEMPLATE P1 NOLIST
    P2: PROGRAM;
      DECLARE MATRIX, Q, R, S;
      DECLARE X, Y, Z;
      I1: PROCEDURE(V) ASSIGN(W);
        DECLARE V, W;
        ...
        CLOSE I1;
        ... (OTHER INTERNAL BLOCKS)
    PROGRAM
      SCHEDULE P1 PRIORITY(40);
      ...
    COMSUB
      CALL R1(X) ASSIGN(S);
      ...
    COMPOOL VARIABLE
      Q = C;
      ...
      CLOSE P2;
```

SOURCE

```
C          OODAP+POOL PEVISION 7                                000000
M OODAP+POOL: COMPOOL;                                         000001
M DECLARE PRINT+DFCS+DATA INTEGER SINGLE INITIAL(0);         000002
M DECLARE JSLECT+PRINTFL BOOLEAN INITIAL(OFF);                000002
C                                                                 000003
C                                                                 000004
C          *** FROM ENGINE+PRETHRUST+TRIM, OMS+ENG+CMD, DELTA+OMEGA+OMS+ENGINE, 000005.
C          ENGINE+CG+CMD ***                                     000006.
C                                                                 000007
M DECLARE VECTOR(3) SINGLE INITIAL(9.3, 0., .3),              000008
M OMS1+CG, OMS2+CG; /* VEH COORDS, OMS TO CG */              000009
M DECLARE LBM+TO+KG CONSTANT(1./2.2046);                       000009
M DECLARE G+TO+MTRS+PER+SEC+SQ CONSTANT(9.8066);             000009
M DECLARE LBF+TO+N CONSTANT(LBM+TO+KG G+TO+MTRS+PER+SEC+SQ); 000009
M DECLARE VEHICLE+CG VECTOR SINGLE INITIAL(9.9974, -.005, .6858); 000009
M DECLARE VEHICLE+MASS INITIAL(122470);                       000009
M DECLARE ARRAY(44) BOOLEAN INITIAL(OFF), JFRIL, JONLST;     000009
M DECLARE VEHICLE+INVERSE+INERTIA MATRIX(3,3) SINGLE;       000010
M DECLARE VEHICLE+INERTIA MATRIX SINGLE                       000011
M   INITIAL(1056182., 4067.454, -216930.9, 4067.454, 7421747., 000011
M -1355.818, -216930.9, -1355.818, 7721383.);                 000011
M DECLARE RAD+TO+DEG SCALAR SINGLE CONSTANT(57.2957795);    000012
M DECLARE DEG+TO+RAD SCALAR SINGLE CONSTANT(.0174532925);    000013
M DECLARE PI SCALAR DOUBLE CONSTANT(3.141592654);            000014
M DECLARE TWO+PI SCALAR DOUBLE CONSTANT(2. PI);              000014
M DECLARE THR SCALAR SINGLE CONSTANT(26689.2);               000015
M DECLARE FT+TO+M CONSTANT(.3048);                           000015
M DECLARE IN+TO+M CONSTANT(FT+TO+M/12.);                     000015.
```

SOURCE (CON'T.)

```

C                                     000017
C      **** FROM MS&C ****          000018
C                                     000019
M DECLARE BARBEQUE+RATE SCALAR SINGLE INITIAL(2. DEG+TO+RAD); 000020
M      DECLARE BOOLEAN INITIAL(ON), RCS+ROTATION, 000021
M      ATT+MNVR, RCS+TRANSLATION, RCS+TRANS+AUTO+MANUAL, 000022
M      OFC+RESTART, RCS+ROT+AUTO+MANUAL, TVC+AUTO+MANUAL, 000023
M CLOSED+OPEN+LOOP+TRIM, ACCLFS+NEEDED; 000024
M      DECLARE BOOLEAN INITIAL(OFF), TWO+AXIS, THREE+AXIS, 000025
M      LCL+VERT+ATT, PAYLD+SUP+CMDS, TRACKING, BBQ, OMS+FAIL+DETECT, 000026
M      OMS1+ON+CMD, OMS2+ON+CMD, OMS+PRETHRUST, 000027
M      OMS1+FAIL, OMS2+FAIL; 000028
M      DECLARE ARRAY(2) BOOLEAN INITIAL(OFF), OMS+ARM+REQ, OMS+ON+REQ; 000029
M DECLARE VECTOR SINGLE INITIAL(0), POINTING+VECTOR+CMD, 000030
M      POSITION, VELOCITY, BODY+POINTING+VECTOR; 000031
M DECLARE ELF SCALAR SINGLE CONSTANT(11. DEG+TO+RAD); 000032
M DECLARE COSELF SCALAR SINGLE CONSTANT(COS(ELF)); 000033
M DECLARE SINELF SCALAR SINGLE CONSTANT(SIN(ELF)); 000034
M DECLARE BODY+TO+NB MATRIX SINGLE CONSTANT(COSELF, 0., SINELF, 000035
M      0., 1., 0., -SINELF, 0., COSELF); 000036
M DECLARE MATRIX SINGLE, NB+TO+SM, NB+TO+QA; 000037
M DECLARE SCALAR SINGLE INITIAL(0), SINOGA, SINMOA, COSOGA, COSMOA; 000038
M DECLARE ARRAY(3) SCALAR SINGLE INITIAL(0), QA+DESIRED, 000039
M      AUTO+QA+DESIRED, QA+MANEUVER+TERMINAL; 000040
M DECLARE NEWORD1 BIT(16) INITIAL(BIN'0111000000111000'); 000041
M DECLARE NEWORD2 BIT(16) INITIAL(BIN'0000111000000111'); 000042
M DECLARE NEWORD3 BIT(16) INITIAL(BIN'0001011001000111'); 000043
M DECLARE ROT+OPTION+ACK ARRAY(3) INTEGER INITIAL(0); 000044
M DECLARE TRANS+OPTION+ACK ARRAY(3) BIT(1) INITIAL(OFF); 000045
M DECLARE VECTOR SINGLE, OMEGA+C; 000046
M CLOSE OODAP+POOL; 000047

```

OUTPUT WRITER LISTING

SYMBOLS = 200
 MACROSIZE = 500
 LITSTRINGS = 2000
 COMPUNIT = 0
 XREFSIZE = 2000
 CARDTYPE =
 LABELSIZE = 1200

HAL/S COMPILATION			I N T E R M E T R I C S , I N C .		OCTOBER
SRN STMT			SOURCE		
000000	CT	00DAP+POOL REVISION 7			
HAL/S COMPILATION			I N T E R M E T R I C S , I N C .		OCTOBER
SRN STMT			SOURCE		
000001	1 M†	00DAP+POOL:			
000001	1 M†	CONPOOL;			
000002	2 M†	DECLARE PRINT+DFCS+DATA INTEGER SINGLE INITIAL(0);			
000002	3 M†	DECLARE JSLECT+PRINTFL BOOLEAN INITIAL(OFF);			
000003	CT				
000004	CT				
000005	CT	**** FROM ENGINE+PRETHRUST+TRIM,OMS+END+CMD,DELTA+OMEGA+OMS+ENGINE,			
000006	CT	ENGINE+CG+CMD ****			
000007	CT				
000008	4 M†	DECLARE VECTOR(3) SINGLE INITIAL(9.3, 0, .3),			
000008	4 M†	OMS1+CG, OMS2+CG;			
000009	5 M†	DECLARE LBM+TO+KG CONSTANT(1. / 2.2046);			
000009	6 M†	DECLARE G+TO+MTRS+PER+SEC+SQ CONSTANT(9.8066);			
000009	7 M†	DECLARE LBF+TO+M CONSTANT(LBM+TO+KG G+TO+MTRS+PER+SEC+SQ);			
000009	8 M†	DECLARE VEHICLE+CG VECTOR SINGLE INITIAL(9.9974, -.005, .6850);			
000009	9 M†	DECLARE VEHICLE+MASS INITIAL(122470);			
000009	10 M†	DECLARE ARRAY(44) BOOLEAN INITIAL(OFF),			
000009	10 M†	JFAIL, JONLST;			
000010	11 M†	DECLARE VEHICLE+INVERSE+INERTIA MATRIX(3, 3) SINGLE;			
000011	12 M†	DECLARE VEHICLE+INERTIA MATRIX SINGLE INITIAL(1056182., 4067.454, -216930.9,			
000011	12 M†	-1355.818, -216930.9, -1355.818, 7721383.))			
000012	13 M†	DECLARE RAD+TO+DEG SCALAR SINGLE CONSTANT(57.2957795);		4067.454, 7421747.,	
000013	14 M†	DECLARE DEG+TO+RAD SCALAR SINGLE CONSTANT(.0174532925);			
000014	15 M†	DECLARE PI SCALAR DOUBLE CONSTANT(3.141592654);			
000014	16 M†	DECLARE TWO+PI SCALAR DOUBLE CONSTANT(2. PI);			
000015	17 M†	DECLARE THR SCALAR SINGLE CONSTANT(26689.2);			
000015	18 M†	DECLARE FT+TO+M CONSTANT(.3048);			
000015	19 M†	DECLARE IN+TO+M CONSTANT(FT+TO+M / 12.);			

1-14

COMPILATION TEMPLATES (COMPOOL)

@@OODAPP

```
OODAP+POOL: EXTERNAL COMPOOL ; DECLARE PRINT+DFCS+DATA INTEGER SINGLE I
NITIAL ( 0 ) ; DECLARE JSLECT+PRINTFL BOOLEAN INITIAL ( OFF ) ; DECLARE
VECTOR ( 3 ) SINGLE INITIAL ( 9.3 , 0 , .3 ) , OMS1+CO , OMS2+CO ; DEC
LARE LBM+TO+KG CONSTANT ( 1. / 2.2046 ) ; DECLARE Q+TO+MTRS+PER+SEC+SQ
CONSTANT ( 9.8066 ) ; DECLARE LBF+TO+N CONSTANT ( LBM+TO+KG Q+TO+MTRS+P
ER+SEC+SQ ) ; DECLARE VEHICLE+CO VECTOP SINGLE INITIAL ( 9.9974 , - .00
5 , .6858 ) ; DECLARE VEHICLE+MASS INITIAL ( 122470 ) ; DECLARE ARRAY (
44 ) BOOLEAN INITIAL ( OFF ) , JFAIL , JONLST ; DECLARE VEHICLE+INVERS
E+INERTIA MATRIX ( 3 , 3 ) SINGLE ; DECLARE VEHICLE+INERTIA MATRIX SING
LE INITIAL ( 1056182. , 4067.454 , - 216930.9 , 4067.454 , 7421747. , -
1355.818 , - 216930.9 , - 1355.818 , 7721303. ) ; DECLARE RAD+TO+DEG S
CALAR SINGLE CONSTANT ( 57.2957795 ) ; DECLARE DEG+TO+RAD SCALAR SINGLE
CONSTANT ( .0174532925 ) ; DECLARE PI SCALAR DOUBLE CONSTANT ( 3.14159
2654 ) ; DECLARE TWO+PI SCALAR DOUBLE CONSTANT ( 2. PI ) ; DECLARE THR
SCALAR SINGLE CONSTANT ( 26609.2 ) ; DECLARE FT+TO+M CONSTANT ( .3048 )
; DECLARE IN+TO+M CONSTANT ( FT+TO+M / 12. ) ; DECLARE BARBEQUE+RATE S
CALAR SINGLE INITIAL ( 2. DEG+TO+RAD ) ; DECLARE BOOLEAN INITIAL ( ON )
, RCS+ROTATION , ATT+MNVR , RCS+TRANSLATION , RCS+TRANS+AUTO+MANUAL ,
OFC+RESTART , RCS+ROT+AUTO+MANUAL , TVC+AUTO+MANUAL , CLOSED+OPEN+LOOP+
TRIM , ACCLFS+NEEDED ; DECLARE BOOLEAN INITIAL ( OFF ) , TWO+AXIS , THR
EE+AXIS , LCL+VERT+ATT , PAYLD+SUP+CMDS , TRACKING , BBQ , OMS+FAIL+DET
ECT , OMS1+ON+CMD , OMS2+ON+CMD , OMS+PRETHRUST , OMS1+FAIL , OMS2+FAIL
; DECLARE ARRAY ( 2 ) BOOLEAN INITIAL ( OFF ) , OMS+ARM+REQ , OMS+ON+R
EQ ; DECLARE VECTOR SINGLE INITIAL ( 0 ) , POINTING+VECTOR+CMD , POSITI
ON , VELOCITY , BODY+POINTING+VECTOR ; DECLARE ELF SCALAR SINGLE CONSTA
NT ( 11. DEG+TO+RAD ) ; DECLARE COSELF SCALAR SINGLE CONSTANT ( COS ( E
LF ) ) ; DECLARE SINELF SCALAR SINGLE CONSTANT ( SIN ( ELF ) ) ; DECLAR
E BODY+TO+NB MATRIX SINGLE CONSTANT ( COSELF , 0. , SINELF , 0. , 1. ,
0. , - SINELF , 0. , COSELF ) ; DECLARE MATRIX SINGLE , NB+TO+SM , NB+T
O+QA ; DECLARE SCALAR SINGLE INITIAL ( 0 ) , SINOGA , SINMGA , COSOGA ,
COSMGA ; DECLARE ARRAY ( 3 ) SCALAR SINGLE INITIAL ( 0 ) , QA+DESIRED
, AUTO+QA+DESIRED , QA+MANEUVER+TERMINAL ; DECLARE NEWORD1 BIT ( 16 ) I
NITIAL ( BIN '0111000000111000' ) ; DECLARE NEWORD2 BIT ( 16 ) INITIAL
( BIN '0000111000000111' ) ; DECLARE NEWORD3 BIT ( 16 ) INITIAL ( BIN '
0001011001000111' ) ; DECLARE ROT+OPTION+ACK ARRAY ( 3 ) INTEGER INITIA
L ( 0 ) ; DECLARE TRANS+OPTION+ACK ARRAY ( 3 ) BIT ( 1 ) INITIAL ( OFF
) ; DECLARE VECTOR SINGLE , OMEGA+C ;
CLOSE ;
D VERSION : 1
```

(PROGRAM)
@@OFCREC

OFC+RECON: EXTERNAL PROGRAM ;
CLOSE ;

COMPILATION TEMPLATES (COMSUBS)

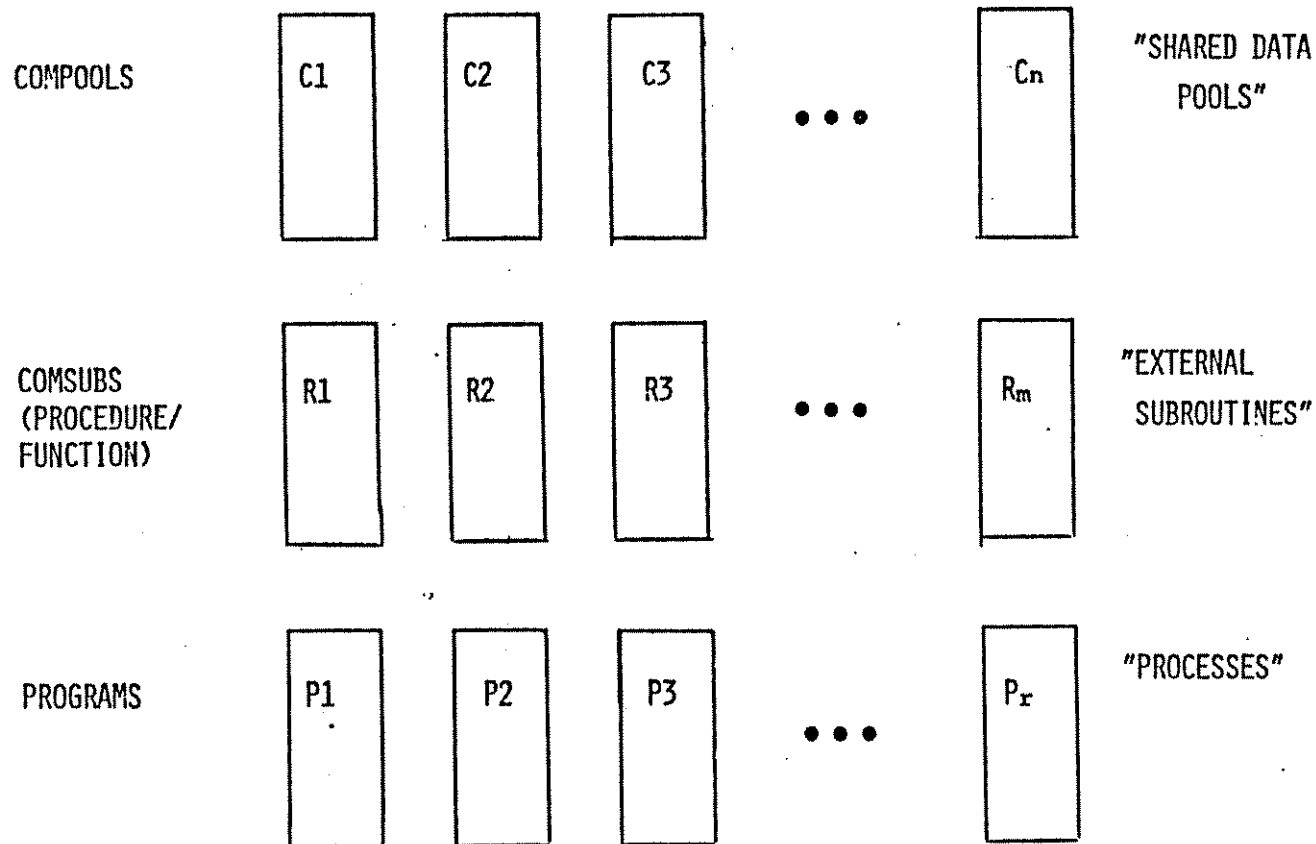
@@ECITOG

```
ECITOGEO: EXTERNAL PROCEDURE ( T , R , V ) ASSIGN ( H , LAT , LONG , VMAQ , GAM  
M , AZIM ) ; DECLARE T SCALAR DOUBLE , R VECTOR DOUBLE , V VECTOR DOUBLE , H ,  
LAT , LONG , VMAQ , GAMM , AZIM ;  
CLOSE ;  
D VERSION : 1
```

@@ROTATE

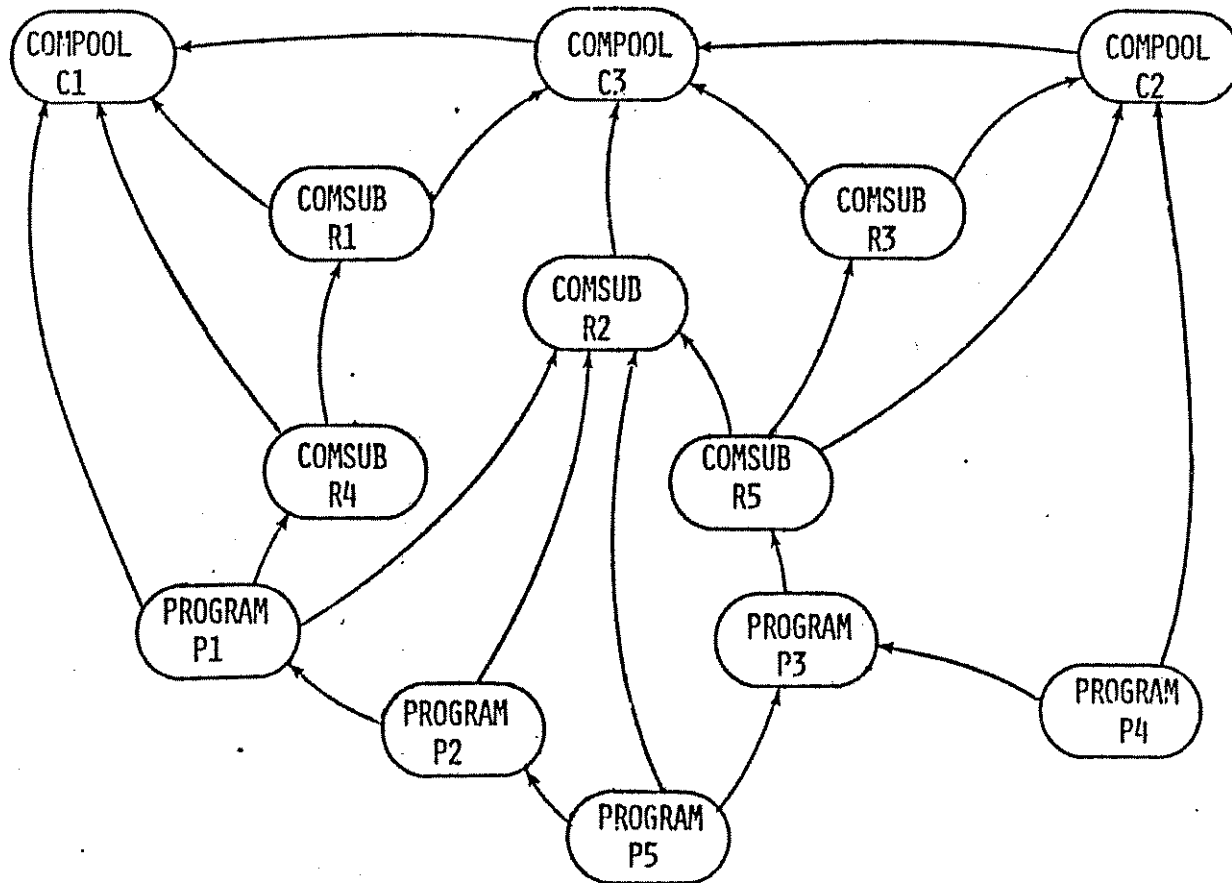
```
ROTATE: EXTERNAL FUNCTION ( RZERO , DLONG ) VECTOR DOUBLE ; DECLARE SCALAR DOUB  
LE , DLONG ; DECLARE VECTOR DOUBLE , RZERO ;  
CLOSE ;  
D VERSION : 1
```

PROGRAM COMPLEX



EACH OF THESE REPRESENT SEPARATE COMPILATION UNITS AND WILL PRODUCE TEMPLATES.

COMPILATION ORDER



A CORRECT ORDER:

C1 C3 C2 R1 R2 R3 R4 R5 P1 P2 P3 P4 P5

NAME-SCOPE

DEFINITION: THE NAME-SCOPE OF A BLOCK IS THE REGION WITHIN WHICH DATA DECLARED IN THE BLOCK IS VISIBLE. THE NAME-SCOPE ENCOMPASSES THE ENTIRE CONTENTS OF THE BLOCK INCLUDING ALL BLOCKS NESTED WITHIN IT.

RULES

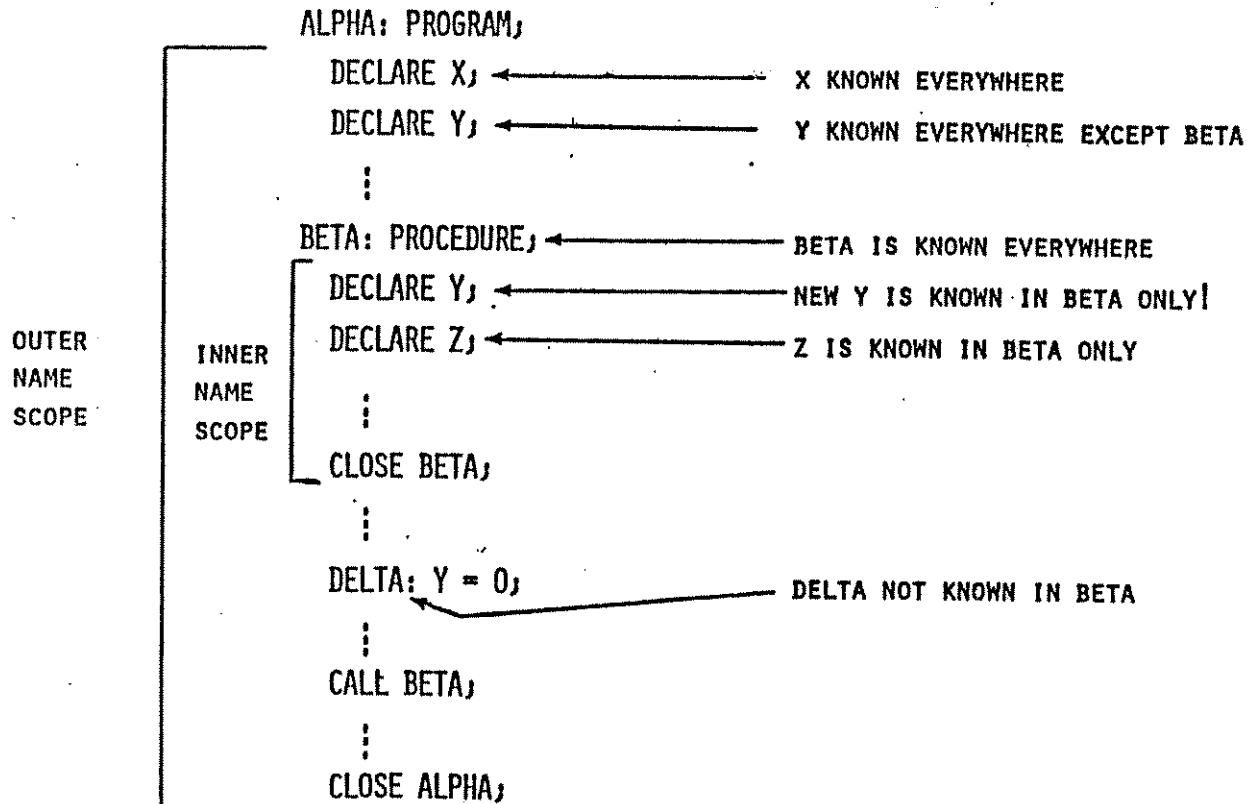
- (1) A NAME DEFINED IN A NAME-SCOPE IS KNOWN, AND THEREFORE ABLE TO BE REFERENCED, THROUGHOUT THAT NAME-SCOPE, INCLUDING ALL NESTED BLOCKS NOT REDEFINING IT. A NAME DEFINED IN A NAME-SCOPE IS NOT KNOWN OUTSIDE THAT NAME-SCOPE.
- (2) ALL COMPOOL DATA IS CONSIDERED TO BE DEFINED IN ONE NAME-SCOPE WHICH ENCLOSES THE OUTERMOST CODE BLOCK OF THE COMPILATION UNIT.

ALSO,

THE NAME OF A CODE BLOCK IS CONSIDERED TO BELONG TO THE NAME-SCOPE IMMEDIATELY ENCLOSING THE BLOCK.

NAME-SCOPE (CON'T.)

EXAMPLE:



NAME-SCOPE (CON'T.)

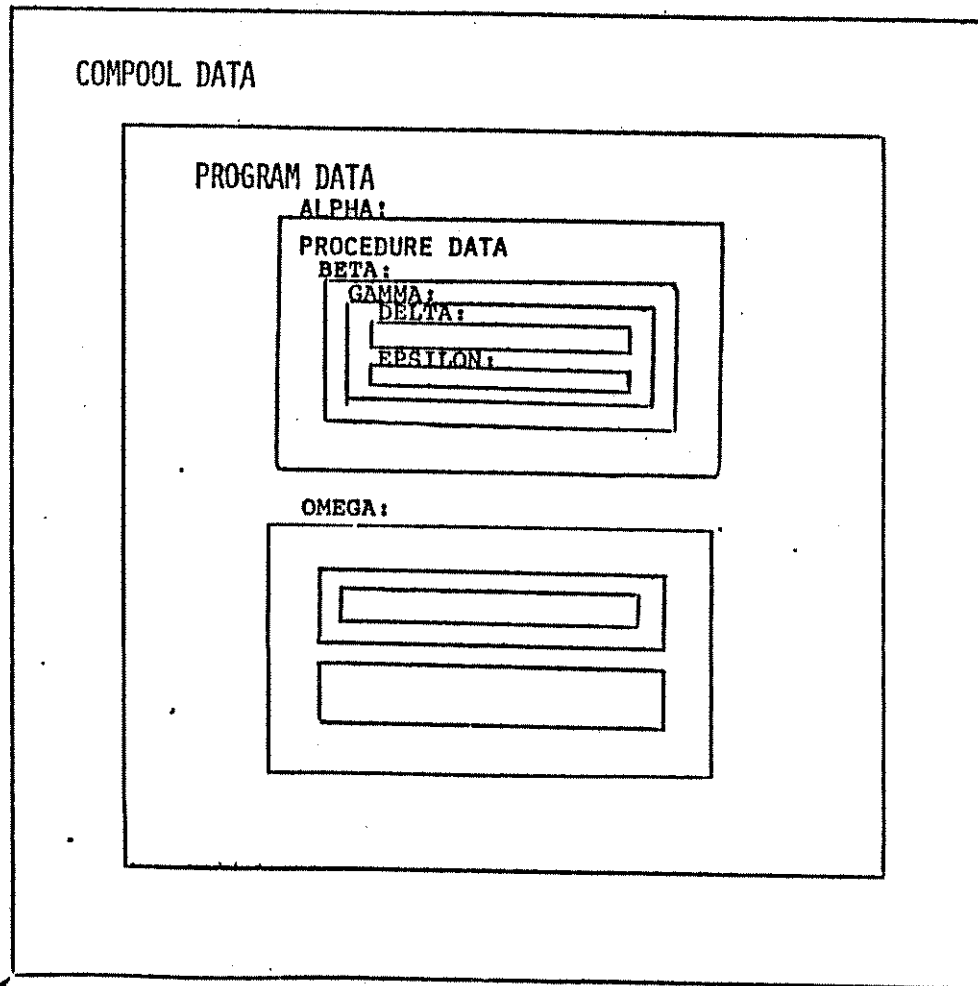
WHY IS BETA CONSIDERED TO BE IN THE NAME-SCOPE OF ALPHA?

BECAUSE ... IF IT WERE OTHERWISE

```
[ALPHA: PROGRAM;  
  DECLARE X;  
  DECLARE Y;  
  ⋮  
  [BETA: PROCEDURE;  
    ⋮  
  ] CLOSE BETA;  
  ⋮  
  CALL BETA; ← ILLEGAL -- NAME BETA IS NOT VISIBLE!!  
  ⋮  
] CLOSE ALPHA;
```

NAME-SCOPE (CON'T.)

VISUALIZE IT THIS WAY.....



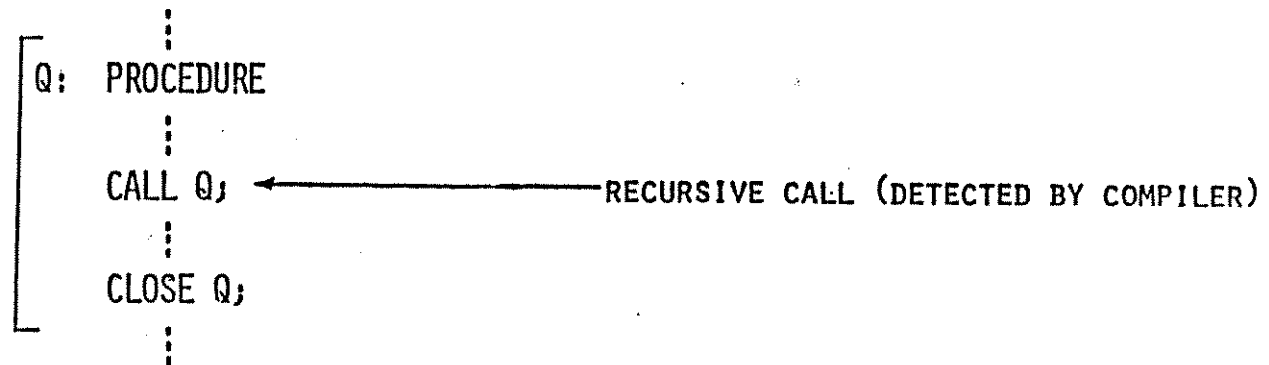
1-23

OUTERMOST NAME-SCOPE

NAME-SCOPE (CON'T.)

ALL RECURSION IS ILLEGAL IN HAL/S. OBVIOUS RECURSION IS DETECTED AT COMPILATION TIME. DEVIOS RECURSION IS DETECTED BY HALLINK OR AP-101 LINKAGE EDITOR DURING STACK CALCULATION.

EXAMPLE:



NAME-SCOPE (CON'T.)

FURTHER COMMENTS:

- (1) AS A GENERAL RULE, DATA MUST BE DECLARED (DEFINED) BEFORE IT CAN BE USED.
- (2) BLOCKS CAN BE CALLED FROM A POINT PRIOR TO THEIR DEFINITION, BUT WE RECOMMEND ALWAYS DEFINING A BLOCK BEFORE USING IT.
- (3) BLOCK LABELS MUST BE UNIQUE THROUGHOUT A UNIT OF COMPILATION.
- (4) AS AN EXCEPTION TO THE NAME-SCOPE RULES, STATEMENT LABELS ARE NOT VISIBLE WITHIN BLOCKS NESTED IN SCOPE WHERE LABEL IS DEFINED.

NAME-SCOPE (CON'T.)

THIS IS LEGAL,

⋮
CALL Q;

⋮
[Q: PROCEDURE;
⋮
CLOSE Q;

BUT WE PREFER,

⋮
[Q: PROCEDURE;
⋮
CLOSE Q)
⋮
CALL Q;

NAME-SCOPE (CON'T.)

WHY (?)

- (1) UNLIKE HAL/S, MANY LANGUAGES DO NOT ALLOW FORWARD REFERENCES.
- (2) A FORWARD REFERENCE USES UP A SYMBOL TABLE ENTRY.
- (3) FORWARD REFERENCES TO FUNCTIONS REQUIRE AN ADDITIONAL DECLARE:

```
    DECLARE F FUNCTION SCALAR,
```

```
        ⋮
```

```
        S = F(X),
```

```
        ⋮
```

```
    [ F: FUNCTION(ARG) SCALAR,
```

```
        ⋮
```

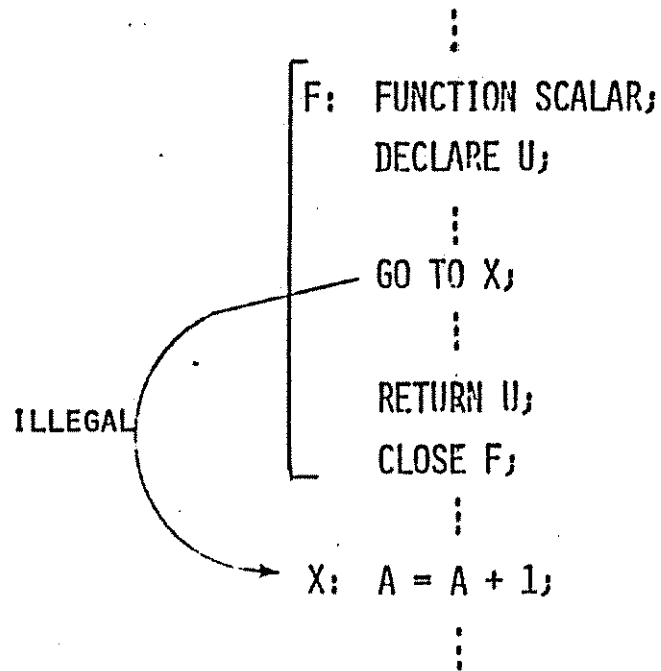
```
    ] CLOSE F;
```


NAME-SCOPE (CON'T.)

Q: WHY THE SPECIAL RULE FOR STATEMENT LABELS?

A: SO ONE CANNOT USE "GO TO" TO EXIT A BLOCK.

EXAMPLE



HAL/S PRIMITIVES
3 MAJOR CLASSES

A. RESERVED WORDS (NAMES WITH SPECIAL MEANING TO THE COMPILER)

(1) KEYWORDS

EXAMPLES: IF, ELSE, GO, TO, VECTOR, TASK, IN, LOCK,
TRUE, READ ...

(2) %-MACRO NAMES

EXAMPLES: %COPY, %SVC, %NAMECOPY

(3) BUILT-IN FUNCTION NAMES

EXAMPLES: ABS, SIGN, COS, LOG, UNIT, DET, RANDOM,
TRIM, SHR, XOR

B. IDENTIFIERS (NAMES INVENTED BY THE PROGRAMMER) -- LABELS AND
VARIABLES

EXAMPLES:

```
ALPHA: PROGRAM;  
  DECLARE (Q) SCALAR;  
  REPLACE (T) BY "6";
```

C. LITERALS (THINGS THAT EXPRESS THEIR OWN VALUE)

EXAMPLES: 6 4.95E3 'ABC'

PRIMITIVES (CON'T.)

A. RESERVED WORDS -

SEE APPENDICES B, C, AND I OF LANGUAGE SPECIFICATION.

B. IDENTIFIERS -

RULES:

- (1) MUST NOT CONFLICT WITH RESERVED WORDS!!
- (2) MUST BE 32 CHARACTERS OR LESS
- (3) FIRST CHARACTER MUST BE A-Z
- (4) BREAK CHARACTER '_' (UNDERSCORE) MAY BE USED -- BUT MUST NOT BE FIRST OR LAST CHARACTER.
- (5) ONLY ALPHANUMERIC CHARACTERS ARE LEGAL:

EXAMPLES:

LEGAL

STATE_VECTOR

Q174X203

B_A_CVAL

R

ILLEGAL

3xY

_PT1

VECTOR_

VECTOR

IN

A#B

Errors Detected:

<u>Error #</u>	<u>Cause</u>	<u>Fixup</u>
11	Argument outside range: $ X < \pi * 2^{50}$	Return 1
12	Argument too near a singularity of the tangent function	Return maximum positive floating point number

Comments:

Error gets very large near a singularity, before error #12 is sent.

The value used in the routine for $\pi * 250$ is hex '4DC90FDA' $\approx 3.53711870600810E+15$.

Registers Unsafe Across Call: F0,F1,F2,F3,F4,F5.

Algorithm:

Multiply X by $\frac{4}{\pi}$, and give the characteristic of this to X'0000000000000008' for use as a comparand to determine nearness to a singularity.

The integer part of $|X * \frac{4}{\pi}|$ is the octant.

If the octant is even, let $w =$ fraction part of $|X| * \frac{4}{\pi}$.

If the octant is odd, let $w = -(1 - \text{fraction})$ part of $|X| * \frac{4}{\pi}$.

Next, compute two polynomials P(w) and Q(w).

If $w \geq 2^{-46}$, then the forms of the polynomials are:

$$P(w) = w(a_0 + a_1 w^2 + a_2 w^4 + w^6)$$

$$Q(w) = b_0 + b_1 w^2 + b_2 w^4 + b_3 w^6$$

If $w < 2^{-46}$, then with $u = w$ if $|X| * \frac{4}{\pi} < 1$, and $u = -w$ otherwise.

$$P(w) = w(a_0 + u)$$

$$Q(w) = b_0 + b_3 u$$

where the values of the constants are:

$$a_0 = X'C58AFDD0A41992D4' = -569309.04006345$$

$$a_1 = X'44AFFA6393159226' = 45050.3889630777$$

$$a_2 = X'C325FD4A87357CAF' = -607.8306953515$$

$$b_0 = X'C5B0F82C871A3B68' = -724866.7829840012$$

$$b_1 = X'4532644B1E45A133' = 206404.6948906228$$

$$b_2 = X'C41926DBBB1F469B' = -6438.8583240077$$

PRIMITIVES (CON'T.)

C. LITERALS -

ARITHMETIC

(NO DISTINCTION MADE BETWEEN INTEGERS AND SCALARS).

EXAMPLES:

+6	-80,5E-6
9	4B30
3.0	-6H-4
-3.14159	3106245

GENERIC FORM:

±ddd.dddd<exponents>

SIGNS AND DECIMAL POINT OPTIONAL

B ~ POWER OF 2

E ~ POWER OF 10

H ~ POWER OF 16

PRIMITIVES (CON'T.)

C. LITERALS -

CHARACTER

TWO FORMS:

'CCCCCCC ... C'

OR

CHAR <repetition> 'CCC ... CC'

OPTIONAL

EXAMPLES:

NULL STRING

''

ABC

'ABC'

CHAR'ABC'

ABCABC

'ABCABC'

CHAR(2)'ABC'

ISN'T

'ISN''T'

QQQQQQ

CHAR(6)'Q'

'QQQQQQ'

NOTE: IF A SINGLE QUOTE IS DESIRED IN THE CHARACTER STRING, USE 2!
ALSO, CHARACTERS DO NOT HAVE TO BE ALPHANUMERIC NECESSARILY;

#, @, \$, ...

PRIMITIVES (CON'T.)

C. LITERALS -

BOOLEAN

TRUE = ON = BIN'1'

FALSE = OFF = BIN'0'

BIT

TRUE = ON = BIN'000 ... 01'

FALSE = OFF = BIN'000 ... 00'

BIN <repetition>'bbbb'	b = BINARY DIGIT
OCT <repetition>'oooo'	o = OCTAL DIGIT
HEX <repetition>'hhhh'	h = HEX DIGIT
DEC <repetition>'dddd'	d = DECIMAL DIGIT
<repetition> = (n)	OPTIONAL

EXAMPLES:

BIN'10110' → 10110₂

BIN(4)'101' → 101101101101₂

DEC(3)'9' → 999₁₀

HEX(7)'F' → FFFFFFFF₁₆

OCT'2037' → 2037₈

SOURCE FORMAT (CON'T.)

- (1) NOTE: IF CARD NUMBERS (SRNs) ARE PRESENT, OPTIONS 'SDL' OR 'SRN' MUST BE SPECIFIED TO THE COMPILER -- OTHERWISE, THE COMPILER WILL GO OUT TO COL. 80 LOOKING FOR TEXT.
- (2) ALTHOUGH THE STREAM-ORIENTED ASPECT OF THE HAL/S SCANNER ALLOWS MULTIPLE STATEMENTS ON ONE CARD, THIS IS NOT GENERALLY RECOMMENDED. IN FACT, SINCE AN "IF ... THEN" CONSTRUCTION IS CONSIDERED TO BE 2 SEPARATE STATEMENTS, THINGS GET SLIGHTLY COMPLICATED.
- (3) IT IS QUITE PERMISSABLE, ON THE OTHER HAND, TO EXTEND A SINGLE STATEMENT ACROSS MANY CARDS.
- (4) ENDS OF STATEMENTS ARE INDICATED BY A SEMICOLON (;).

SOURCE FORMAT (CON'T.)

DO THIS:

2	73
↓	↓
A = 2;	003100
B = 3;	003105

NOT THIS:

A = 2, B = 3;	003100
---------------	--------

THIS IS OK:

2	
↓	
A = Q + R + S + SIN(T)	004000
+ U + V W;	004005

AS LONG AS YOU DO NOT BREAK UP IDENTIFIERS AND RESERVED WORDS

THIS IS NOT:

A = CG1K_QVECTOR_	005000
FINAL + 6;	005005

ALSO, (MORE ON THIS LATER)

WE PREFER:

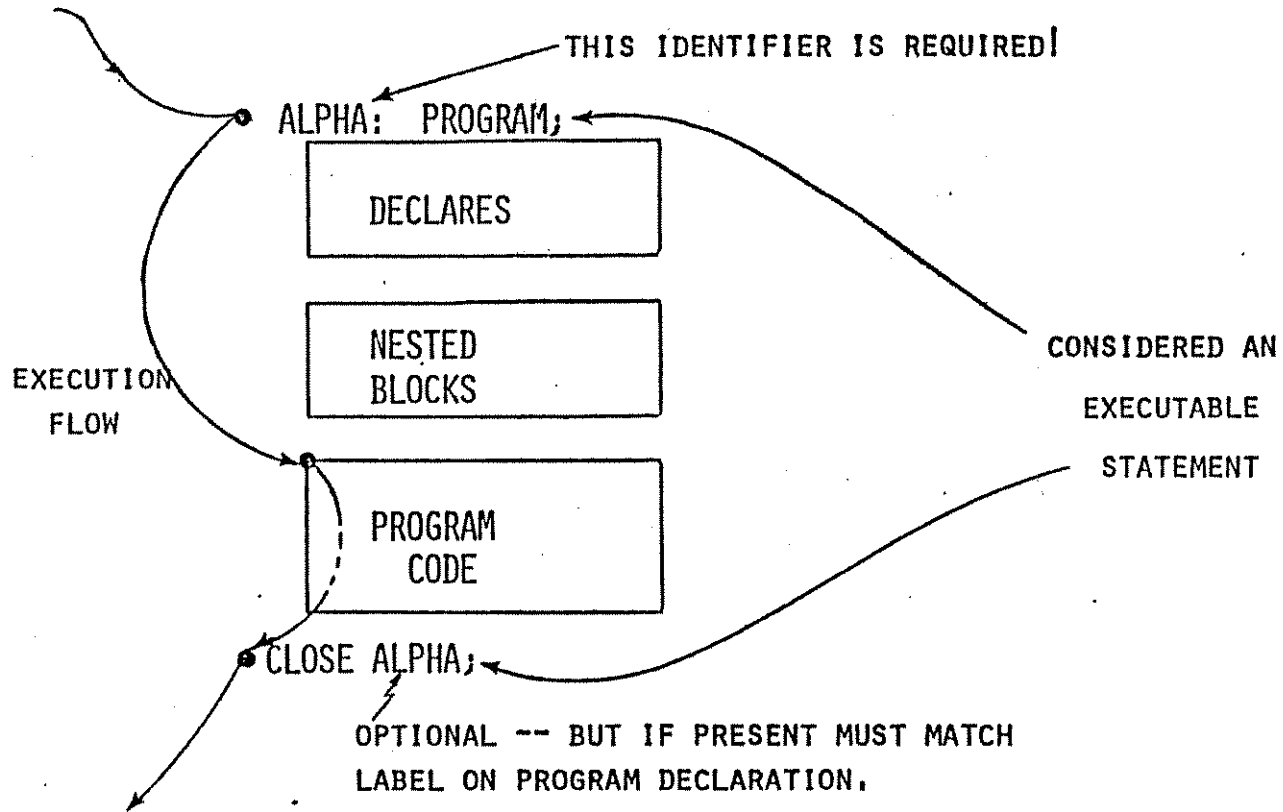
TO:

A, B, C = 0;

A = 0; B = 0; C = 0;

PROGRAM BLOCK

EXAMPLE:



NOTE: A PROGRAM CAN BE EXITED BY A "RETURN" STATEMENT -- BUT NORMALLY THIS IS ...

PROGRAM BLOCK (CON'T.)

... ACCOMPLISHED BY HITTING THE "CLOSE" STATEMENT.

ALSO, A "CLOSE" STATEMENT CAN HAVE A LABEL (AND CAN BE JUMPED TO VIA A "GO TO" STATEMENT)

EXAMPLE

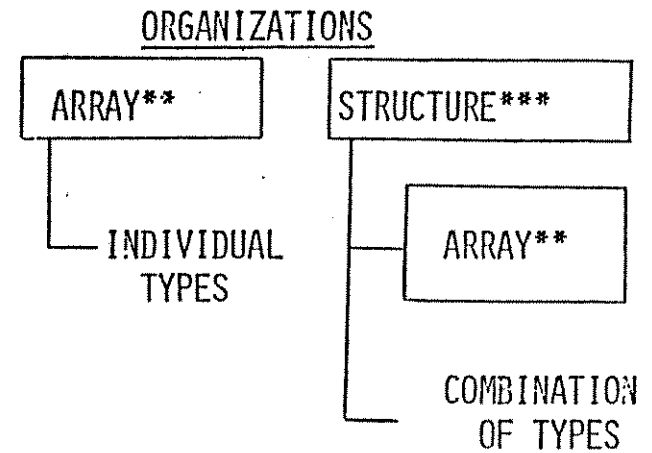
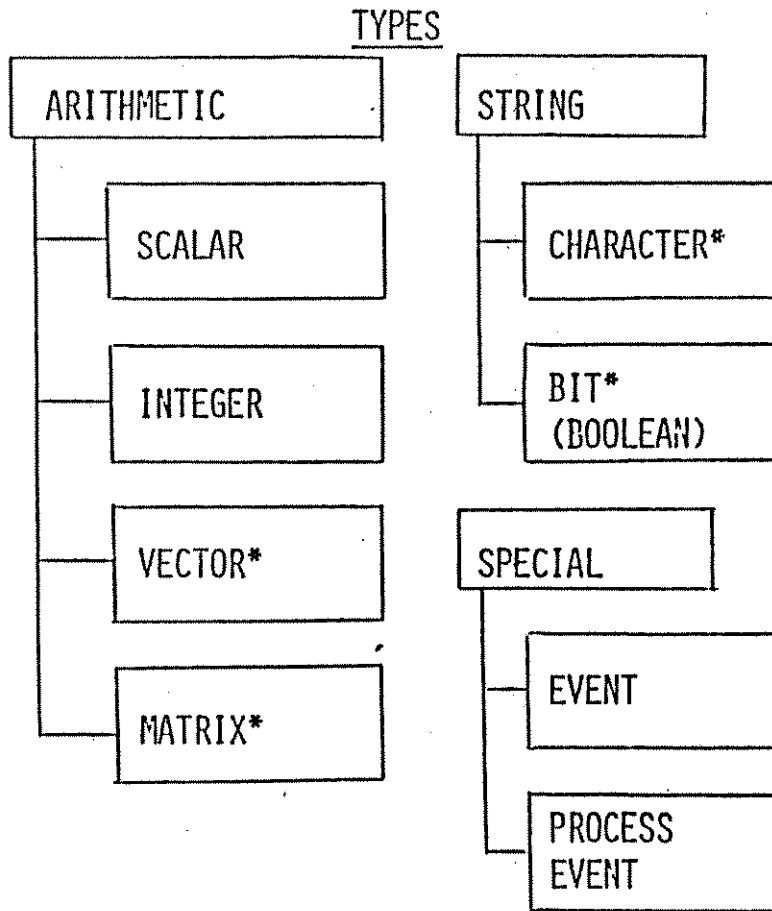
```
      ⋮
      IF A = 0 THEN RETURN;
      IF A = 2 THEN
      DO;
      B = B + 1;
      GO TO EXEUNT;
      END;
      B = SIN(C) + PI;
EXEUNT: CLOSE ALPHA;
```

EXIT VIA RETURN

EXIT VIA JUMP TO CLOSE!

EXIT BY FALLING INTO "CLOSE"

HAL/S DATA TYPES AND ORGANIZATIONS



- * - COMPONENT SUBSCRIPTING
- ** - ARRAY SUBSCRIPTING
- *** - STRUCTURE SUBSCRIPTING

DECLARES (CON'T.)

FOR THE TIME BEING WE WILL RESTRICT ATTENTION TO THE FOLLOWING
DATA TYPES:

ARITHMETIC

INTEGER

SCALAR

VECTOR

MATRIX

STRING

CHARACTER

BOOLEAN

AS FOR DATA ORGANIZATIONS, ONLY ARRAYS WILL BE CONSIDERED NOW.

A. INTEGERS

- ESSENTIALLY FIXED-POINT "WHOLE" NUMBERS (SIGNED)
- TWO CHOICES OF PRECISION

	360		FC
SINGLE	2 BYTES	≡	1 HALFWORD
DOUBLE	4 BYTES	≡	(2 HALFWORDS) 1 FULLWORD

DECLARES

(1) ALL DATA MUST BE DECLARED BEFORE IT CAN BE USED -- HAL/S HAS NO IMPLICIT DECLARATIONS.

(2) DATA IS DECLARED WITHIN A DECLARE GROUP WHICH FOLLOWS THE COMPOOL, PROGRAM, ETC; HEADER AND PRECEDES THE FIRST REAL EXECUTABLE STATEMENT.

NOTE (1): WITHIN A DECLARE GROUP, PARAMETERS SHOULD BE DECLARED FIRST.

NOTE (2): IT IS ALSO GOOD PROGRAMMING PRACTICE TO PLACE DATA DECLARED WITH THE CONSTANT ATTRIBUTE BEFORE OTHER DATA (THIS WILL BE DISCUSSED LATER UNDER THE SUBJECT OF COMPILE-TIME COMPUTATIONS).

DECLARES (CON'T.)

RANGES:

SINGLE $-32,768 \leq i \leq 32,767$

DOUBLE:

$-2,147,483,648 \leq i \leq 2,147,483,647$

USAGE:

SINGLE PRECISION INTEGERS ARE GENERALLY USED FOR LOOP VARIABLES.

CAUTION:

IT IS COMMONLY OBSERVED THAT USERS OVERESTIMATE THE RANGE OF A SHORT INTEGER, E.G.,

```
    DECLARE I INTEGER;  
    .  
    DO FOR I = 1 TO 99999,  
    . . .  
    END;
```


DECLARES (CON'T.)

NOTE: ALTHOUGH HAL/S DOES NOT HAVE IMPLICIT DECLARATIONS,
IT DOES HAVE DEFAULTS:
(1) DEFAULT PRECISION IS
 SINGLE
(2) DEFAULT DATA TYPE IS
 SCALAR

INTEGER DECLARATIONS

{ DECLARE I INTEGER; }
{ DECLARE I INTEGER SINGLE; }

EQUIVALENT FORMS -- YIELDS SHORT INTEGER

DECLARE I INTEGER DOUBLE;

NOTE: ORDER COUNTS, YOU CANNOT SAY
 DECLARE I DOUBLE INTEGER;

DECLARES (CON'T.)

B. SCALARS

- ESSENTIALLY FLOATING-POINT (SCIENTIFIC NOTATION) NUMBERS
- TWO CHOICES OF PRECISION

	360			FC
SINGLE	4 BYTES	=	{	2 HALFWORDS
			}	1 FULLWORD
DOUBLE	8 BYTES	=		2 FULLWORDS

RANGES:

BOTH SINGLE AND DOUBLE HAVE A DYNAMIC RANGE FROM
 $N \times 10^{-79}$ TO $N \times 10^{75}$

ACCURACY:

SINGLE	~	7 DECIMAL DIGITS
DOUBLE	~	17 (OR LESS)

DECLARES (CON'T.)

{ DECLARE S;
 DECLARE S SCALAR;
 DECLARE S SCALAR SINGLE; }

EQUIVALENT -- ALL YIELD SHORT SCALAR

{ DECLARE S DOUBLE;
 DECLARE S SCALAR DOUBLE; }

EQUIVALENT -- ALL YIELD LONG SCALAR

NOTE: MATRICES AND VECTORS ARE CONSIDERED TO BE
MADE UP OF SCALARS.

DECLARES (CON'T.)

C. VECTORS

- TWO CHOICES OF PRECISION
- LENGTHS MAY RANGE FROM

$$2 \leq L \leq 64$$

NOTE: DEFAULT LENGTH IS 3

VECTOR DECLARATIONS.

DECLARE V VECTOR;
DECLARE V VECTOR(3);
DECLARE V VECTOR SINGLE;

}
}

SINGLE-PRECISION 3-VECTORS

DECLARE V VECTOR DOUBLE;
DECLARE V VECTOR(64) DOUBLE;
DECLARE V VECTOR(2) DOUBLE;

NOTE: HAL/S DOES NOT DISTINGUISH BETWEEN (NOR IS THERE A
NEED TO) ROW AND COLUMN VECTORS.

--- MORE ON THIS LATER ---

DECLARES (CON'T.)

D. MATRICES

- TWO CHOICES OF PRECISION
- ROW AND COLUMN LENGTHS MAY RANGE FROM
 $2 \leq L \leq 64$

NOTE: DEFAULT ROW AND COLUMN LENGTHS ARE 3.

MATRIX DECLARATIONS

DECLARE M MATRIX;
DECLARE M MATRIX(3,3);
DECLARE M MATRIX SINGLE;

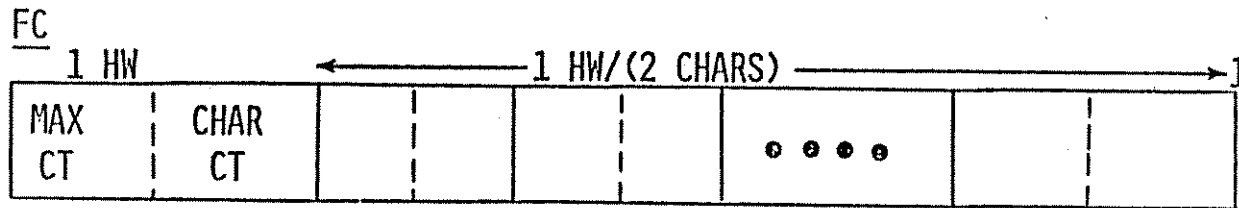
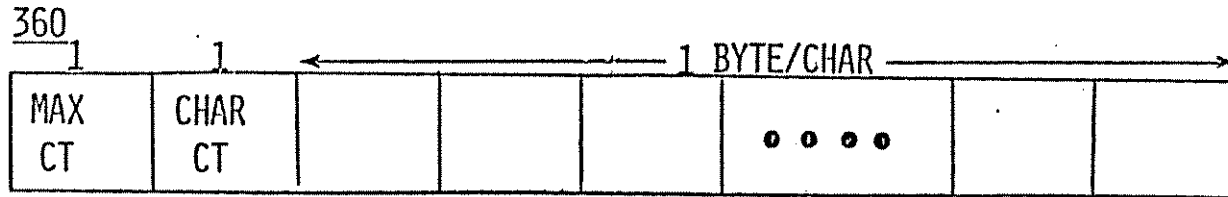
} SINGLE-PRECISION 3x3 MATRICES

DECLARE M MATRIX DOUBLE;
DECLARE M MATRIX(2,64) DOUBLE;
DECLARE M MATRIX(2,2) DOUBLE;

DECLARES (CON'T.)

E. CHARACTER STRINGS

PHYSICAL FORMAT:



POSSIBLY AN
UNUSED SLOT

CHARACTER DECLARATIONS

DECLARE STRING CHARACTER(4);

NOTES: (1) DECLARED LENGTH OF CHARACTER STRING MAY RANGE FROM

1 TO 255

(2) ACTUAL LENGTH MAY RANGE FROM

0 (NULL) TO 255

DECLARES (CON'T.)

F. BOOLEANS

- ESSENTIALLY DEGENERATE BIT STRINGS (BIT(1))
- ONLY TWO POSSIBLE VALUES

TRUE ≡ ON ≡ BIN'1'

FALSE ≡ OFF ≡ BIN'0'

BOOLEAN DECLARATIONS

DECLARE BOOL BOOLEAN;

NOTE: ALTHOUGH WE WILL DEFER DISCUSSION OF BIT STRINGS,
NOTE THAT IT IS ENTIRELY EQUIVALENT TO SAY,

DECLARE BOOL BIT(1);

ARRAYS

INTEGERS, SCALARS, VECTORS, MATRICES, CHARACTERS, AND BOOLEANS
MAY BE ARRAYED (OR POSSESS ARRAYNESS).

TO CREATE AN ARRAY, INSERT

ARRAY(N ₁)	1-DIM
ARRAY(N ₁ ,N ₂)	2-DIM
ARRAY(N ₁ ,N ₂ ,N ₃)	3-DIM

FOLLOWING THE IDENTIFIER IN THE DECLARE STATEMENT.

EXAMPLE:

```
DECLARE QMAT ARRAY(100) MATRIX;  
DECLARE B ARRAY(50) BOOLEAN;  
DECLARE S ARRAY(10,5) MATRIX(16,16) DOUBLE;
```

NOTE: ARRAY(.) MUST PRECEDE THE ATTRIBUTES.

ILLEGAL

```
DECLARE B BOOLEAN ARRAY(3);
```


ARRAY (CON'T.)

IN THE FORMS:

ARRAY(N₁)

ARRAY(N₁, N₂)

ARRAY(N₁, N₂, N₃)

$$2 \leq N_1 \leq 32,767$$

$$2 \leq N_2 \leq 32,767$$

$$2 \leq N_3 \leq 32,767$$

THE BIGGEST DATA ITEM WE CAN THINK OF AT THIS POINT IS:

DECLARE A ARRAY(32767, 32767, 32767) MATRIX(64, 64) DOUBLE;
= 1.15×10^{18} BYTES

INTEGERS AND SCALARS

-COMMENTS-

HAL/S UTILIZES CONTEXT TO DECIDE WHETHER LITERALS ARE INTEGERS OR SCALARS. IN GENERAL, WHENEVER A USER HAS A SCALAR THAT IS INTEGRAL IN FORM, WE ENCOURAGE WRITING IT AS THOUGH IT WERE AN INTEGER.

EXAMPLES:

IF...

DECLARE A, B, C;

THEN WRITE

A = 2 B;

INSTEAD OF

A = 2.0 B;

IN OTHER WORDS, ONE DOES NOT NEED TO APPEND ".0" TO WHOLE-NUMBER SCALARS.

INTEGERS AND SCALARS

-COMMENTS-

HAL/S IS VERY FORGIVING ABOUT MIXING INTEGERS (SINGLE OR DOUBLE) AND SCALARS (SINGLE OR DOUBLE) IN EXPRESSIONS. IMPLICIT CONVERSIONS ARE AUTOMATICALLY PERFORMED SO AS TO MAINTAIN MAXIMUM ACCURACY. OF COURSE, SINCE THESE CONVERSIONS AFFECT CPU AND CORE, USERS SHOULD BE AWARE OF THE POTENTIAL COST THAT CAN RESULT FROM THIS FREEDOM. (MORE ON THIS WHEN WE DISCUSS EXPRESSIONS AND ASSIGNMENTS.)

COMPOUND DECLARATIONS

SIMPLE DECLARATION

```
DECLARE S;  
DECLARE M MATRIX DOUBLE;  
DECLARE C CHARACTER(6);
```

COMPOUND DECLARATION

```
DECLARE S,  
        M MATRIX DOUBLE,  
        C CHARACTER(6);
```

(OR, ON ONE CARD)

```
DECLARE S, M MATRIX DOUBLE, C CHARACTER(6);
```

NOTE: IN A COMPOUND DECLARATION THE KEYWORD DECLARE APPEARS ONCE
AND INDIVIDUAL ELEMENTS ARE DELIMITED BY ",",.

FACTORED DECLARATIONS

IF THE IDENTIFIERS IN A COMPOUND DECLARATION HAVE SOME ATTRIBUTES
IN COMMON, A FACTORED FORM MAY BE EMPLOYED:

THUS, THE PROGRESSION IS:

SIMPLE DECLARE V1 VECTOR(4) INITIAL(0);
 DECLARE V2 VECTOR(4) INITIAL(0);
 DECLARE V3 VECTOR(4) INITIAL(0);

TO

COMPOUND DECLARE V1 VECTOR(4) INITIAL(0),
 V2 VECTOR(4) INITIAL(0),
 V3 VECTOR(4) INITIAL(0);

TO

FACTORED DECLARE VECTOR(4) INITIAL(0), V1, V2, V3;

THIS IS A BIG KEYPUNCH SAVER:

```
DECLARE ARRAY(16) MATRIX(4, 4) DOUBLE  
INITIAL(0), M1, M2, M3;
```

NOTE: THE COMMA MUST BE PRESENT!

DATA INITIALIZATION

DECLARE ORDER:

DECLARE a <array> <type> <precision> <initialization>

UNARRAYED DATA

INTEGER:

DECLARE I INTEGER INITIAL(0);

DECLARE J INTEGER DOUBLE

INITIAL(-20);

DECLARE K INTEGER INITIAL(4**3);

SCALAR:

DECLARE S INITIAL(0);

NOTE: DO NOT NEED 0.0!

DECLARE PI SCALAR DOUBLE

CONSTANT(3.14159265);

NOTES:

INITIAL AND CONSTANT BOTH CAUSE INITIALIZATION OF DATA. BUT SOMETHING DECLARED WITH THE CONSTANT ATTRIBUTE CANNOT EVER BE CHANGED.

DATA INITIALIZATION (CON'T.)

ALSO, DATA DECLARED CONSTANT "MAY" TURN OUT TO BE INACCESSIBLE TO RUN-TIME DIAGNOSTICS.

ON THE OTHER HAND, SOME DATA DECLARED CONSTANT, SINCE THE VALUE IS KNOWN BY THE COMPILER, MAY BE USED IN COMPILE-TIME EXPRESSIONS.

	REFERENC- ABLE	ASSIGN- ABLE	DIAGNOSTICS (I.E. DUMP)	COMPILE-TIME EXP.
<u>CONSTANT</u>	YES	NO	MAYBE	MAYBE
<u>INITIAL</u>	YES	YES	YES	NO

NOW, TO BE MORE PRECISE!

IF UNARRAYED INTEGERS, SCALARS, BIT STRINGS, OR CHARACTER STRINGS, ARE DECLARED CONSTANT, THEN THESE ITEMS CAN BE UTILIZED IN COMPILE-TIME EXPRESSIONS -- AT THE SAME TIME THIS IMPLIES THAT THE DATA ITEM IS UNAVAILABLE TO DIAGNOSTICS.

*** NOTE THAT VECTORS AND MATRICES ARE INELIGIBLE!

DATA INITIALIZATION (CON'T.)

THE REASON THESE ITEMS ARE UNAVAILABLE TO DIAGNOSTICS IS THAT THEY ARE PUT IN A SPECIAL LITERAL AREA BY THE COMPILER -- SUCH ITEMS DO NOT OCCUPY STORAGE WHERE THEY ARE DECLARED.

FINALLY, USE OF CONSTANT ALLOWS A GREATER RANGE OF COMPILER OPTIMIZATION:

EXAMPLE 1

```
DECLARE PI SCALAR DOUBLE
      CONSTANT(3.1415926535);
DECLARE RAD_TO_DEG SCALAR
      DOUBLE CONSTANT(180/PI);
DECLARE SIN15 SCALAR
      CONSTANT(SIN(15/RAD_TO_DEG));
```

EXAMPLE 2

```
DECLARE V1 VECTOR CONSTANT(1, 1, 1);
DECLARE V2 VECTOR CONSTANT(V1);
ILLEGAL!
```


DATA INITIALIZATION (CON'T.)

EXAMPLE 3

```
DECLARE K1 INTEGER CONSTANT(3);  
DECLARE S1 SCALAR CONSTANT(16.5);
```

```
⋮
```

```
W = SQRT(S1) ** K1;
```

THE EXPRESSION "SQRT(S1) ** K1" WILL BE EVALUATED AT COMPILE TIME. THE CODE WILL LOOK LIKE:

```
LE    0, { COMPILER_CALCULATED }  
      {          CONSTANT          }  
STE   0, W
```

DATA INITIALIZATION (CON'T.)

BOOLEAN INITIALIZATION

EQUIVALENT

{
 DECLARE BOOL BOOLEAN INITIAL(ON);
 DECLARE BOOL BOOLEAN INITIAL(BIN'1');
 DECLARE BOOL BOOLEAN INITIAL(TRUE);

EQUIVALENT

{
 DECLARE BOOL BOOLEAN INITIAL(OFF);
 DECLARE BOOL BOOLEAN INITIAL(BIN'0');
 DECLARE BOOL BOOLEAN INITIAL(FALSE);

AND OF COURSE;

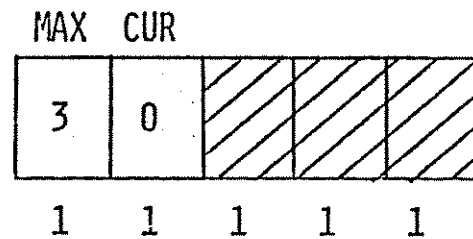
 DECLARE BOOL BOOLEAN CONSTANT(TRUE);
(ALTHOUGH THIS HARDLY MAKES SENSE)

DATA INITIALIZATION (CON'T.)

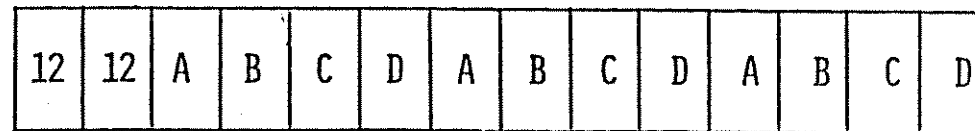
CHARACTER INITIALIZATION

- (1) DECLARE C CHARACTER(3) INITIAL('');
- (2) DECLARE C CHARACTER(12) INITIAL(CHAR(3)'ABCD');
- (3) DECLARE C CHARACTER(255) INITIAL(CHAR(255)'A');

IN (1) WE GET (ON THE 360)



IN (2) WE GET (ON THE 360)



I.E., A CHARACTER STRING REQUIRES 2 BYTES PLUS 1 BYTE PER CHARACTER.

DATA INITIALIZATION (CON'T.)

MULTI-VALUED DATA ITEMS

- A VECTOR OR MATRIX IS CONSIDERED TO BE A MULTI-VALUED DATA ITEM.
- ARRAYS ARE OBVIOUSLY MULTI-VALUED.
- IMPORTANT NOTE -- HAL/S MAKES LIFE EASY IF AN ENTIRE MULTI-VALUED ITEM IS TO BE INITIALIZED TO A SINGLE VALUE:

```
DECLARE Q MATRIX(16, 16) INITIAL(0);
```

```
DECLARE R ARRAY(400) SCALAR DOUBLE INITIAL(6);
```

- IF THE FOREGOING IS NOT SUITABLE THEN YOU MUST SUPPLY THE REQUISITE NUMBER OF DATA ITEMS.

VECTOR INITIALIZATION

```
DECLARE X_VECT VECTOR CONSTANT(1, 0, 0);
```

```
DECLARE R_VECT VECTOR DOUBLE INITIAL(6, -3.5, 44.82);
```


DATA INITIALIZATION (CON'T.)

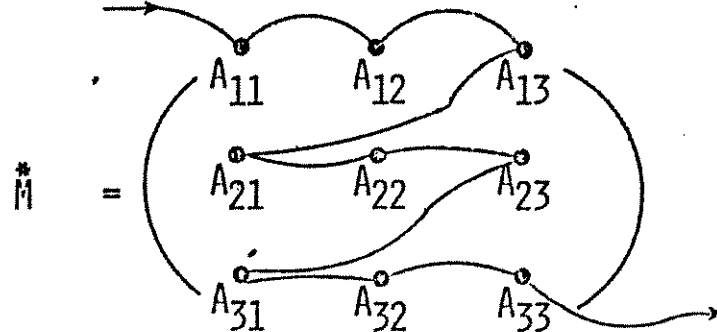
INITIALIZATION ORDER:

VECTOR -- BY INCREASING INDEX

MATRIX -- ROW-BY-ROW

ARRAY -- ELEMENT-BY-ELEMENT IN ORDER OF INCREASING INDEX,
IF AN ELEMENT IS MULTI-VALUED IT IS INITIALIZED
IN FULL BEFORE GOING TO THE NEXT ELEMENT.

MATRIX ORDER EXAMPLE:



I.E., THE MATRIX IS STORED IN CORE AS
[A11 A12 A13 A21 A22 A23 A31 A32 A33]

REPLACE STATEMENT

THERE ARE TWO KINDS OF REPLACE STATEMENTS: SIMPLE AND PARAMETERIZED
SYNTAX

SIMPLE:

REPLACE IDENTIFIER BY "ANY TEXT";

Ⓜ MUST BE PRESENT

EXAMPLES: (LEGAL USAGES)

(1) REPLACE PREC BY "SINGLE";

DECLARE SCALAR PREC, A, B, C;

(2) REPLACE S BY "A + B + LOG(C)";

DECLARE SCALAR, A, B, C, D;

D = S;

(3) REPLACE DV BY "VECTOR DOUBLE INITIAL(0)";

DECLARE VEC1 DV,
VEC2 DV,
VEC3 DV;

(4) REPLACE N BY "4";

DECLARE V1 VECTOR(N),
M1 MATRIX(N, N),
M2 MATRIX(2, N);

REPLACE STATEMENTS (CON'T.)

(ILLEGAL USAGES)

- (5) REPLACE SINGLE BY "DOUBLE";
 ↑
 └ RESERVED WORD (KEYWORD)
- (6) REPLACE SIN BY "SINH";
 ↑
 └ RESERVED WORD (BUILT-IN FUNCTION)
- (7) REPLACE + BY "/"; MISSING " "
 ↑
 └ NOT AN IDENTIFIER
- (8) REPLACE 16.2E3 BY "18.4E3";
 ↑
 └ ARITHMETIC LITERAL (NOT AN IDENTIFIER)

REPLACE STATEMENTS RESULT IN SOURCE-LEVEL SUBSTITUTION. THE ACTUAL SUBSTITUTIONS CAN BE SEEN IN THE LISTING IF CENT (¢) SIGNS ARE USED AROUND THE REPLACE NAME, I.E.,

D = ¢S¢;

WILL RESULT IN

D = A + B + LOG(C);

IN THE LISTING.


```

0      1      2      3      4      5      6      7
1234567890123456789012345678901234567890123456789012345678901.
EQUATE EXTERNAL FICDBF2P TO CZ1B_C_DIT_MSG_HEADRS(2:);
EQUATE EXTERNAL FICDBF3P TO CZ1B_C_DIT_MSG_HEADRS(3:);
EQUATE EXTERNAL FICDBF1B TO CZ1B_D_DEL_MSG_HDRS(1:);
EQUATE EXTERNAL FICDBF2B TO CZ1B_D_DEL_MSG_HDRS(2:);
EQUATE EXTERNAL FICDBF3B TO CZ1B_D_DEL_MSG_HDRS(3:);
C**
C** SYSTEM CONTROL
C**
C** DECLARE CZ1V_A_TSIP SCALAR DOUBLE INITIAL(0);
C** DECLARE CZ1V_A_BFCS_ENGAGE INTEGER INITIAL(0);
C** DECLARE CZ1B_A_TM_ET_STAT BIT(16) INITIAL(HEX'0000');
C**
C** EQUATES FOR FCS
C**
C** EQUATE EXTERNAL FIOBFCS TO CZ1V_A_BFCS_ENGAGE;
C**
C** GNC
C**
C** DECLARE CZ1V_G_SUM_WD INTEGER ;
C**
C** DECLARE CZ1B_V_VALID_CFS BOOLEAN;
C** DECLARE CZ1B_V_HQA BOOLEAN INITIAL(OFF);
C** CHECKOUT SOFTWARE ACTIVE FLAG
C**
C** DECLARE CZ1B_R_D_RDYFLG BOOLEAN INITIAL(OFF);
C** STATUS FROM LANDING SEP TO USER INTERFACE
C**
C** CLOSE CZ1 COMMS;
C** NAME= CZ1ACOM , CONCLUDE= CO, NE= RVL=AN,
C** CODE WAS 00000000
C**
DATE.

```

REPLACE STATEMENTS (CON'T.)

NOTES:

- (1) A REPLACE STATEMENT TAKES EFFECT ONLY AFTER IT APPEARS.
- (2) A REPLACE STATEMENT ADHERES TO NAME-SCOPE RULES.
- (3) A REPLACE CAN CAUSE TROUBLE BECAUSE IT CAN ONLY BE OVER-
RIDDEN BY ANOTHER REPLACE.

EXAMPLES:

(1) DECLARE V1 VECTOR(N);
REPLACE N BY "4";
DECLARE V2 VECTOR(N);

*** ERROR UNDECLARED IDENTIFIER



(2) ALPHA: PROGRAM;

REPLACE M BY "6";

...

CLOSE ALPHA;

SINCE REPLACE IS AT PROGRAM LEVEL, IT WILL BE
EFFECTIVE EVERYWHERE



REPLACE STATEMENTS (CON'T.)

(3) POTENTIAL TROUBLE

GAMMA: EXTERNAL COMPOOL,

. . .

REPLACE I BY "8",

. . .

CLOSE GAMMA,

ALPHA: PROGRAM,

DECLARE I INTEGER

. . .

DO FOR I = 1 TO 10,

. . .

END,

. . .

CLOSE ALPHA,

*** ERROR



REPLACE STATEMENTS (CON'T.)

REPLACE STATEMENTS CAN BE OVERRIDDEN.

EXAMPLE:

```
ALPHA: PROGRAM,  
  REPLACE X BY "6",  
  . . .  
  WRITE(X) A, B, C,  
  . . .  
BETA: PROCEDURE,  
  REPLACE X BY "7",  
  . . .  
  WRITE(X) A, B, C,  
  . . .  
CLOSE BETA,  
  . . .  
CLOSE ALPHA,
```


SUBSCRIPTING

HAL/S ALLOWS THREE DISTINCT TYPES OF SUBSCRIPTING: (ALL SUBSCRIPTING STARTS AT 1),

- (1) COMPONENT -- APPLICABLE TO STRINGS (I.E., BIT STRINGS* AND CHARACTER STRINGS) AS WELL AS VECTORS AND MATRICES.
- (2) ARRAY
- (3) STRUCTURE (DEFERRED UNTIL LATER)

* SINCE BOOLEANS ARE IN EFFECT DEGENERATE BIT STRINGS, USERS MUST CONSIDER THIS WHEN SETTING UP SUBSCRIPTS FOR ARRAYS OF BOOLEANS (MORE ON THIS LATER).

IN 1-LINE SOURCE FORMAT, SUBSCRIPTING IS INDICATED BY A "\$",

EXAMPLES: VECT\$7 W\$I MAT\$(2,4) X\$(I+2J+1)

NOTE THAT PARENTHESES ARE ONLY REQUIRED WHEN SUBSCRIPT IS MORE THAN ONE TOKEN!!!

SUBSCRIPTING (CON'T.)

52
208

THE OUTPUT WRITER OF PHASE 1, HOWEVER WILL ALWAYS PRINT SUBSCRIPTS (PARENTHESES REMOVED) ON AN "S" CARD:

EXAMPLE:

```
SOURCE:          I = Q$(2J + 1);
LISTING:  M      I = Q      ;
          S      2J + 1
```

COMPONENT SUBSCRIPTING

CHARACTER STRINGS

$$1 \leq L \leq 255$$

DECLARE STRING CHARACTER (L);

- TO SELECT ITH CHARACTER FROM STRING:

STRING\$I

WHERE I IS AN INTEGER EXPRESSION AND

$$1 \leq I \leq L$$

EXAMPLE:

```
STRING$(21)
STRING$3
STRING$(1**3-1)
```

SUBSCRIPTING (CON'T.)

SR
209

- TO SELECT 'I' CHARACTERS STARTING AT 'J':

STRING\$(I AT J)

WHERE I AND J ARE INTEGER EXPRESSIONS AND

$$1 \leq J \leq L$$

$$0 \leq I \leq L - J + 1$$

0 WILL PRODUCE A NULL STRING.

EXAMPLE:

STRING\$(3 AT K)

STRING\$(I**2 AT 2L)

- TO SELECT A SUBSTRING STARTING WITH THE ITH CHARACTER AND ENDING WITH THE JTH:

STRING\$(I TO J)

WHERE I AND J ARE INTEGER EXPRESSIONS AND

$$I \leq J$$

$$1 \leq I, J \leq L$$

SUBSCRIPTING (CON'T.)

EXAMPLES: STRING\$(I TO J)
 STRING\$(3 TO 4)
 STRING\$(I**2 TO 8)

MORE EXAMPLES

ASSUME:

```
DECLARE STRING CHARACTER(10) INITIAL('ABCDEFGHIJ'),  
DECLARE I INTEGER CONSTANT(2),  
          J INTEGER CONSTANT(8),  
          K INTEGER INITIAL(2);
```

. . .

- (A) STRING\$4 ≡ 'D'
- (B) STRING\$0 ILLEGAL
- (C) STRING\$11 ILLEGAL
- (D) STRING\$I ≡ 'B'
- (E) STRING\$(I**2) ≡ 'D'
- (F) STRING\$(K**4) RUN-TIME ERROR
- (G) STRING\$(I**4) COMPILE-TIME ERROR
- (H) STRING\$(COS(0)+1) ≡ 'B'
- (I) STRING\$(5 AT 3) ≡ 'CDEFG'
- (J) STRING\$(I AT J) ≡ 'HI'
- (K) STRING\$(J AT 3) ≡ 'CDEFGHIJ'

SUBSCRIPTING (CON'T.)

(L) STRING\$(5 TO 8) = 'EFGH'

(M) STRING\$(4 TO 3) ILLEGAL

VECTOR

ELEMENTS (WHICH ARE SCALARS OF COURSE) ARE INDEXED STARTING FROM 1

DECLARE V VECTOR(L), $2 \leq L \leq 64$

- TO SELECT ITH SCALAR FROM VECTOR:

V\$I (A SCALAR)

WHERE I IS AN INTEGER EXPRESSION AND

$$1 \leq I \leq L$$

- TO SELECT A SUB-VECTOR OF LENGTH I STARTING FROM THE JTH ELEMENT:

V\$(I AT J) (A VECTOR)

WHERE I IS AN INTEGER LITERAL AND

$$2 \leq I \leq L$$

AND J IS AN INTEGER EXPRESSION WITH

$$1 \leq J \leq L - I + 1$$

SUBSCRIPTING (CON'T.)

- TO SELECT A SUB-VECTOR (PARTITION) STARTING FROM THE ITH ELEMENT AND ENDING WITH THE JTH:

V\$(I TO J) (A VECTOR)

WHERE I, J ARE INTEGER LITERALS AND

$$1 \leq I < J \leq L$$

- Q. WHY IN THE ..AT.. AND ..TO.. SUBSCRIPTING FORMS, CAN THE LIMITS BE INTEGER EXPRESSIONS FOR CHARACTER STRINGS, BUT ARE REQUIRED TO BE INTEGER LITERALS FOR VECTORS?
- A. BECAUSE CHARACTER STRINGS IN HAL/S ARE INNATELY OF DYNAMIC LENGTH WHEREAS VECTOR LENGTHS MUST BE KNOWN ABSOLUTELY.

SUBSCRIPTING (CON'T.)

A. EXAMPLES OF LEGAL VECTOR SUBSCRIPTING

$$\text{LET } V = \begin{bmatrix} 2.0 \\ 3.5 \\ -1.4 \\ 6.9 \end{bmatrix}$$

(1) $V\$1 \equiv 2.0$ (SCALAR)

(2) $V\$4 \equiv 6.9$ (SCALAR)

(3) $V\$(2 \text{ AT } 2) \equiv \begin{bmatrix} 3.5 \\ -1.4 \end{bmatrix}$ (2-VECTOR)

(4) $V\$(1 \text{ TO } 3) \equiv \begin{bmatrix} 2.0 \\ 3.5 \\ -1.4 \end{bmatrix}$ (3-VECTOR)

IF ADDITIONALLY WE HAVE

DECLARE I INTEGER CONSTANT(3);

THEN THE FOLLOWING ARE ALSO VALID:

(5) $V\$(I \text{ AT } 2) \equiv \begin{bmatrix} 3.5 \\ -1.4 \\ 6.9 \end{bmatrix}$ (3-VECTOR)

(6) $V\$(1 \text{ TO } I) \equiv \begin{bmatrix} 2.0 \\ 3.5 \\ -1.4 \end{bmatrix}$ (3-VECTOR)

SUBSCRIPTING (CON'T.)

B. EXAMPLES OF ILLEGAL VECTOR SUBSCRIPTING

SUPPOSE:

```
DECLARE V VECTOR(5) INITIAL(2,3,4,6,8);
```

```
DECLARE I INTEGER INITIAL(4);
```

```
DECLARE J INTEGER CONSTANT(5);
```

```
DECLARE K INTEGER INITIAL(1);
```

I.E.,

$$V = \begin{bmatrix} 2 \\ 3 \\ 4 \\ 6 \\ 8 \end{bmatrix}$$

THEN THE FOLLOWING ARE ILLEGAL:

- (1) V\$0
- (2) V\$6
- (3) V\$(J+1) J+1 > LENGTH
- (4) V\$(K AT 1)
- (5) V\$(I TO K)

SUMMARY OF VECTOR COMPONENT SUBSCRIPTING:

- (I) COMPONENT SUBSCRIPTING OF A VECTOR RESULTS IN A SCALAR OR A SMALLER VECTOR;
- (II) IF SUBSCRIPTING RESULTS IN A VECTOR, ITS LENGTH MUST BE COMPUTABLE AT COMPILE TIME!!!

SUBSCRIPTING (CON'T.)

MATRIX

(MOST OF THE SUBSCRIPTING RULES ARE GENERALIZATIONS OF THE VECTOR RULES.)

- ① COMPONENT SUBSCRIPTING OF A MATRIX CAN RESULT IN A SCALAR, A VECTOR, OR A SUBMATRIX. IF A VECTOR OR MATRIX RESULTS, THE VECTOR LENGTH OR MATRIX ROW/COLUMN LENGTHS, RESPECTIVELY, MUST BE COMPUTABLE AT COMPILE TIME.

- ② MATRIX SUBSCRIPTING INVOLVES TWO DIMENSIONS: ROW AND COLUMN. AGAIN, ELEMENTS ARE INDEXED FROM 1.
 - TO EXTRACT THE SCALAR ELEMENT IN THE I^{TH} ROW AND J^{TH} COLUMN OF THE $M \times N$ MATRIX Q WHERE I, J ARE INTEGER EXPRESSIONS (I.E. NEED NOT BE KNOWN AT COMPILE TIME), AND $1 \leq I \leq M, 1 \leq J \leq N$:

$Q(I, J)$

SUBSCRIBING (CON'T.)

53
214

- TO SELECT THE I^{TH} ROW:
 $Q(I, *)$ (AN N -VECTOR)
- TO SELECT THE J^{TH} COLUMN:
 $Q(*, J)$ (AN M -VECTOR)
- TO SELECT A SUBMATRIX OF DIMENSIONS $\alpha \times \beta$:
 $Q(\alpha \text{ AT } I, \beta \text{ AT } J)$

EXAMPLES

LET Q BE THE 4×5 MATRIX:

$$\begin{bmatrix} 6 & 3 & 9 & 0 & 1 \\ 2 & 4 & 6 & 2 & 8 \\ 1 & 0 & 0 & 6 & 3 \\ 5 & 7 & 10 & 6 & 2 \end{bmatrix}$$

THEN

(1) $Q(1,1) \equiv 6$ (SCALAR)

(2) $Q(4,5) \equiv 2$ (SCALAR)

(3) $Q(2,*) \equiv \begin{bmatrix} 2 \\ 4 \\ 6 \\ 2 \\ 8 \end{bmatrix}$ (5-VECTOR)

SUBSCRIPTING (CON'T.)

53
2/7

AGAIN,

$$Q = \begin{bmatrix} 6 & 3 & 9 & 0 & 1 \\ 2 & 4 & 6 & 2 & 8 \\ 1 & 0 & 0 & 6 & 3 \\ 5 & 7 & 10 & 6 & 2 \end{bmatrix}$$

$$(4) \quad Q(\$(*,3)) = \begin{bmatrix} 9 \\ 6 \\ 0 \\ 10 \end{bmatrix}$$

$$(5) \quad Q\$(3 \text{ AT } 1,3) = \begin{bmatrix} 9 \\ 6 \\ 0 \end{bmatrix}$$

$$(6) \quad Q\$(2 \text{ TO } 3, 3 \text{ TO } 5) = \begin{bmatrix} 6 & 2 & 8 \\ 0 & 6 & 3 \end{bmatrix}$$

$$(7) \quad Q\$(*, 3 \text{ AT } 2) = \begin{bmatrix} 3 & 9 & 0 \\ 4 & 6 & 2 \\ 0 & 0 & 6 \\ 7 & 10 & 6 \end{bmatrix}$$

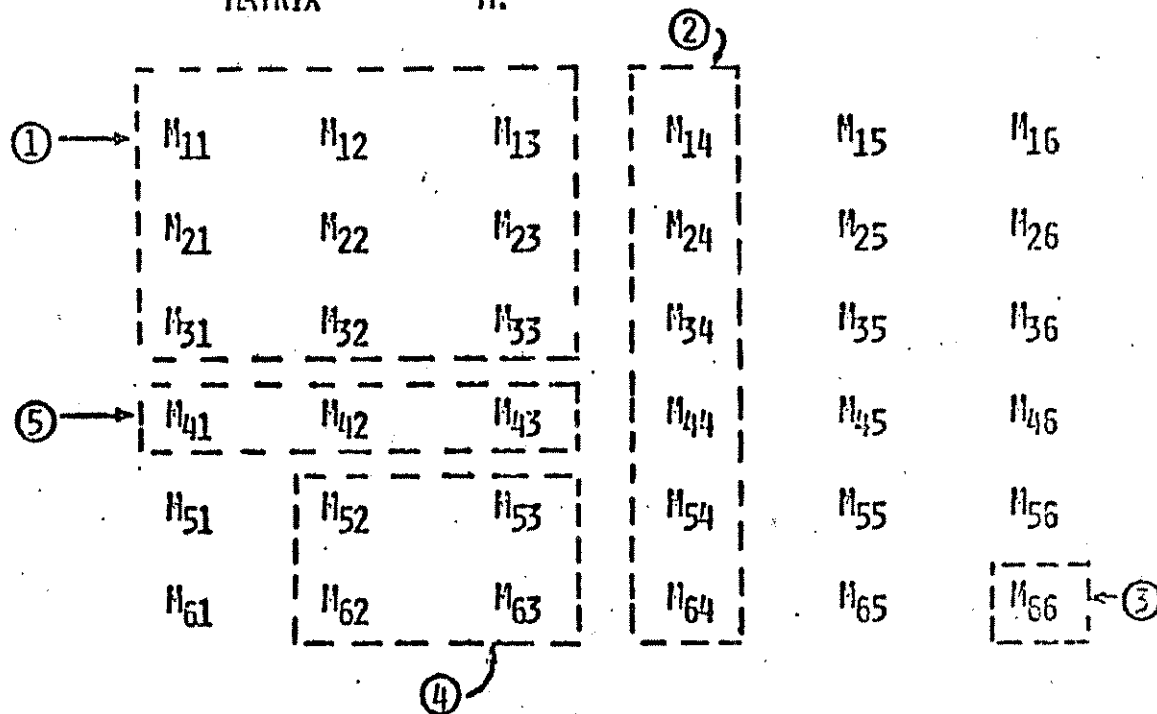
53
2/18

PARTITIONING

EXAMPLE:

MATRIX

\bar{M} :



- 1) \bar{M}_1 TO 3, 1 TO 3
- 2) $\bar{M}_{3,4}$
- 3) M_{65}
- 4) \bar{M}_5 TO 6, 2 TO 3

- 5) M_4 , 1 TO 3

32
219

ARRAY SUBSCRIPTING

- ARRAYS MAY BE OF 1, 2, OR 3 DIMENSIONS (WITH A MAXIMUM RANGE OF 1 TO 32767 IN EACH DIMENSION). HERE WE WILL ONLY CONSIDER SINGLE DIMENSION ARRAYS.
- ARRAY SUBSCRIPTING IS EASY FOR ARRAYS OF INTEGERS AND SCALARS SINCE NO COMPONENT SUBSCRIPTING IS POSSIBLE. FOR ARRAYS OF CHARACTER STRINGS, BIT STRINGS (AND BOOLEANS), VECTORS, AND MATRICES, LIFE IS MORE COMPLICATED.
- SUBSCRIPTING AN ARRAY CAN RESULT IN A SINGLE ELEMENT OR A SUB-ARRAY OF ELEMENTS. IN THE LATTER CASE (AS WAS TRUE FOR VECTOR AND MATRIX COMPONENT SUBSCRIPTING) THE ARRAYNESS (I.E., RANGES OF ALL ARRAY DIMENSIONS) MUST BE COMPUTABLE AT COMPILE TIME.

ARRAY SUBSCRIPTING (CON'T.)

LET "TABLE" BE AN ARRAY OF LENGTH L OF ANY DATA (INTEGER, SCALAR, BOOLEAN, CHARACTER, VECTOR, MATRIX, BIT STRING)

- TO SELECT THE ITH ARRAY ELEMENT:

TABLE\$(I:)
 ← PARENTHESES NEEDED BECAUSE "I" AND ":" ARE 2
 TOKENS

WHERE I IS AN INTEGER EXPRESSION AND

$$1 \leq I \leq L$$

NOTE: IF TABLE IS AN ARRAY OF INTEGERS OR SCALARS SO THAT NO COMPONENT SUBSCRIPTING IS POSSIBLE, THEN THE COLON MAY BE OMITTED, THUS:

TABLE\$I SUFFICES.

- TO SELECT A SUB-ARRAY OF LENGTH I STARTING AT THE JTH ARRAY ELEMENT OF TABLE:

TABLE\$(I AT J:)
 ↗ COLON OPTIONAL FOR INTEGER/SCALAR

WHERE I IS AN INTEGER LITERAL WITH

$$1 \leq I \leq L$$

AND J IS AN INTEGER EXPRESSION WITH

$$1 \leq J \leq L - I + 1$$

3

53
221

ARRAY SUBSCRIPTING (CON'T.)

- TO SELECT A SUB-ARRAY STARTING FROM THE ITH ARRAY ELEMENT AND ENDING WITH THE JTH:

TABLE\$(I TO J:)

OPTIONAL FOR INTEGER/SCALAR

WHERE I AND J ARE BOTH INTEGER LITERALS AND

$$1 \leq I < J \leq L$$

EXAMPLE ①

DECLARE S ARRAY(3) VECTOR(2) INITIAL(6, 9, 4, 2, 0, 8);

COMPONENT
OF ARRAY

→ S\$(1:) = $\begin{bmatrix} 6 \\ 9 \end{bmatrix}$

(AN UNARRAYED VECTOR)

PARTITION
OF ARRAY

→ S\$(2 AT 1:) = $\left(\begin{bmatrix} 6 \\ 9 \end{bmatrix}, \begin{bmatrix} 4 \\ 2 \end{bmatrix} \right)$

STILL AN ARRAY

→ S\$(2 TO 3:) = $\left(\begin{bmatrix} 4 \\ 2 \end{bmatrix}, \begin{bmatrix} 0 \\ 8 \end{bmatrix} \right)$

STILL AN ARRAY

NO COMPONENT SUBSCRIPTING HERE.

ARRAY SUBSCRIPTING (CON'T.)

EXAMPLE (2)

NO COLON
NEEDED

```
DECLARE A ARRAY(4) INTEGER INITIAL(9, 0, -6, 3);  
A$1 ≡ 9  
A$4 ≡ 3  
A$(1 TO 3) ≡ (9, 0, -6)  
A$(2 AT 2) ≡ (0, -6)
```

EXAMPLE (3)

```
DECLARE C ARRAY(3) CHARACTER(4) INITIAL('THIS', 'IS', 'HARD');  
C$(2:) ≡ 'IS'  
C$(2 AT 2:) ≡ ('IS', 'HARD')
```

EXAMPLE (4)

```
DECLARE BOOL ARRAY(5) BOOLEAN INITIAL(TRUE, FALSE, ON, OFF, BIN'O');  
BOOL$(4:) ≡ OFF  
BOOL$(3 AT 3:) ≡ (ON, OFF, BIN'O')  
BOOL$(1 TO 4:) ≡ (TRUE, FALSE, ON, OFF)
```

ARRAY AND COMPONENT SUBSCRIPTING

32
223

GENERIC FORM:

TABLE\$(ARRAY: COMPONENT)

IN OTHER WORDS, THE ":" IS USED TO ISOLATE THE ARRAY SUBSCRIPTS FROM THE COMPONENT SUBSCRIPTS.

EXAMPLE ①

DECLARE S ARRAY(3) VECTOR(2) INITIAL(6, 9, 4, 2, 0, 8);

S\$(1:2) ≡ 9 (A SCALAR)

S\$(2 AT 2:1) ≡ (4,0) AN ARRAY OF 2 SCALARS

EXAMPLE ②

DECLARE C ARRAY(3) CHARACTER(4) INITIAL('THIS', 'IS', 'HARD');

C\$(2:2) ≡ 'S'

C\$(3:3) ≡ 'R'

C\$(2 AT 1: 1 TO 2) ≡ ('TH', 'IS')

C\$(2 TO 3: 2) ≡ ('S', 'A')

ARRAY AND COMPONENT SUBSCRIPTINGS (CON'T.)

53
224

EXAMPLE (3)

DECLARE M ARRAY(3) MATRIX INITIAL(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
16,17,18,19,20,21,22,23,24,25,26,27);

I.E.,

$$M = \left(\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \begin{bmatrix} 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{bmatrix}, \begin{bmatrix} 19 & 20 & 21 \\ 22 & 23 & 24 \\ 25 & 26 & 27 \end{bmatrix} \right)$$

M\$(2: 2,1) = 13

M\$(3: 3,3) = 27

M\$(*: 3,3) = (9, 18, 27)

INDICATES DO FOR EACH ELEMENT OF ARRAY

M\$(2 AT 1: 2 AT 2, 2 AT 2) = $\left(\begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix}, \begin{bmatrix} 14 & 15 \\ 17 & 18 \end{bmatrix} \right)$

COMPONENT SUBSCRIPTING ONLY

52
225

IF THE DATA TO BE COMPONENT - SUBSCRIPTED (BIT AND CHARACTER STRING, VECTOR, OR MATRIX) IS ALSO ARRAYED, A TOKEN AMOUNT OF ARRAY SUBSCRIPTING MUST BE KEPT. IF TABLE IS AN ARRAYED DATA ITEM AND WE WISH TO PERFORM COMPONENT SUBSCRIPTING SIMULTANEOUSLY ON ALL ARRAY ELEMENTS THEN WE NEED THE FOLLOWING FORM:

TABLE\$(*: COMPONENT SUB)

WITH THIS FORM WE WILL PRODUCE A NEW ARRAY (WITH THE SAME NUMBER OF DIMENSIONS AND SAME RANGE IN EACH DIMENSION AS THE ORIGINAL) WITH EACH ELEMENT THEREOF BEING A COMPONENT-SUBSCRIPTED VERSION OF THE ORIGINAL ELEMENT.

EXAMPLE ①

DECLARE C ARRAY(3) CHARACTER(4) INITIAL('THIS', 'IS', 'HARD');

C\$(*:2) ≡ ('H', 'S', 'A')

225

COMPONENT SUBSCRIPTING ONLY (CON'T.)

53
226

EXAMPLE (2)

LET M BE AN ARRAY 3 OF 3 x 4 MATRICES:

$$M \equiv \left(\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}, \begin{bmatrix} 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \end{bmatrix}, \begin{bmatrix} 25 & 26 & 27 & 28 \\ 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 \end{bmatrix} \right)$$

THEN

$$M\$(2: 3, 4) \equiv 24$$

$$M\$(2 \text{ AT } 2: *, 3) \equiv \left(\begin{bmatrix} 15 \\ 19 \\ 23 \end{bmatrix}, \begin{bmatrix} 27 \\ 31 \\ 35 \end{bmatrix} \right)$$

$$M\$(1 \text{ TO } 3: 2 \text{ TO } 3, 3 \text{ TO } 4) \equiv \left(\begin{bmatrix} 7 & 8 \\ 11 & 12 \end{bmatrix}, \begin{bmatrix} 19 & 20 \\ 23 & 24 \end{bmatrix}, \begin{bmatrix} 31 & 32 \\ 35 & 36 \end{bmatrix} \right)$$

M\$(*: 2 TO 3, 3 TO 4) \equiv (THE SAME)

$$\left(\begin{bmatrix} 7 & 8 \\ 11 & 12 \end{bmatrix}, \begin{bmatrix} 19 & 20 \\ 23 & 24 \end{bmatrix}, \begin{bmatrix} 31 & 32 \\ 35 & 36 \end{bmatrix} \right)$$

2-20

SUBSCRIPTING SUMMARY

52
227

RECOMMENDATION: EVEN WHERE OPTIONAL, I.E., ARRAY SUBSCRIPTING ON INTEGERS
AND SCALARS, USE THE TRAILING "COLON".

- UNARRAYED DATA -

COMPONENT SUBSCRIPTING:

VAR\$(7 AT 1)

VAR\$3

VAR\$(4 TO 5, 3 TO 4)

- ARRAYED DATA -

NO COMPONENT SUBSCRIPTING:

VAR\$(7:) ← NEED PARENTHESES SINCE "7" AND ":" ARE 2 TOKENS.

VAR\$(3 AT 2:)

ARRAY AND COMPONENT SUBSCRIPTING:

VAR\$(7:3)

VAR\$(4 AT 1: 3 TO 4, 1 TO 2)

COMPONENT SUBSCRIPTING ONLY: (ARRAYNESS UNCHANGED)

VAR\$(*: 3 TO 4, 1 TO 2)

VAR\$(*: 7)

3/
228

MORE DATA INITIALIZATION

SUMMARY OF OLD MATERIAL:

- ① TO INITIALIZE A SINGLE-VALUED DATA ITEM, SUPPLY 1 LITERAL VALUE IN THE INITIAL/CONSTANT LIST:

```
DECLARE I INTEGER INITIAL(7);  
DECLARE S CONSTANT(9.3E-4);
```

- ② TO INITIALIZE A MULTI-VALUED DATA ITEM, (ARRAY, VECTOR, MATRIX) WE HAVE 3 CHOICES:

- A. INITIALIZE EVERYTHING TO THE SAME VALUE: (1 ITEM IN LIST)

```
DECLARE M MATRIX INITIAL(0);  
DECLARE A ARRAY(15000) INITIAL(6.5);
```

- B. INITIALIZE TO DIFFERENT VALUES: (NEED 1 LITERAL FOR EVERY ELEMENT)

```
DECLARE M MATRIX INITIAL(1,0,0,0,1,0,0,0,1);  
DECLARE A ARRAY(15000) INITIAL(3.5, 6.5, 3.5, 6.5, 3.5,  
6.5, 3.5, 6.5, 3.5, 6.5, ....., 6.5);
```

15000 OF THEM!!

51
229

MORE DATA INITIALIZATION (CON'T.)

C. IF THE DATA IS ARRAYED, SUPPLY EXACTLY ENOUGH LITERALS TO INITIALIZE A SINGLE ELEMENT OF THE ARRAY. IN THIS CASE ALL ELEMENTS OF THE ARRAY WILL BE INITIALIZED IDENTICALLY.

EXAMPLE:

INSTEAD OF

```
DECLARE MM ARRAY(3) MATRIX(2,2) INITIAL(1,0,0,1,1,0,0,1,1,0,0,1);
```

CODE

```
DECLARE MM ARRAY(3) MATRIX(2,2) INITIAL(1,0,0,1);
```

IN BOTH CASES

$$MM \equiv \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right)$$

2-27

MORE DATA INITIALIZATION (CON'T.)

52
230

- SHORT CUTS -

① USE OF REPETITION FACTOR #

EXAMPLES

```
DECLARE C ARRAY(6) CHARACTER(4) INITIAL(3#'ABC', 3#'DEF');  
C ≡ ('ABC', 'ABC', 'ABC', 'DEF', 'DEF', 'DEF')
```

```
DECLARE I4 MATRIX(4,4) INITIAL(1, 3#(0,0,0,0,1)),
```

```
I4 ≡ 
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```

I.E., A SEQUENCE OF LITERALS CAN BE REPEATED.

MORE DATA INITIALIZATION (CON'T.)

53
231

- REPEATED GROUPS CAN EVEN BE NESTED

```
DECLARE I4 MATRIX(4,4) INITIAL(1, 3#(4#0,1));
```

M ≡
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
DECLARE V ARRAY(3,2,2) INITIAL(1,2,3,2,3,1,2,3,2,3,1,2);
```



```
DECLARE V ARRAY(3,2,2) INITIAL(2#(1,2,3,2,3),1,2);
```



```
DECLARE V ARRAY(3,2,2) INITIAL(2#(1,2#(2,3)),1,2);
```


MORE DATA INITIALIZATION (CON'T.)

52
232

② PARTIAL INITIALIZATION

- SKIP OVER VALUES NOT TO BE INITIALIZED (JUST USE N#)

```
DECLARE M MATRIX(4,4) INITIAL(1, 3#(4#,1));
```

$$M = \begin{bmatrix} 1 & x & x & x \\ x & 1 & x & x \\ x & x & 1 & x \\ x & x & x & 1 \end{bmatrix}$$

x → NOT INITIALIZED

- LEAVE REMAINDER OF LIST UNINITIALIZED (USE OF * SYMBOL)

```
DECLARE M MATRIX(4,4) INITIAL(1,2,3,*);
```

$$M = \begin{bmatrix} 1 & 2 & 3 & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{bmatrix}$$

AUTOMATIC/STATIC INITIALIZATION

51
233

TO DESCRIBE THE WAY IN WHICH DATA INITIALIZATION IS EFFECTED, HAL/S HAS 2 INITIALIZATION ATTRIBUTE KEYWORDS THAT ARE OPPOSITE IN MEANING:

STATIC AND AUTOMATIC.

MOST DATA IS OF THE STATIC TYPE AND THIS IS THE DEFAULT. STATIC DATA IS INITIALIZED WHEN THE DATA MODULES ARE BROUGHT INTO CORE (I.E., THE DATA IS ALREADY EXISTENT IN THE LOAD MODULES). COMPOOLS ALWAYS CONSIST OF STATIC DATA.

THE KEYWORD STATIC CAN BE SPECIFIED BY THE USER, BUT THIS IS NOT RECOMMENDED SINCE IT CLUTTERS THE LISTING:

```
{ DECLARE I INTEGER INITIAL(5) STATIC;  
  DECLARE I INTEGER STATIC INITIAL(5);  
  DECLARE I INTEGER INITIAL(5);
```

ALL EQUIVALENT! NOTE THAT STATIC CAN PRECEDE OR FOLLOW THE INITIAL LIST.

5/
234

AUTOMATIC/STATIC INITIALIZATION (CON'T.)

AUTOMATIC DATA GENERALLY RESULTS IN EXECUTABLE CODE IN THE PROLOGUE OF CODE BLOCKS TO INITIALIZE THE DATA EACH TIME THE BLOCK IS ENTERED. FOR MULTI-VALUED DATA THIS CAN BE VERY EXPENSIVE -- AND AUTOMATIC, IN GENERAL, SHOULD BE SPECIFIED ONLY WHEN IT IS REALLY NEEDED. THIS WILL BE DISCUSSED IN MORE DETAIL LATER WHEN REENTRANCY IS CONSIDERED.

THE FOLLOWING ARE EQUIVALENT:

```
DECLARE I INTEGER AUTOMATIC INITIAL(5);  
DECLARE J INTEGER INITIAL(5) AUTOMATIC;
```

RESTRICTIONS ON USE OF STATIC/AUTOMATIC

- DATA INITIALIZED CONSTANT MAY NOT POSSESS EITHER THE STATIC OR AUTOMATIC KEYWORD.
- COMPOOL DATA MAY NOT HAVE EITHER A STATIC OR AN AUTOMATIC SPECIFICATION.

AUTOMATIC/STATIC INITIALIZATION (CON'T.)

SR
205

EXAMPLES

1 A: COMPOOL;
DECLARE I INTEGER INITIAL(4) STATIC; ILLEGAL
...
CLOSE A;

2 B: PROGRAM;
DECLARE M MATRIX INITIAL(0) STATIC; (DEFAULT) RECOMMEND NEVER SPECIFYING
... STATIC
CLOSE B;

- COMSUB EXAMPLE -

3 C: PROCEDURE;
DECLARE I INTEGER INITIAL(10);
... UPON FIRST CALL I = 10
I = 4; ON SECOND CALL I = 4
CLOSE C;

C: PROCEDURE;
DECLARE I INTEGER INITIAL(10) AUTOMATIC;
... I = 10 ALWAYS UPON ENTRY
I = 4;
...
CLOSE C;

(PROLOG OF COMSUB WILL CONTAIN CODE, I.E.:

LA RX, 10)
ST RX, I

5/
236

INTEGER/SCALAR CONVERSIONS

INTERPRETATION

- ① IN HAL/S EXPRESSIONS, ARBITRARY MIXTURES OF INTEGERS AND SCALARS (BOTH OF EITHER SINGLE OR DOUBLE PRECISION) CAN OCCUR.
- ② THE FOLLOWING CONVERSIONS ARE CHEAP (I.E., DONE VIA INLINE CODE)
SINGLE INTEGER ←————→ DOUBLE INTEGER
SINGLE SCALAR ←————→ DOUBLE SCALAR

ALL OTHER CONVERSIONS REQUIRE CALLS TO LIBRARY ROUTINES.

- ③ EXPLICIT CONVERSIONS ARE NEITHER MORE NOR LESS EFFICIENT THAN IMPLICIT ONES.
- ④ CONVERSIONS ARE PERFORMED ON THE RIGHT SIDE OF AN = SIGN WITHOUT REGARD FOR WHAT IS ON THE LEFT SIDE. ONLY UPON ASSIGNMENT, IS THE LEFT SIDE TAKEN INTO ACCOUNT.

31
237

INTEGER/SCALAR CONVERSIONS (CON'T.)

INTERPRETATION (CON'T.)

⑤ WHEN ONLY INTEGERS APPEAR IN AN EXPRESSION, ALL INTEGERS ARE CONVERTED TO THE PRECISION OF THE MOST PRECISE INTEGER.

EXCEPTION: IN "I/J" ALL INTEGERS ARE CONVERTED TO SCALARS.

⑥ WHEN ONLY SCALARS APPEAR IN AN EXPRESSION, ALL SCALARS ARE CONVERTED TO THE PRECISION OF THE MOST PRECISE SCALAR.

⑦ IF INTEGERS AND SCALARS ARE MIXED, ALL INTEGERS ARE CONVERTED TO SCALARS OF THE REQUISITE PRECISION.

2-37

51
238

EXPRESSIONS

EXPRESSION \equiv A MEANINGFUL COMBINATION OF OPERATORS AND OPERANDS
THAT RESULTS IN SOMETHING BELONGING TO A LEGAL HAL/S
DATA TYPE -- THIS DEFINES THE TYPE OF THE EXPRESSION.

ARITHMETIC OPERATIONS

**	}	EXPONENTIATION	I^{**3}
		INVERSION	$M^{**(-1)}$
		TRANSPOSITION	M^{**T}
(BLANK)		MULTIPLICATION	$2\ I$
		(VECTOR OUTER PRODUCT)	$\bar{V}\ \bar{W}$
		(MATRIX MULTIPLICATION)	$\bar{M}\ \bar{N}$
*		VECTOR CROSS PRODUCT	$\bar{V} * \bar{W}$
•		VECTOR DOT PRODUCT	$\bar{V} \bullet \bar{W}$
/		DIVISION	
+		ADDITION	
-		SUBTRACTION	$A - B$
		(NEGATION -- UNARY OP)	$- B$

3-1

EXPRESSIONS (CON'T.)

52
239

NEGATION

- MATRIX (ALL ELEMENTS)
- VECTOR (ALL ELEMENTS)
- SCALAR
- INTEGER

ADDITION AND SUBTRACTION

- MATRIX \pm MATRIX (MUST BE CONFORMABLE)
- VECTOR \pm VECTOR (MUST BE CONFORMABLE)
- SCALAR \pm SCALAR
- SCALAR \pm INTEGER*
- INTEGER \pm INTEGER

* RESULT IS SCALAR. THE INTEGER IS CONVERTED TO A SCALAR OF REQUISITE PRECISION.

NOTE: LEFT AND RIGHT-HAND SIDES CAN BE SUBSCRIPTED VARIABLES OR MORE COMPLEX EXPRESSIONS PROVIDED THEY ARE OF THE CORRECT TYPE.

EXPRESSIONS (CON'T.)

33
240

DECLARE M ARRAY(3) MATRIX INITIAL(1,2,3,4,5,6,7,8,9,10,11,12,13,14,
15,16,17,18,19,20,21,22,23,24,25,26,27);

$$M \equiv \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}, \begin{bmatrix} 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{bmatrix}, \begin{bmatrix} 19 & 20 & 21 \\ 22 & 23 & 24 \\ 25 & 26 & 27 \end{bmatrix}$$

DECLARE V VECTOR INITIAL(1.5, 2.5, 3.5);

DECLARE S INITIAL(50.3);

DECLARE I INTEGER INITIAL(5);

$I + 1$	\equiv	6	(INTEGER RESULT)
$I + 0.3$	\equiv	5.3	(SCALAR RESULT)
$S + 4$	\equiv	54.3	(SCALAR RESULT)
$I + S$	\equiv	55.3	(SCALAR RESULT)
$S + \bar{V}$	ILLEGAL		(TYPE MISMATCH)
$S + V\$2$	\equiv	52.8	(SCALAR RESULT)
$\bar{V} + \bar{M}\$(2:)$	ILLEGAL		(TYPE MISMATCH)

$$V + \bar{M}\$(2:*, 2) \equiv \begin{bmatrix} 1.5 \\ 2.5 \\ 3.5 \end{bmatrix} + \begin{bmatrix} 11 \\ 14 \\ 17 \end{bmatrix} \equiv \begin{bmatrix} 12.5 \\ 16.5 \\ 20.5 \end{bmatrix}$$

(VECTOR RESULT)

S2
241

EXPRESSIONS (CON'T.)

DIVISION

MATRIX/SCALAR	<u>OR</u>	MATRIX/INTEGER*
VECTOR/SCALAR	<u>OR</u>	VECTOR/INTEGER*
SCALAR/SCALAR	<u>OR</u>	SCALAR/INTEGER*
INTEGER*/SCALAR	<u>OR</u>	INTEGER*/INTEGER*

* INTEGER WILL BE CONVERTED TO A SCALAR OF REQUISITE PRECISION.

NOTE: IF USERS WANT A TRUE INTEGER DIVISION, E.G., IN THE FORTRAN SENSE, THE DIV FUNCTION SHOULD BE USED:

$$I = \text{DIV}(J,4);$$

IF J = 10, I WILL BE 2.

EXAMPLES

1/3 IS 0.333333 (A SCALAR -- NOT AN INTEGER 0)

$$\text{IF } V = \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} \text{ THEN } V/2 = \begin{bmatrix} 5 \\ 10 \\ 15 \end{bmatrix}$$

SR
242

EXPRESSIONS (CON'T.)

DOT PRODUCT

$$\vec{V} \cdot \vec{W}$$

STANDARD MATHEMATICAL DEFINITION WITH THE FOLLOWING EQUIVALENCE:

$$v^T w = \vec{V} \cdot \vec{W}$$

CLEARLY V AND W MUST HAVE IDENTICAL LENGTHS. THE RESULT IS A SCALAR AND OF SAME PRECISION AS MOST PRECISE VECTOR.

CROSS PRODUCT

$$\vec{V} * \vec{W}$$

STANDARD MATHEMATICAL DEFINITION. BOTH V AND W MUST BE 3-VECTORS AND THE RESULT IS A 3-VECTOR.

NOTE: HAL/S DOES NOT ALLOW THE FORM \vec{V}^T (I.E., THE TRANSPOSE OF A VECTOR). AS WILL BE SEEN LATER, CONTEXT IS USED TO DETERMINE MEANING.

3-5

EXPRESSIONS (CON'T.)

52
243

MULTIPLICATION

A * B

MULTIPLICATION IS INDICATED BY LOGICAL ADJACENCY, I.E., A BLANK. THE BLANK IS NOT NEEDED PRIOR TO A PARENTHESIZED EXPRESSION, BUT IT IS MANDATORY IN ALL BUT A FEW SPECIAL CASES.

EXAMPLES:

3(I + J)
└ NO BLANK NEEDED

COMPARE 3I
└ BLANK NOT NEEDED BUT IS DESIRABLE

{
3E
3B
3H
}

WOULD LOOK LIKE ILLEGAL ARITHMETIC LITERALS

M N
└ BLANK NEEDED

3-6

EXPRESSIONS (CON'T.)

SZ
244

MULTIPLICATION ASSUMES 5 FORMS:

- ① INTEGER* SCALAR
- SCALAR INTEGER*
- INTEGER INTEGER
- SCALAR SCALAR

* INTEGER WILL BE CONVERTED TO A SCALAR OF THE REQUISITE PRECISION.

- ② INTEGER* VECTOR
- SCALAR VECTOR
- VECTOR INTEGER*
- VECTOR SCALAR
- INTEGER* MATRIX
- SCALAR MATRIX
- MATRIX INTEGER*
- MATRIX SCALAR



ELEMENT-BY-ELEMENT
MULTIPLICATION

* INTEGER WILL BE CONVERTED TO A SCALAR OF THE REQUISITE PRECISION.

EXPRESSIONS (CON'T.)

53
245

- ③ VECTOR VECTOR
DEFINES THE DYADIC PRODUCT (VECTOR OUTER PRODUCT)

$$\bar{V} \bar{W}$$

IS EQUIVALENT TO THE MATHEMATICAL FORM $V W^T$

EXAMPLE:

$$V = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

$$W = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix}$$

$$\bar{V} \bar{W} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 6 & 10 \\ 3 & 9 & 15 \\ 4 & 12 & 20 \end{bmatrix}$$

- ④ MATRIX MATRIX
DEFINES STANDARD MATRIX MULTIPLICATION, I.E.,

$$\bar{M} \bar{N}$$

WHERE M IS A MxN MATRIX AND N IS A NxP MATRIX. AGAIN, NOTE THAT M AND N COULD BE SUBSCRIPTED EXPRESSIONS, E.G., PARTITIONS OF LARGER MATRICES.

3-4

53
246

EXPRESSIONS (CON'T.)

⑤

VECTOR MATRIX
MATRIX VECTOR

DEFINES STANDARD VECTOR-MATRIX MULTIPLICATION.

A. IF $\bar{V} \overset{*}{M}$ THEN IF V HAS LENGTH L, M MUST HAVE DIMENSIONS $L \times P$.
THE RESULT IS A VECTOR OF LENGTH P, EQUIVALENT MATHEMATICALLY
TO

$$V^T M \equiv [\text{ROW VECTOR}] [\text{MATRIX}]$$

B. IF $\overset{*}{M} \bar{V}$ THEN IF V HAS LENGTH L, M MUST HAVE DIMENSIONS $M \times L$.
THE RESULT IS A VECTOR OF LENGTH M, EQUIVALENT MATHEMATICALLY
TO

$$M V \equiv [\text{MATRIX}] [\text{COLUMN VECTOR}]$$

EXPRESSIONS (CON'T.)

S2
247

EXPONENTIATION (INTEGERS AND SCALARS)

- ① INTEGER**INTEGER
- ② INTEGER**SCALAR
- ③ SCALAR**INTEGER
- ④ SCALAR**SCALAR

IN CASE ① IF THE EXPONENT IS A NON-NEGATIVE INTEGER, THE RESULT OF THE EXPRESSION IS AN INTEGER. IN CASE ② - ④ THE RESULT IS ALWAYS A SCALAR.

EXAMPLES

IF I IS AN INTEGER WITH I = 5 THEN

$$I**2 \quad \equiv \quad 25 \quad \text{(INTEGER RESULT)}$$

$$I**(-1) \quad \equiv \quad 0.2 \quad \text{(SCALAR RESULT)}$$

$$2**,.5 \quad \equiv \quad \sqrt{2} \quad \text{(SCALAR RESULT)}$$

NOTE: AN EXPONENT OF ,5 (OR 1/2) IS RECOGNIZED AS A SPECIAL CASE AND IS ENTIRELY EQUIVALENT TO USING THE SQRT BUILT-IN FUNCTION.

$$S**,.5 \quad \equiv \quad \text{SQRT}(S)$$

EXPRESSIONS (CON'T.)

52
248

EXPONENTIATION (SQUARE MATRIX TO AN INTEGRAL POWER)

M^{**I}

M IS A SQUARE MATRIX (N×N)

I IS AN INTEGER LITERAL

<u>I</u>	<u>RESULT</u>
≤ -2	REPEATED PRODUCT OF INVERSE
-1	MATRIX INVERSE
0	UNIT MATRIX
1	NO-OP
≥ 2	REPEATED PRODUCT

NOTE THAT THE VALUE OF I MUST BE KNOWN AT COMPILE-TIME.

$M^{**(-1)}$ IS EQUIVALENT TO USING THE MATRIX INVERSION FUNCTION:

INVERSE(M)

M^{**0} IS A FAST WAY OF BUILDING AN N×N UNIT MATRIX DYNAMICALLY.

53
249

EXPRESSIONS (CON'T.)

$$M^{**(-2)} \equiv (M^{**(-1)})^{**2}$$

$$M^{**3} \equiv \overset{*}{M} \overset{*}{M} \overset{*}{M}$$

$$M^{**0} \equiv \begin{bmatrix} 1 & & & \\ & 1 & & 0 \\ & & \ddots & \\ & & & 1 \\ 0 & & & & 1 \end{bmatrix}$$

NOT
USED

EXAMPLE

IF A MATRIX M HAS CHARACTERISTIC EQUATION

$$A\lambda^2 + B\lambda + C = 0$$

WE COULD CODE THE FOLLOWING IN HAL/S:

$$A M^{**2} + B M + C M^{**0}$$

THE RESULT WILL BE AN $N \times N$ ZERO MATRIX.

EXPRESSIONS (CON'T.)

32
250

EXPONENTIATION (TRANSPOSE OF A MATRIX)

M**T

THIS IS EXACTLY EQUIVALENT TO USING THE BUILT-IN FUNCTION
TRANSPOSE

M**T ≡ TRANSPOSE(M)

IF M IS AN MxN MATRIX, M^T WILL BE AN NxM MATRIX.

NOTE: VECTORS CANNOT BE TRANSPosed. V**T IS ILLEGAL.

NOTE: EVEN IF THE USER HAS DEFINED T TO BE A VARIABLE, M**T
MEANS TRANSPOSE -- IT IS NOT THE SAME "T"!

COMMENTARY ON VECTOR/MATRIX OPS

52
251

RULE: IF A VECTOR TRANSPOSE (I.E., ROW VECTOR) APPEARS IN A MATHEMATICAL STATEMENT, LEAVE THE TRANSPOSE OPERATION OFF WHEN CODING THE HAL/S STATEMENTS.

	<u>MATH</u>	<u>HAL/S</u>
<u>DOT PRODUCT</u>	$V^T W$	$V.W$
<u>CROSS PRODUCT</u>	$V \times W$	$V*W$
<u>OUTER PRODUCT</u>	$V W^T$	$V W$
<u>MATRIX/ VECTOR PRODUCT</u>	$M V$	$M V$
<u>VECTOR/ MATRIX PRODUCT</u>	$V^T M$	$V M$

NOTE: THE MATRIX TRANSPOSE OPERATION IN HAL/S IS EXPENSIVE, AVOID IT IF YOU CAN.

COMMENTARY ON VECTOR/MATRIX OPS (CON'T.)

83
202

EXAMPLES

① MATH: $\bar{X} = M^T \bar{Y}$

HAL/S: $X = M^{**T} Y$

BUT THIS CAN BE IMPROVED UPON! IN THE MATH CASE WE COULD WRITE

$$\bar{X}^T = (M^T \bar{Y})^T = \bar{Y}^T M$$

IN HAL/S THIS WOULD BECOME

$$X = Y M \quad (\text{ELIMINATION OF A MATRIX TRANSPOSE})$$

SINCE VECTOR TRANSPOSE IS ILLEGAL (AND NOT NEEDED!).

ACTUALLY, THE LATEST RELEASE OF THE COMPILER (WITH THE GLOBAL OPTIMIZATION OF PHASE 1.5) MAKES THIS SUBSTITUTION AUTOMATICALLY!

② MATH: $\bar{X} = M^T N^T P \bar{Y} + Q^T \bar{R}$

HAL/S: $X = (M^{**T})(N^{**T}) P Y + (Q^{**T}) R$

BUT BETTER IS:

HAL/S: $X = Y(P^{**T}) N M + R Q$

MATRIX/VECTOR OPTIMIZATION

52
253

```
1 M↑ PROGRAM;  
2 M↑   DECLARE MATRIX,  
2 M↑       M INITIAL(1),  
2 M↑       N INITIAL(2),  
2 M↑       P INITIAL(3),  
2 M↑       Q INITIAL(4);  
3 M↑   DECLARE VECTOR,  
3 M↑       X,  
3 M↑       Y INITIAL(1, 2, 3),  
3 M↑       R INITIAL(4, 5, 6);  
C↑  
E↑   - *T*T* - *T-  
4 M↑   X = M N P Y + Q R;
```

LA	3,72(0,10)	P
LA	4,156(0,10)	Y
LA	2,76(0,13)	
BAL	14,184(0,12)	VM653
LA	4,36(0,10)	N
LA	2,88(0,13)	
BAL	14,176(0,12)	VM653
LA	4,0(0,10)	M
LA	2,76(0,13)	
BAL	14,176(0,12)	VM653
LA	3,168(0,10)	R
LA	4,108(0,10)	Q
LA	2,88(0,13)	
BAL	14,176(0,12)	VM653
LA	3,76(0,13)	
LA	4,88(0,13)	
LA	2,100(0,13)	
BAL	14,112(0,12)	VV253
LA	3,100(0,13)	
LA	2,144(0,10)	X
BAL	14,48(0,12)	VV153

53
254

MATRIX/VECTOR OPTIMIZATION (CON'T.)

$$\begin{matrix} E \uparrow & - & - & *T* & * & - & * \\ 6 \text{ M} \uparrow & X = & Y & P & N & N & + & R & Q; \end{matrix}$$

LA 3,100(0,13)
LA 2,144(0,10)
BAL 14,48(0,12)

X
VV153

#4 $CSE = (\bar{P} \bar{Y}) \bar{N} \bar{M} + \bar{R} \bar{Q}$

$\bar{X} = CSE$

#6 $\bar{X} = CSE$

3-17

EXPRESSIONS (CON'T.)

51
255

CHARACTER OPERATIONS

CATENATION || OR CAT

CHAR||INTEGER*
INTEGER*||CHAR
INTEGER*||INTEGER*
CHAR||CHAR
CHAR||SCALAR*
SCALAR*||CHAR
SCALAR*||SCALAR*
INTEGER*||SCALAR*
SCALAR*||INTEGER*

* INTEGERS AND SCALARS ARE IMPLICITLY CONVERTED TO CHARACTER STRINGS:

EXAMPLE:

WRITE(6) 'THRUST OF ENGINE '||5|| 'IS' ||40095|| 'POUNDS';

← WILL BE CONVERTED TO CHARACTER

← WILL BE CONVERTED TO CHARACTER

8/
252

EXPRESSIONS (CON'T.)

BOOLEAN OPERATIONS

- | | | |
|---|---------|------------------------|
| 1 | & ≡ AND | (LOGICAL INTERSECTION) |
| 2 | ≡ OR | (LOGICAL CONJUNCTION) |
| 3 | ¬ ≡ NOT | (LOGICAL COMPLEMENT) |

ASSUME:

DECLARE B1 BOOLEAN INITIAL(TRUE);
DECLARE B2 BOOLEAN INITIAL(TRUE);
DECLARE B3 BOOLEAN INITIAL(FALSE);
DECLARE B4 BOOLEAN INITIAL(FALSE);

COMPLEMENT (¬)

UNARY OPERATION

¬ B1 ≡ FALSE

CONJUNCTION (|)

BINARY OPERATION

B1|B2 ≡ TRUE

T|T ≡ T

B1|B3 ≡ TRUE

T|F ≡ T

B3|B1 ≡ TRUE

F|T ≡ T

B3|B4 ≡ FALSE

F|F ≡ F

EXPRESSIONS (CON'T.)

52
207

INTERSECTION (&)

BINARY OPERATION

B1 & B2 ≡ TRUE

T&T ≡ T

B1 & B3 ≡ FALSE

T&F ≡ F

B3 & B1 ≡ FALSE

F&T ≡ F

B3 & B4 ≡ FALSE

F&F ≡ F

OTHER EXAMPLES

B1 & FALSE ≡ FALSE

B3 | TRUE ≡ TRUE

¬B1 | B2&B3 | ¬B4 ≡ TRUE

FALSE FALSE TRUE

3-20

EXPRESSIONS (CON'T.)

31
258

ARITHMETIC AND CHARACTER PRECEDENCE

↑	HI	**	1	EXPONENTIATION
		∅	2	MULTIPLICATION
		*	3	CROSS PRODUCT
		•	4	DOT PRODUCT
		/	5	DIVISION
		+	6	ADDITION
		-	6	SUBTRACTION, NEGATION
			7	CHARACTER CATENATION
↓	LO			

- SEQUENCES OF OPERATIONS OF THE SAME PRECEDENCE ARE EVALUATED FROM LEFT TO RIGHT -- EXCEPT FOR ** AND /, WHICH ARE EVALUATED FROM RIGHT TO LEFT, I.E.,

$$A/B/C/D \equiv (AC)/(BD) \quad A/B(C/D) = A/(BD/C) = AC/BD$$

- SEQUENCES OF MULTIPLICATIONS ARE SOMETIMES REORDERED TO MINIMIZE THE NUMBER OF ELEMENTAL PRODUCTS INVOLVED.

DIVIDE EXAMPLE

```

1 MT PROGRAM;
2 MT   DECLARE INITIAL(1),
2 MT     S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19,
2 MT     S20;
3 MT   DECLARE T;
  CT
4 MT   T = S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11 S12 S13 S14 S15 S16 S17 S18 S19 S20;

```

	ST#4	EQU	*	
000002A	\$0TEST1	CSECT		ESDID= 0001
00009 7905	0002	LE	0,2(1)	S1
0000A 6000	0006	ME	0,6(1)	S3
0000B 6015	000A	ME	0,10(1)	S5
0000C 6010	000E	ME	0,14(1)	S7
0000D 6025	0012	ME	0,18(1)	S9
0000E 6020	0016	ME	0,22(1)	S11
0000F 6035	001A	ME	0,26(1)	S13
00010 6030	001E	ME	0,30(1)	S15
00011 6045	0022	ME	0,34(1)	S17
00012 6040	0026	ME	0,38(1)	S19
00013 7A19	000C	LE	2,12(1)	S6
00014 6231	0018	ME	2,24(1)	S12
00015 6211	0008	ME	2,8(1)	S4
00016 6239	001C	ME	2,28(1)	S14
00017 6221	0010	ME	2,16(1)	S8
00018 6241	0020	ME	2,32(1)	S16
00019 6209	0004	ME	2,4(1)	S2
0001A 6249	0024	ME	2,36(1)	S18
0001B 6229	0014	ME	2,20(1)	S10
0001C 6251	0028	ME	2,40(1)	S20
0001D 7CE0		LER	4,0	
0001E 64E2		MER	4,2	
0001F 3C55	002A	STE	4,42(1)	T

DIVIDE EXAMPLE (CON'T.)

C↑
 5 N↑ T = S1 / S2 / S3 / S4 / S5 / S6 / S7 / S8 / S9 / S10 / S11 / S12 / S13 / S14 / S15 / S16 / S17 /
 5 N↑ S18 / S19 / S20
 6 N↑ CLOSE TEST1

0000020		ST#5	EQU	*	
00000 68E2			DER	0,2	
00001 3955	002A		STE	0,42(1)	T
0000022		ST#6	EQU	*	
0000022		LBL#2	EQU	*	
00022 09F9 002C	002C		SVC	44(1)	H'21'

3-23

MATRIX INVERTER (4x4)

```
INVERTER:PROGRAM;
REPLACE PREC BY "S INGLE";
DECLARE MATRIX(4,4) PREC,Q,Q1,Q2;
DECLARE I MATRIX(4,4) PREC CONSTANT(1,3#(0,0,0,0,1));
DECLARE MATRIX(2,2) PREC,C,S,T,M,N;
INVERT:FUNCTION(IN_MAT) MATRIX(2,2) PREC;
DECLARE MATRIX(2,2) PREC,IN_MAT;
RETURN MATRIX$(@PREC,2,2)(IN_MAT$(2,2),-IN_MAT$(1,2),
-IN_MAT$(2,1),IN_MAT$(1,1))/(IN_MAT$(1,1) IN_MAT$(2,2)
-IN_MAT$(1,2) IN_MAT$(2,1));
CLOSE INVERT;
ON ERROR$(10:5) RETURN;
DO WHILE TRUE;
READ(5) Q;
Q1=Q**(-1);
S=INVERT(Q$(1 TO 2,1 TO 2));
T=S Q$(1 TO 2,3 TO 4);
C=Q$(3 TO 4,1 TO 2);
N=INVERT(Q$(3 TO 4,3 TO 4)-C T);
M=-N C S;
Q2$(1 TO 2,1 TO 2)=S-T M;
Q2$(1 TO 2,3 TO 4)=- T N;
Q2$(3 TO 4,1 TO 2)=M;
Q2$(3 TO 4,3 TO 4)=N;
```

MATRIX INVERTER (4x4) (CON'T.)

```
WRITE(6)'Q = ',C,SKIP(2);  
WRITE(6)'DET(Q) = ',DET(Q),SKIP(2);  
WRITE(6)'Q1 = ',C1,SKIP(2);  
WRITE(6)'DET(Q1) = ',DET(Q1),SKIP(2);  
WRITE(6)'Q2 = ',C2,SKIP(2);  
WRITE(6)'DET(Q2) = ',DET(Q2),SKIP(2);  
WRITE(6)'I - Q Q1 = ',I-Q Q1,SKIP(2);  
WRITE(6)'I - Q Q2 = ',I-Q Q2,SKIP(2);  
END;  
CLOSE;
```

MATRIX INVERTER (4x4)

HAL/S COMPILATION	I N T E R M E T R I C S , I N C .	O C T O B E R 1 8 , 1 9 7 5
STMT	S O U R C E	
1 M	INVERTER;	
1 M	PROGRAM;	
2 M	REPLACE PREC BY "SINGL";	
3 M	DECLARE MATRIX(4, 4) PBEC, Q, C1, Q2;	
4 M	DECLARE I MATRIX(4, 4) PBEC CONSTANT(1, 3#(0, 0, 0, 1));	
5 M	DECLARE MATRIX(2, 2) PBEC, C, S, T, M, N;	
6 M	INVERT:	
6 M	FUNCTION(IN_MAT) MATRIX(2, 2) PBEC;	
7 M	DECLARE MATRIX(2, 2) PBEC, IN_MAT;	
8 M	RETURN MATRIX	
SI	$\frac{\begin{pmatrix} \text{IN_MAT}_{1,1} & -\text{IN_MAT}_{1,2} \\ -\text{IN_MAT}_{2,1} & \text{IN_MAT}_{2,2} \end{pmatrix}}{\begin{vmatrix} \text{IN_MAT}_{1,1} & \text{IN_MAT}_{1,2} \\ \text{IN_MAT}_{2,1} & \text{IN_MAT}_{2,2} \end{vmatrix}}$	
8 M	IN_MAT	
SI	$\begin{pmatrix} \text{IN_MAT}_{1,2} & \text{IN_MAT}_{2,1} \end{pmatrix}$	
9 M	CLOSE INVERT;	

MATRIX INVERTER (4x4)

```

10 M|      ON ERROR
   S|      10:5
11 M|      RETURN;
12 M|      DO WHILE TRUE;
13 E|      *
   M|      READ(5) Q;
14 E|      *
   M|      Q1 = Q-1;
15 E|      *
   M|      S = INVERT(Q*
   S|      1 TO 2, 1 TO 2);
16 E|      *
   M|      T = S Q*
   S|      1 TO 2, 3 TO 4;
17 E|      *
   M|      C = Q*
   S|      3 TO 4, 1 TO 2;
18 E|      *
   M|      N = INVERT(C*
   S|      3 TO 4, 3 TO 4 - C T);
19 E|      *
   M|      H = -N C S;
20 E|      *
   M|      Q2 = S* - T* H;
   S|      1 TO 2, 1 TO 2;
21 E|      *
   M|      Q2 = -T* N;
   S|      1 TO 2, 3 TO 4;
22 E|      *
   M|      Q2 = N;
   S|      3 TO 4, 1 TO 2;
23 E|      *
   M|      Q2 = N;
   S|      3 TO 4, 3 TO 4;
24 E|      *
   M|      WRITE(6) 'C = ', Q, SKIP(2);
25 F|
   M|      WRITE(6) 'DET(Q) = ', DET(Q), SKIP(2);

```

MATRIX INVERTER (4x4)

```
25 E| WRITE(6) 'Q1 = ', Q1, SKIP(2);  
27 E| WRITE(6) 'DET(Q1) = ', DET(Q1), SKIP(2);  
28 E| WRITE(6) 'Q2 = ', Q2, SKIP(2);  
29 E| WRITE(6) 'DET(Q2) = ', DET(Q2), SKIP(2);  
30 E| WRITE(6) 'I - Q Q1 = ', I - Q Q1, SKIP(2);  
31 E| WRITE(6) 'I - Q Q2 = ', I - Q Q2, SKIP(2);  
32 M| ENDI  
33 M| CLOSE;
```

MATRIX INVERTER (4x4)

1ST MATRIX

HAL/S-360 V13.0 START TIME: 11:27:03.55 DAY: 75/291

0 =	1.0000000E+00	4.1555555E-01	3.3999996E-01	6.5999996E-01
	4.1999995E-01	1.0000000E+00	3.1999999E-01	4.3999999E-01
	5.3599996E-01	3.2000000E+00	1.0000000E+00	2.1999995E-01
	6.5999996E-01	4.3999999E-01	2.1999996E-01	1.0000000E+00

DET(Q) = -2.0837659E+00

Q1 =	3.0490074E+00	-5.9853372E+00	1.3890212E-01	5.9064388E-01
	1.6895326E-02	-1.8294560E-01	3.5900391E-02	6.1446364E-02
	-1.8314371E+00	8.6166076E+00	-2.1035748E-01	-2.5362777E+00
	-1.6168603E+00	2.1351613E+00	-6.1152073E-02	1.1411180E+00

DET(Q1) = -4.7989875E-01

Q2 =	3.0490131E+00	-5.9853506E+00	1.3890260E-01	5.9064686E-01
	1.6896486E-02	-1.8294620E-01	3.5900425E-02	6.1446640E-02
	-1.8314485E+00	8.6166362E+00	-2.1035836E-01	-2.5362844E+00
	-1.6168642E+00	2.1351680E+00	-6.1193086E-02	1.1411190E+00

DET(Q2) = -4.7990131E-01

I - Q Q1 =	0.0	1.4177550E-07	3.5082852E-08	6.7152376E-07
	-8.7906359E-07	1.6093254E-06	-2.0574393E-08	1.5908807E-07
	2.3281415E-06	-1.7261700E-06	2.9802322E-07	4.8220463E-07
	-2.6787356E-06	3.6383580E-06	-5.7375285E-08	1.6689300E-06

I - Q Q2 =	2.8610229E-06	-6.1119417E-06	1.7110704E-07	5.5097114E-07
	1.8972659E-06	-3.8146972E-06	1.2515891E-07	3.4833280E-07
	4.0907016E-06	-5.5217942E-06	0.0	-3.4025161E-06
	-1.9337113E-07	-2.5740047E-07	2.2192899E-08	1.1920928E-07

MATRIX INVERTER (4x4)

2ND MATRIX

Q =	5.0000000E+00	7.0000000E+00	6.0000000E+00	5.0000000E+00
	7.0000000E+00	1.0000000E+01	8.0000000E+00	7.0000000E+00
	6.0000000E+00	8.0000000E+00	1.0000000E+01	9.0000000E+00
	5.0000000E+00	7.0000000E+00	9.0000000E+00	1.0000000E+01

DET(Q) = 1.0000171E+00

Q1 =	6.7991653E+01	-4.0994903E+01	-1.6997940E+01	9.9987649E+00
	-4.0994903E+01	2.4996887E+01	9.9987497E+00	-5.9992455E+00
	-1.6997940E+01	9.9987449E+00	4.9994935E+00	-2.9996938E+00
	9.9987649E+00	-5.9992465E+00	-2.9996957E+00	1.9998158E+00

DET(Q1) = 1.0000457E+00

Q2 =	6.8000000E+01	-4.1000000E+01	-1.7000000E+01	1.0000000E+01
	-4.1000000E+01	2.5000000E+01	1.0000000E+01	-6.0000000E+00
	-1.7000000E+01	1.0000000E+01	5.0000000E+00	-3.0000000E+00
	1.0000000E+01	-6.0000000E+00	-3.0000000E+00	2.0000000E+00

DET(Q2) = 1.0000562E+00

I - Q Q1 =	-1.2683868E-04	7.0571899E-05	-3.0517578E-05	-2.6702880E-05
	-3.7288665E-04	2.1934509E-04	4.7683715E-06	-6.5803527E-05
	-1.7642974E-04	9.3460083E-05	-3.1471252E-05	-3.8145972E-05
	-1.3160705E-04	6.8664553E-05	-3.2424926E-05	-2.9563903E-05

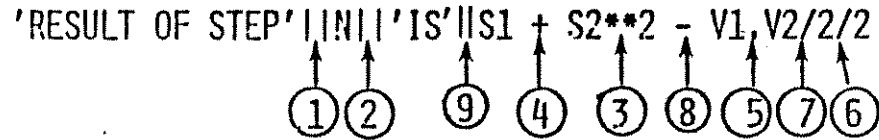
I - Q Q2 =	0.0	0.0	0.0	0.0
	0.0	0.0	0.0	0.0
	0.0	0.0	0.0	0.0
	0.0	0.0	0.0	0.0

EXPRESSIONS (CON'T.)

52
257

PRECEDENCE EXAMPLE:

(FROM PROGRAMMER'S GUIDE)



ALSO,

$$\bar{V} \cdot \bar{W} \cdot \bar{X} \equiv \bar{V} \cdot (\bar{W} \cdot \bar{X}) \quad (\text{A SCALAR})$$

$$\bar{V} \cdot \bar{W} \cdot \bar{X} \equiv (\bar{V} \cdot \bar{W}) \cdot \bar{X}$$

$$2^{**}3^{**}2 \equiv 2^{**}(3^{**}2) \equiv 2^{**}9 \equiv 512$$

NOT! $(2^{**}3)^{**}2 \equiv (8)^{**}2 \equiv 64$

$$4/3/10/3 \equiv 4/3/(10/3) \equiv 4/(9/10) \equiv 40/9$$

NOT! $(4/3)/10/3 \equiv (4/30)/3 \equiv 4/90$

5/
260

EXPRESSIONS (CON'T.)

BOOLEAN PRECEDENCE

HI	↑ ↓	\neg , NOT	1	COMPLEMENT
		&, AND	2	INTERSECTION
LO		, OR	3	CONJUNCTION

- SEQUENCES OF OPERANDS OF THE SAME PRECEDENCE ARE EVALUATED FROM LEFT TO RIGHT.

EXAMPLES

- ① IF $B1|\neg B2|B3\&\neg B4$ THEN DO;
EQUIVALENT TO:
IF $(B1)|(\neg B2)|(B3\&\neg B4)$ THEN DO;
- ② IF $B1\&B2|B3\&B4|\neg B5\&\neg B6\dots$
EQUIVALENT TO:
IF $(B1\&B2)|(B3\&B4)|((\neg B5)\&(\neg B6))\dots$

EXPRESSIONS (CON'T.)

52
261

OVERRIDING PRECEDENCE ORDER

- PRECEDENCE ORDER CAN ALWAYS BE ALTERED BY USE OF PARENTHESES.

EXAMPLES

- ① IF (B1|(B2|B3))&¬B4 THEN DO;
- ② IF B1&(B2|B3)&(B4|¬B5)&¬B6 ...
- ③ (A**B)**C
- ④ (A/B)/(C/D)

EXPLICIT CONVERSIONS

51
261A

A. VECTOR CONVERSION

VECTORS CAN BE DYNAMICALLY CONSTRUCTED VIA THE CONSTRUCTION

$\text{VECTOR}(\ell) (\text{exp}_1, \text{exp}_2, \dots \text{exp}_\ell)$

TO CREATE AN ℓ -VECTOR. IF THE DEFAULT IS DESIRED THE LENGTH SPECIFICATION CAN BE OMITTED, I.E.,

$\text{VECTOR}(\text{exp}_1, \text{exp}_2, \text{exp}_3)$

IF exp IS AN ARRAY OF ℓ INTEGERS OR SCALARS, WE CAN SAY:

$\text{VECTOR}(\ell)(\text{exp})$

- ALL EXPRESSIONS (exp_i) MUST BE OF INTEGER OR SCALAR TYPE. (AGGREGATES OF THESE TYPES MAY ALSO BE USED.)
- THE RESULT OF THE VECTOR CONVERSION FUNCTION IS A SINGLE-PRECISION VECTOR. IF A DOUBLE-PRECISION VECTOR IS NEEDED USE THE FORM:

$\text{VECTOR}(\text{@DOUBLE}, \ell) (\text{exp}_1, \text{exp}_2, \dots)$

CLEARLY $2 \leq \ell \leq 64$

EXPLICIT CONVERSIONS (CON'T.)

52
262

VECTOR EXAMPLES

① VECTOR(1,0,0) \equiv $\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ SINGLE PRECISION (3-VECTOR)

VECTOR\$(@DOUBLE)(1,0,0) \equiv $\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ DOUBLE PRECISION (3-VECTOR)

② VECTOR\$4(3, 2**2, 8-6, SIN(0)) \equiv $\begin{bmatrix} 3 \\ 4 \\ 2 \\ 0 \end{bmatrix}$ SINGLE PRECISION (4-VECTOR)

VECTOR\$(@DOUBLE,4)(3, 2**2, 8-6, SIN(0)) \equiv $\begin{bmatrix} 3 \\ 4 \\ 2 \\ 0 \end{bmatrix}$ DOUBLE PRECISION (4-VECTOR)

③ DECLARE Q ARRAY(12) INITIAL(1,2,3,4,5,6,7,8,9,10,11,12);

VECTOR\$(@DOUBLE, 5)(Q\$(5 AT 6)) \equiv $\begin{bmatrix} 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{bmatrix}$

EXPLICIT CONVERSIONS (CON'T.)

52
263

B. MATRIX CONVERSIONS

MATRICES CAN ALSO BE CONSTRUCTED DYNAMICALLY. USE THE CONSTRUCTION:

`MATRIX$(r,c) (exp1, exp2, ...exprc)`

TO CREATE AN $r \times c$ SINGLE PRECISION MATRIX.

$$2 \leq r \leq 64$$

$$2 \leq c \leq 64$$

r AND c DEFAULT TO A 3x3 MATRIX, E.G.,

`MATRIX(exp1, exp2, ...exp9)`

WILL CREATE A 3x3 MATRIX (SINGLE PRECISION).

IF `exp` IS AN ARRAY OF rc INTEGERS OR SCALARS WE CAN SAY

`MATRIX$(r,c)(exp)`

- ALL EXPRESSIONS (`exp1`) MUST BE OF INTEGER OR SCALAR TYPE.
- IF A DOUBLE-PRECISION MATRIX IS NEEDED, USE THE FORM:

`MATRIX$(@DOUBLE,r,c)(exp1, ... exprc)`

EXPLICIT CONVERSIONS (CON'T.)

53
264

- MATRICES ARE ASSEMBLED ROW-BY-ROW FROM THE LIST (JUST AS IS DONE IN INITIAL LISTS IN DECLARE STATEMENTS)

MATRIX EXAMPLES

① $\text{MATRIX}(\overbrace{1,0,0,0}^{\text{ROW 1}}, \overbrace{0,1,0,0,0}^{\text{ROW 2}}, \overbrace{0,0,0,1}^{\text{ROW 3}}) \equiv \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ A 3x3 SINGLE-PRECISION MATRIX

$\text{MATRIX}(\text{@DOUBLE})(1,0,0,0,1,0,0,0,1)$

$\equiv \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ A 3x3 DOUBLE-PRECISION MATRIX

② $\text{MATRIX}(2,2)(4,5,7,9) \equiv \begin{bmatrix} 4 & 5 \\ 7 & 9 \end{bmatrix}$ A 2x2 SINGLE-PRECISION MATRIX

$\text{MATRIX}(\text{@DOUBLE}, 2, 2)(4, 5, 7, 9) \equiv \begin{bmatrix} 4 & 5 \\ 7 & 9 \end{bmatrix}$ A 2x2 DOUBLE-PRECISION MATRIX

EXPLICIT CONVERSIONS (CON'T.)

MATRIX EXAMPLES

53
265

- ③ DECLARE Q ARRAY(50) SCALAR DOUBLE INITIAL(10#1, 10#2,
10#3, 10#4, 10#5);

MATRIX\$(3DOUBLE,4,5)(Q\$(20 AT 6))

$$= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \end{bmatrix}$$

- ④ MATRIX(3#(1/2, SQRT(2)/2, SQRT(3)/2))

$$= \begin{bmatrix} 1/2 & 1/2\sqrt{2} & 1/2\sqrt{3} \\ 1/2 & 1/2\sqrt{2} & 1/2\sqrt{3} \\ 1/2 & 1/2\sqrt{2} & 1/2\sqrt{3} \end{bmatrix}$$

NOTE THAT THE REPETITION SYMBOL '#', PREVIOUSLY INTRODUCED
IN INITIAL LISTS, IS ALSO APPLICABLE HERE.

ASSIGNMENTS

51
266

THERE ARE 3 CLASSES OF ASSIGNMENTS DEPENDING ON WHETHER THE LEFT-HAND SIDE OF THE STATEMENT IS:

- ① ARITHMETIC (MATRIX, VECTOR, INTEGER, OR SCALAR)
- ② CHARACTER
- ③ BIT/BOOLEAN

GENERAL FORM: $L = R$

WHERE THE RECEIVER L IS A (POSSIBLY SUBSCRIPTED) DATA ITEM AND R IS EITHER A DATA ITEM OR AN EXPRESSION.

ARITHMETIC ASSIGNMENTS: WE WILL CONSIDER LEFT-HAND SIDES OF MATRICES, VECTORS, INTEGERS, AND SCALARS IN TURN.

MATRIX

MATRIX = MATRIX

MATRIX = 0 (CREATES A NULL MATRIX)

NOTE: MATRIX = 6 IS INVALID!

BOTH LEFT AND RIGHT MATRICES MUST MATCH IN ROW AND COLUMN DIMENSIONS. PRECISIONS, HOWEVER, NEED NOT MATCH!

3 39

52
267

ASSIGNMENTS (CON'T.)

EXAMPLES:

M1 IS 3x3 $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

M2 IS 2x2 $\begin{bmatrix} 20 & 40 \\ 60 & 80 \end{bmatrix}$

M3 IS 2x3 $\begin{bmatrix} -1 & -2 & -3 \\ -4 & -5 & -6 \end{bmatrix}$

M1 = 0 RESULTS IN $\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

M1 = 6 ILLEGAL

M1 = MATRIX(2,2,2,3,3,3,4,4,4)

RESULTS IN $\begin{bmatrix} 2 & 2 & 2 \\ 3 & 3 & 3 \\ 4 & 4 & 4 \end{bmatrix}$

M2 = M3 ILLEGAL

M2 = -M3\$(*, 2 AT 1)

RESULTS IN $\begin{bmatrix} +1 & +2 \\ +4 & +5 \end{bmatrix}$

M1\$(2 AT 2, *) = M3 RESULTS IN $\begin{bmatrix} 1 & 2 & 3 \\ -1 & -2 & -3 \\ -4 & -5 & -6 \end{bmatrix}$

ASSIGNMENTS (CON'T.)

52
268

VECTOR

VECTOR = VECTOR

VECTOR = 0 (CREATES A ZERO VECTOR)

NOTE: VECTOR = 6 IS INVALID!

BOTH LEFT AND RIGHT VECTORS MUST MATCH IN LENGTHS. PRECISIONS, HOWEVER, NEED NOT MATCH!

EXAMPLES:

V1 IS $\begin{bmatrix} -5 \\ 10 \\ 15 \end{bmatrix}$ V2 IS $\begin{bmatrix} 2 \\ 4 \\ 6 \\ 8 \end{bmatrix}$ V3 IS $\begin{bmatrix} 100 \\ -200 \end{bmatrix}$

V1 = 0 RESULTS IN V1 = $\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$

V1 = 6 ILLEGAL

V1 = V2 ILLEGAL

V1 = V2\$(3 AT 2) RESULTS IN

$\begin{bmatrix} 4 \\ 6 \\ 8 \end{bmatrix}$

V1 = V3 ILLEGAL

V1\$(2 AT 2) = V3 RESULTS IN

$\begin{bmatrix} -5 \\ 100 \\ -200 \end{bmatrix}$

52
269

ASSIGNMENTS (CON'T.)

INTEGER/SCALAR

THIS HAS ALREADY BEEN COVERED IN SOME DETAIL. WE WILL MERELY POINT OUT THE FOLLOWING:

- ① INTEGER = INTEGER
- ② INTEGER = SCALAR
- ③ SCALAR = INTEGER
- ④ SCALAR = SCALAR

* IN TYPES ② AND ③ A CHANGE OF DATA WILL BE MADE. SCALARS ARE CONVERTED TO INTEGERS BY ROUNDING.

EXAMPLES:

LET I BE AN INTEGER;
 S BE A SCALAR,
 M A 3x3 MATRIX.

THEN

I = 3;

RESULTS IN I = 3

I = 11.2;

RESULTS IN I = 11

I = 11.9;

RESULTS IN I = 12

S = 16.4;

RESULTS IN S = 16.4

M\$(3,*) = 5; ILLEGAL

M\$(3,3) = 5; RESULTS IN

X	X	X
X	X	X
X	X	5

3-42

ASSIGNMENTS (CON'T.)

32
270

BOOLEAN ASSIGNMENTS

BOOLEAN = BOOLEAN EXPRESSION

EXAMPLES:

DECLARE BOOLEAN, B1, B2, B3, B4, B5;

. . .

B1 = TRUE;	(SAME AS B1 = ON)
B2 = OFF;	(SAME AS B2 = FALSE)
B3 = BIN'1';	(SAME AS B3 = TRUE)
B4 = BIN'0';	(SAME AS B4 = FALSE)
B5 = B1;	B5 ≡ TRUE
B5 = B1 & B2;	B5 ≡ FALSE
B5 = \neg B3 \neg B4;	B5 ≡ TRUE
B5 = B2 OR B3;	B5 ≡ TRUE
B5 = B1 AND B3;	B5 ≡ TRUE
B5 = NOT B1;	B5 ≡ FALSE

3-43

ASSIGNMENTS (CON'T.)

52
271

MULTIPLE ASSIGNMENTS

MULTIPLE ASSIGNMENTS ARE RECOMMENDED BECAUSE THEY REDUCE THE NUMBER OF SEPARATE STATEMENTS OCCURRING IN A PROGRAM, AND THUS ENHANCE READABILITY.

FORM:

$$L_1, L_2, L_3, \dots = R$$

REQTS: EACH $L_i = R$ MUST BE A LEGAL TYPE OF ASSIGNMENT.

NOTE: THE EXACT ORDER IN WHICH THE ASSIGNMENTS ARE MADE IS NOT EASILY PREDICTED.

EXAMPLES

LET M1 BE A 2x2 MATRIX, M2 A 3x3 MATRIX, S1 AND S2 SCALARS, AND I1 AND I2 INTEGERS.

MULTIPLE ASSIGNMENT

① IF S1 = 6 THEN M1, M2, S1, S2, I1, I2 = 0;

ASSIGNMENTS (CON'T.)

33
272

CONTRAST THIS WITH:

IF S1 = 6 THEN DO,

M1 = 0,

M2 = 0,

S1 = 0,

S2 = 0,

I1 = 0,

I2 = 0,

END,

② M1, S1 = 6, IS ILLEGAL BECAUSE M1 = 6 IS ILLEGAL;

M1, S1 = 0, IS OK.

③ POTENTIAL HAZARD:

M1\$(I1, I2), I1 = I2,

ASSUME I1 = 2, I2 = 3 BEFORE ASSIGNMENT, COMPILER IS FREE TO
PICK BEST ORDER ON A MULTIPLE ASSIGNMENT. EITHER

M1\$(2, 3) OR M1\$(3, 3)

WILL BE ASSIGNED IN THIS CASE.

CONDITIONALS

SIMPLE IF STATEMENT

IF *exp* THEN stmt ;

WHERE *exp* IS EITHER A BOOLEAN EXPRESSION (I.E., EVALUATED AS TRUE OR FALSE) OR A RELATIONAL EXPRESSION (I.E., $A = 0$). THE KEY POINT IS THAT *exp* MUST BE SOMETHING THAT CAN BE EVALUATED AS TRUE OR FALSE.

NOTES:

- ① stmt IS EXECUTED ONLY IF *exp* EVALUTES TO TRUE.
- ② stmt CAN HAVE A STATEMENT LABEL.
- ③ IF *exp* IS FALSE, stmt IS BYPASSED.

51
273

52
274

CONDITIONALS (CON'T.)

EXAMPLES (SIMPLE IF)

ASSUME B1, B2, B3 ARE BOOLEANS,
S1, S2, S3 ARE SCALARS.

① IF B1 & B2 THEN } NOTE THAT THE "IF" PART AND "TRUE" PART
S1, S2 = 0) } ARE 2 STMTS.

② IF \neg B3 THEN
B1 = B2;

③ IF B1 & B2 THEN
IF S1 = 6 THEN
IF \neg B3 | B2 THEN
S1 = S2;

NOTE: IF STATEMENTS CAN BE NESTED.

④ IF S1 < 4 | S2 > 6 THEN
DO,
S1 = SIN(S2)**2;
S2 = S2 + S3;
END;

ENTIRE STATEMENT
GROUP IS TRUE PART

52
275

CONDITIONALS (CON'T.)

AUGMENTED IF STATEMENT

THIS IS LIKE A SIMPLE IF EXCEPT THAT IT HAS AN ELSE CLAUSE THAT IS EXECUTED ONLY IF THE "exp" IS FALSE.

AN AUGMENTED IF CAN BE PLACED WITHIN NESTED SIMPLE IFs, BUT AN AUGMENTED IF CAN NEVER NEST INSIDE ANOTHER (WITHOUT A STATEMENT GROUP ACTING AS AN INSULATOR).

EXAMPLES (AUGMENTED IF)

ASSUME B1, B2, B3 ARE BOOLEANS,
S1, S2, S3 ARE SCALARS,
I1, I2, I3 ARE INTEGERS.

- ① IF B1 THEN
 $S1 = S2**2 + S3**2;$
ELSE S1 = S1I(S2 S3);

CONDITIONALS (CON'T.)

- ② IF S1 < S2 THEN
ZERO_IT: S1 = 0;
ELSE S1 = S2**2; } TRUE PART CAN HAVE A LABEL

- ③ IF B1 THEN
B2 = FALSE;
ELSE TROUBLE: B2 = TRUE; } FALSE PART CAN HAVE A LABEL, ALSO

- ④ IF B1 & B2|B3 THEN
IF S1 < 5|S1 > 9 THEN
IF S1 = 0 & S2 = 4 THEN
I1 = 6;
ELSE I1 = 10; } SIMPLE IFS
AUGMENTED IF

NOTE: DANGLING ELSEs ARE NOT A PROBLEM BECAUSE ELSE GOES WITH THE INNERMOST IF.

CONDITIONALS (CON'T.)

S3
277

⑤ IF B1|B2 THEN
 B3 = FALSE;
ELSE IF B2 & B3 THEN
 B3 = TRUE;
ELSE IF S1 = 0 THEN
 S1 = 1;

ELSE CLAUSE MAY ALSO BE AN IF STMT

⑥ IF S1 = 4 THEN
 S2 = 1;
ELSE IF S1 = 6 THEN
 S2 = 2;
ELSE IF S1 = 8 THEN
 S2 = 3;

← CONTROL JUMPS HERE AFTER COMPLETION OF ANY TRUE PART.
ANY LABELS PUT ON TRUE PART OR FALSE PART (ELSE) MAY NOT BE BRANCHED
TO FROM ANYWHERE OUTSIDE OF THE IF STATEMENT.

CONDITIONALS (CON'T.)

32
278

RELATIONAL EXPRESSIONS

RELATIONAL EXPRESSIONS CAN BE USED IN CONDITIONAL STATEMENTS (E.G., IFs) AS CAN BOOLEAN EXPRESSIONS. A RELATIONAL EXPRESSION IS SOME SORT OF A COMPARISON THAT WILL EVALUATE TO A TRUE OR A FALSE CONCLUSION.

RELATIONALS ARE EXPRESSIONS LINKED BY COMPARATIVE OPERATORS: $>$, $<$, $<=$, $>=$, $\neg >$, $\neg <$, $=$, $\neg =$. RELATIONALS, IN TURN, MAY BE COMBINED USING THE 3 BOOLEAN OPERATORS: $\&$, $|$, AND \cdot . RELATIONALS, HOWEVER, ARE NOT QUITE BOOLEANS.

COMPARATIVE OPERATORS ARE LOOSELY GROUPED INTO 2 CLASSES:

CLASS I: $>$, $<$, $<=$, $\neg >$, $>=$, $\neg <$

CLASS II: $=$, $\neg =$

CONDITIONALS (CON'T.)

53
279

CLASS II OPERATORS (=, \neq) ARE THE ONLY ONES THAT CAN BE USED WITH: VECTOR, MATRIX, BOOLEAN, AND BIT STRING DATA TYPES.

- VECTOR AND MATRIX COMPARES ARE PERFORMED ELEMENT-BY-ELEMENT.
- VECTORS AND MATRICES MUST HAVE THE SAME "SHAPES", I.E., VECTORS MUST BE OF IDENTICAL LENGTHS, AND MATRICES MUST HAVE THE SAME NUMBER OF ROWS AND COLUMNS.

EXAMPLES:

$$\text{LET } V1 = \begin{bmatrix} 1 \\ 4 \\ 6 \end{bmatrix} \quad V2 = \begin{bmatrix} 4 \\ 6 \end{bmatrix} \quad M1 = \begin{bmatrix} 1 & 2 \\ 3 & 5 \end{bmatrix} \quad M2 = \begin{bmatrix} 1 & 4 & 6 \\ 2 & 1 & 3 \end{bmatrix}$$

- | | |
|----------------------------------|---|
| (1) $V1 = V2$ | ILLEGAL (VECTORS \neq IN LENGTH) |
| (2) $M1 = M2$ | ILLEGAL (MATRICES HAVE DIFFERENT COLUMN CT) |
| (3) $V1 = M2\$(1, *)$ | TRUE |
| (4) $V1\$(2 \text{ AT } 2) = V2$ | TRUE |
| (5) $V2 = M1\$(*, 1)$ | FALSE |

53
280

CONDITIONALS (CON'T.)

FOR INTEGERS, SCALARS, AND CHARACTER STRINGS, BOTH CLASS I AND CLASS II CONDITIONAL OPERATORS CAN BE USED.

- AS WE WOULD EXPECT, IN AN INTEGER/SCALAR COMPARISON, THE INTEGERS ARE FIRST CONVERTED TO SCALARS.
- CHARACTER STRING COMPARES ALLOW US TO SORT STRINGS ALPHABETICALLY.

SUPPOSE I1 = 5 I2 = -3 I3 = 0
 S1 = -80 S2 = 6.5 S3 = 10.3
 C1 = 'Z' C2 = 'ABC' C3 = 'B'

THEN

I1 > I2	= TRUE
I1 < I3	= FALSE
I1 = 5	= TRUE
C1 < C2	= FALSE
C2\$2 = C3	= TRUE
I1 < S2	= TRUE
I3 < C1	ILLEGAL
(I3-I2) < I1	= TRUE
I1**2 < -S1	= TRUE
C2 < C3	= TRUE

33
281

CONDITIONALS (CON'T.)

- COMPARATIVE OPERATIONS CAN BE COMBINED (AS IF THEY WERE BOOLEAN OPERANDS), USING THE BOOLEAN OPERATORS $\&$, $|$, AND \neg . STILL, COMPARATIVE OPERATIONS ARE NOT BOOLEANS AND THEY CANNOT BE MIXED WITH BOOLEANS IN A CONDITIONAL STATEMENT.

EXAMPLES

ASSUME:	B1, B2, B3	BOOLEANS,
	I1, I2, I3	INTEGERS,
	S1, S2, S3	SCALARS,
	V1, V2, V3	VECTORS.

LEGAL

IF B1 | \neg B2 | (B2 & B3) THEN ...

IF V1 = V2 | S1 < S2 THEN ...

IF I1 + I2 < 0 | S1 < S2**2 THEN ...

IF V1 - V2 = V3 & S1 < S2 + 2 THEN ...

ILLEGAL

IF B1 | B2 | S1 < S2 THEN ...

IF \neg B2 | S1 = S2 THEN ...

IF V1 = V2 & B3 THEN ...

CONDITIONALS (CON'T.)

53
282

BUT WE CAN GET OUT OF TROUBLE BY TURNING THE BOOLEAN EXPRESSIONS
INTO RELATIONAL EXPRESSIONS:

IF (B1|B2) = TRUE | S1 < S2 THEN ...

IF B2 = FALSE | S1 = S2 THEN ...

IF V1 = V2 & B3 = TRUE THEN ...

PRECEDENCE

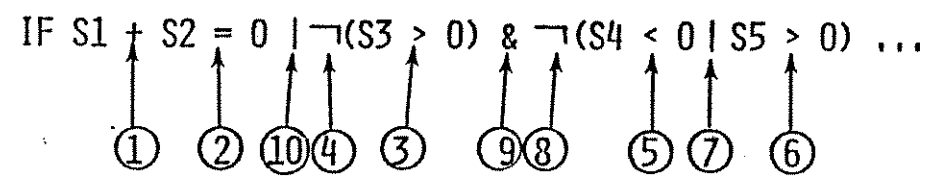
- HI
↑
- 1 ARITHMETIC OPERATIONS (E.G., /, *, ., +, -, **, MULT. (BLANK)).
 - 2 COMPARATIVE OPERATIONS (E.G., <, >, =, ≠, ≠, >, <, >=, <=).
 - 3 NOT (¬)
 - 4 AND (&)
 - 5 OR (|)
- LOGICAL OPERATIONS {

41-10

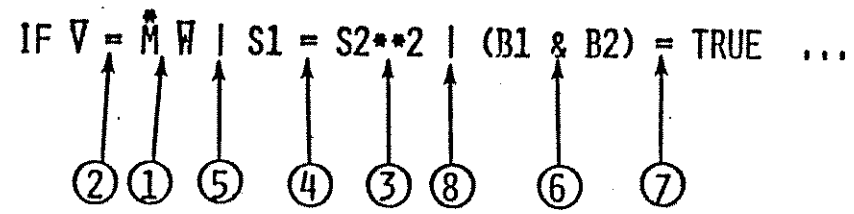
CONDITIONALS (CON'T.)

S3
283

EXAMPLES (PRECEDENCE IN RELATIONAL EXPRESSION)



ARROWS INDICATE ORDER OF EXECUTION OF OPERATIONS



52
284

CONDITIONALS (CON'T.)

LABELS AND BRANCHES

LABELS ARE NAMES CHOSEN BY THE PROGRAMMER AND ATTACHED TO EXECUTABLE STATEMENTS. WE CAN DISTINGUISH BETWEEN BLOCK LABELS (MANDATORY LABELS ON BLOCKS),

- ALPHA: PROGRAM;
- BETA: COMPOOL;
- GAMMA: PROCEDURE;
- DELTA: FUNCTION;
- EPSILON: ~~TASK;~~
- ZETA: ~~UPDATE;~~

AND STATEMENT LABELS. ONLY THE LATTER MAY BE BRANCHED TO.

- A STATEMENT MAY HAVE 0, 1, OR MORE LABELS -- THERE IS NO CORE OR CPU PENALTY FOR SUCH LABELS AND THEY MAY BE USEFUL FOR DIAGNOSTIC PURPOSES (AS WILL BE SEEN).

CONDITIONALS (CON'T.)

53
285-

EXAMPLES

- (1) SET_X: X = SIN(Y);
- (2) SET_X: BOMB_OUT: X = SQRT(-1);
- (3) BIT_LOOP: DO FOR I = 1 TO 10000;
- (4) DEPART: CLOSE ALPHA;
- (5) IF A = 3 THEN
CALL_PROC: CALL UPDATER(A);
ELSE DONE: GO TO END_ALL;
- (6) CHECK_FAIL: IF Z = 0 THEN
CALL FLASH_DISPLAY(6);

GO TO STATEMENT

GENERIC FORM:

GO TO *label*;

THIS DOES THE OBVIOUS -- CONTROL IS TRANSFERRED TO THE STATEMENT WITH LABEL *label* (PROVIDED STRINGENT CONDITIONS ARE MET).

CONDITIONALS (CON'T.)

52
286

ALTHOUGH STRUCTURED PROGRAMMING PRACTICE DISCOURAGES USE OF "GO TO" STATEMENTS, THEY ARE SOMETIMES PREFERABLE TO A DAY OF WORK TO ELIMINATE ONE. THE FOLLOWING EXAMPLE, HOWEVER, SHOWS HOW EASY HAL/S MAKES THE ELIMINATION OF GO TO'S.

EXAMPLE

```
IF VALUE < LOW_LIMIT THEN GO TO LOW;
IF VALUE > HI_LIMIT THEN GO TO HI;
IN_LIMITS: GOOD_FLAG = TRUE;
           OUT_VAL = VALUE;
           GO TO FINISH;
LOW: GOOD_FLAG = FALSE;
     OUT_VAL = LOW_LIMIT;
     GO TO FINISH;
HI:  GOOD_FLAG = FALSE;
     OUT_VAL = HI_LIMIT;
FINISH:
```

CONDITIONALS (CON'T.)

53
287

BECOMES

```
IF VALUE < LOW_LIMIT THEN
  DO,
    GOOD_FLAG = FALSE,
    OUT_VAL = LOW_LIMIT,
  END,
ELSE IF VALUE > HI_LIMIT THEN
  DO,
    GOOD_FLAG = FALSE,
    OUT_VAL = HI_LIMIT,
  END,
ELSE DO,
  GOOD_FLAG = TRUE,
  OUT_VAL = VALUE,
END,
```

91-15

3/
288

STATEMENT GROUPS

A STATEMENT GROUP IS A SET OF HAL/S STATEMENTS THAT ARE CONSIDERED AS A UNIT FOR THE PURPOSES OF CONDITIONAL OR REPETITIVE EXECUTION. STATEMENT GROUPS BEGIN WITH A "DO" STATEMENT, CLOSE WITH AN "END" STATEMENT, AND CAN CONTAIN OTHER STATEMENT GROUPS (OR EVEN CODE BLOCKS) NESTED WITHIN. THE "DO" STATEMENT IS CONSIDERED TO BE AN EXECUTABLE STATEMENT (ALTHOUGH CODE IS NOT ALWAYS GENERATED FOR IT) AND AS SUCH CAN POSSESS A STATEMENT LABEL. THE "DO" STATEMENT HAS THE FORM:

DO (*control*);

(*Control*) IS OPTIONAL AND WILL BE DESCRIBED NEXT. IF (*Control*) IS OMITTED THE STATEMENT GROUP DO ... END IS EXECUTED ONCE.

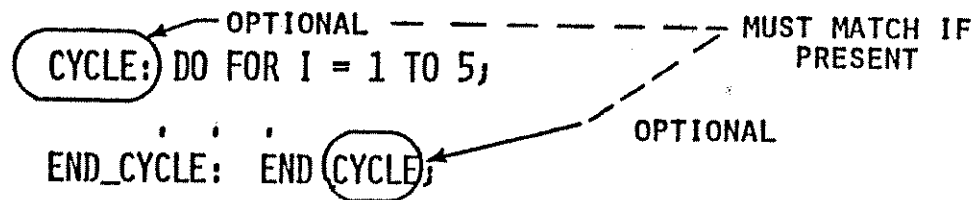
STATEMENT GROUPS (CON'T.)

52
289

THE "END" STATEMENT IS ALSO CONSIDERED TO BE AN EXECUTABLE STATEMENT (AGAIN, IT MIGHT NOT ALWAYS RESULT IN CODE) AND MAY BE LABELLED. THE GENERIC FORM OF THE "END" STATEMENT IS:

END *label*;

label IS OPTIONAL, BUT IF IT IS PRESENT IT MUST MATCH THE LABEL ON THE CORRESPONDING "DO" STATEMENT. THUS,



USE OF THESE OPTIONAL LABELS IS RECOMMENDED FOR THE FOLLOWING REASONS:

- (1) ALLOWS CROSS-CHECKING BY THE COMPILER THAT ENSURES EVERY "DO" IS PROPERLY CLOSED.
- (2) A LABEL PROVIDES A FIXED REFERENCE POINT FOR DIAGNOSTICS.
- (3) MEANINGFUL LABELS INCREASE PROGRAM READABILITY.

51
290

STATEMENT GROUPS (CON'T.)

DO WHILE

```
DO WHILE (condition) ;  
    . . .  
END;
```

- THE STATEMENT GROUP IS REPETITIVELY EXECUTED AS LONG AS THE (condition) REMAINS TRUE.
- IF (condition) IS FALSE, THEN THE STATEMENT GROUP IS NOT EXECUTED AT ALL.
- (condition) IS A RELATIONAL OR A BOOLEAN EXPRESSION (NOT A MIXTURE OF COURSE) AND IS EVALUATED PRIOR TO EACH CYCLIC EXECUTION OF THE STATEMENT GROUP.
- WHEN (condition) BECOMES FALSE, CYCLIC EXECUTION HALTS AND CONTROL IS PASSED TO THE STATEMENT FOLLOWING THE "END" STATEMENT.

EXAMPLE

```
I = 50;  
DO WHILE I > 0;  
    TABLE$I = I**2;  
    I = I - 2;  
END;
```

4-18

STATEMENT GROUPS (CON'T.)

51
291

DO UNTIL

```
DO UNTIL (condition) ;  
    . . .  
END;
```

- THE STATEMENT GROUP IS EXECUTED CYCLICALLY UNTIL (*condition*) BECOMES TRUE (I.E., THE REVERSE OF DO...WHILE).
- THE TEST OF (*condition*) IS MADE AT THE END OF THE CYCLES SO THE STATEMENT GROUP WILL ALWAYS BE EXECUTED AT LEAST ONCE.
- AS WAS THE CASE WITH DO...WHILE, (*condition*) IS A RELATIONAL OR A BOOLEAN EXPRESSION.

LOOPS THAT WILL CYCLE FOREVER:

- (A) DO WHILE (TRUE;) ← THIS BOOLEAN EXPRESSION WILL NEVER BECOME FALSE.
 . . .
END;
- (B) DO UNTIL (FALSE;) ← THIS BOOLEAN EXPRESSION WILL NEVER BECOME TRUE.
 . . .
END;

4-19

STATEMENT GROUPS (CON'T.)

S/
292

DO FOR

THERE ARE 2 BASIC FORMS: ITERATIVE AND DISCRETE:

ITERATIVE

DO FOR (*loop var*) = (*initial*) TO (*final*) BY (*increment*);

- (*loop var*) IS AN UNARRAYED INTEGER OR SCALAR. IT IS THE CONTROL VARIABLE (THE VARIABLE COULD BELONG TO AN ARRAY, BUT IF SO ITS ARRAYNESS MUST BE SUBSCRIPTED AWAY).
- (*initial*), (*final*), AND (*increment*) ARE INTEGER OR SCALAR EXPRESSIONS.
- ON THE FIRST CYCLE, (*loop var*) WILL HAVE THE VALUE (*initial*).
- ON EACH SUCCEEDING CYCLE, (*loop var*) WILL BE AUGMENTED BY (*increment*).
- CYCLIC EXECUTION CONTINUES UNTIL (*loop var*) LIES OUTSIDE THE RANGE BOUNDED BY (*initial*) AND (*final*).

52
293

STATEMENT GROUPS (CON'T.)

- NOTE: ① INCREMENT IS ASSUMED TO BE +1 IF NOT SPECIFIED, I.E., IN THIS CASE THE "BY (*increment*)" IS OPTIONAL.
- ② (*initial*), (*final*), AND (*increment*) ARE EVALUATED ONCE (AND THEIR VALUES SAVED) PRIOR TO INITIATION OF THE LOOP, I.E., VALUES CANNOT BE CHANGED WITHIN THE LOOP.

EXAMPLES

EQUIVALENT

```
① DO FOR I = 1 TO 20,  
  . . .  
  END;  
DO FOR I = 1 TO 20 BY 1,  
  . . .  
  END;
```

```
② DO FOR I = 2J TO (J**2-1) BY 5,  
  . . .  
  J = J - 1; ← CANNOT AFFECT LOOP TERMINATION.  
  . . .  
  END;
```


STATEMENT GROUPS (CON'T.)DISCRETE

DO FOR (*loop var*) = $exp_1, exp_2, \dots, exp_n$;

- (*loop var*) IS AN UNARRAYED INTEGER OR SCALAR -- THE CONTROL VARIABLE.
- THE LOOP WILL BE EXECUTED n TIMES, WITH (*loop var*) = exp_l ON THE l^{th} EXECUTION.

NOTE: EACH EXPRESSION exp_l IS EVALUATED JUST PRIOR TO USING IT, I.E., ON THE l^{th} EXECUTION.

EXAMPLES

① DO FOR I = 1, 3, 5, 9;

...
END;

② DO FOR S = SIN(30), SIN(45), SIN(60);

...
END;

③ J = 1
DO FOR I = J, J+2, J+4, J-6;

...
J = J(J+1);

...
END;

I = 1	1 ST CYCLE
4	2 ND CYCLE
10	3 RD CYCLE
36	4 TH CYCLE

STATEMENT GROUPS (CON'T.)

53
295

BOTH THE ITERATIVE AND DISCRETE "DO FOR" STATEMENTS MAY HAVE AN ADDITIONAL "WHILE" OR "UNTIL" CLAUSE ACTING AS A FURTHER EXECUTION QUALIFIER:

```
DO FOR I = 1 TO J WHILE I < 50;
```

```
...
```

```
END;
```

```
DO FOR I = 1 TO 10000 UNTIL J = 0;
```

```
...
```

```
IF A > 0 THEN J = J - 1;
```

```
...
```

```
END;
```

```
DO FOR I = 1, 3, 7, 9 WHILE B;
```

```
...
```

```
IF SIN(Q) < .4 THEN B = FALSE;
```

```
...
```

```
END;
```

STATEMENT GROUPS (CON'T.)

51
276

DO CASE -- USED TO SELECT ONE STATEMENT TO BE EXECUTED BASED ON A CALCULATION (ROUGHLY EQUIVALENT TO A FORTRAN COMPUTED GOTO).

FORM:

DO CASE (exp);

- (exp) IS AN INTEGER OR SCALAR EXPRESSION. (SCALAR WILL BE CONVERTED TO AN INTEGER BY ROUNDING).
- IF (exp) EVALUATES TO 1, 2, 3, ... K THEN THE 1ST, 2ND, 3RD, ... KTH STATEMENT (OR STATEMENT GROUP) FOLLOWING THE "DO CASE" WILL BE EXECUTED.
- IN THIS SIMPLE FORM OF THE "DO CASE" NO ERROR CHECKING IS PERFORMED, I.E., UNPREDICTABLE AND DISASTROUS BEHAVIOR WILL RESULT IF (exp) EVALUATES TO ≤ 0 OR $> K$.

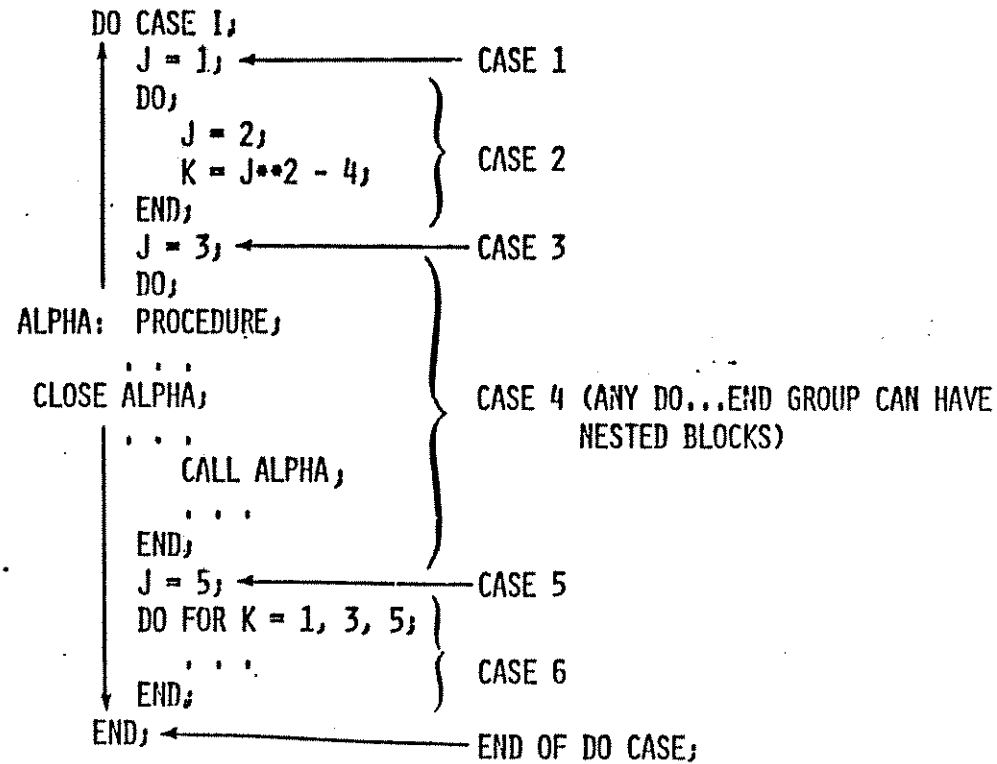
4-24

52
296A

STATEMENT GROUP (CON'T.)

A DO CASE IS POWERFUL BECAUSE A SPECIFIC CASE CAN RANGE FROM A SIMPLE STATEMENT TO A COMPLEX "DO...END" GROUP CONTAINING AN ARBITRARY NUMBER OF STATEMENTS.

EXAMPLE



4-25

STATEMENT GROUPS (CON'T.)

33
297

ERROR CHECKING ON THE CASE VARIABLE IS PERFORMED IF AN "ELSE" CLAUSE IS ADDED TO THE "DO CASE".

```
DO CASE (exp); ELSE stmt ;  
    (case 1)  
    (case 2)  
    . . .  
    (case K)  
END;
```

IF (exp) EVALUATES TO ≤ 0 OR $> K$ THEN stmt IS EXECUTED (WHICH COULD OF COURSE SIGNAL AN ERROR CONDITION).

- AN ELSE CLAUSE SHOULD ALWAYS BE PROGRAMMED IF THERE IS A POSSIBILITY THAT THE CASE VARIABLE COULD EVER MISBEHAVE!

NOTE: A DO CASE MAY HAVE A MAXIMUM OF 256 CASES.

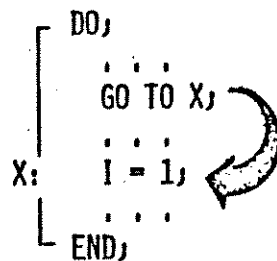
31
218

STATEMENT GROUPS (CON'T.)

BRANCHING (GO TO, EXIT, REPEAT)

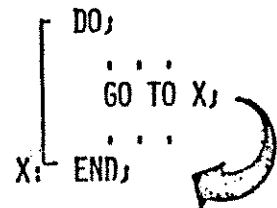
A STATEMENT GROUP, I.E. DO...END, DO WHILE, DO UNTIL, DO FOR, DO CASE, MAY BE BRANCHED OUT OF VIA A GO TO, BUT SUCH A GROUP MAY NOT BE BRANCHED INTO. BRANCHING WITHIN, OF COURSE, IS LEGAL.

LEGAL ①



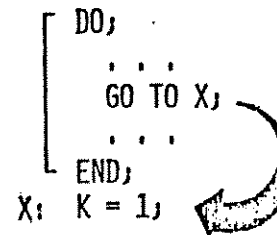
WITHIN

②



TO THE END

③



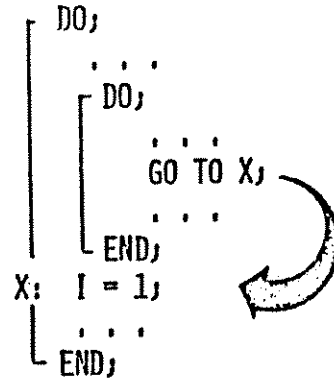
TO THE OUTSIDE

4-27

STATEMENT GROUPS (CON'T.)

LEGAL

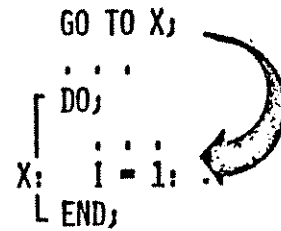
④



OK BECAUSE ALREADY
IN THE OUTER DO!

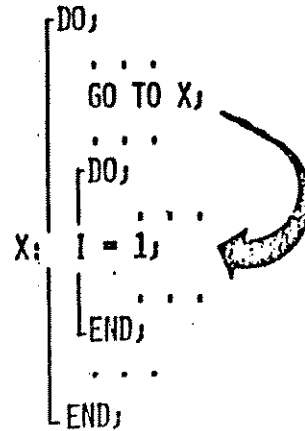
ILLEGAL

①



CANNOT BRANCH INTO

②



CANNOT BRANCH INTO
(EVEN IF NESTED)

STATEMENT GROUPS (CON'T.)

5/
300

SINCE BRANCHING WITHIN (OR OUT OF) "DO" GROUPS IS INVARIABLY NECESSARY, AND SINCE "GO TOs" ARE UNDESIRABLE IN STRUCTURED PROGRAMMING, HAL/S OFFERS TWO ALTERNATE CONSTRUCTIONS:

EXIT & REPEAT

EXIT

FORM 1: EXIT;

- CAUSES IMMEDIATE BRANCH OUT OF THE INNERMOST STATEMENT GROUP IN WHICH IT IS ENCLOSED.
- EXECUTION IS DIRECTED TO THE FIRST STATEMENT FOLLOWING THE "END" OF THE GROUP BRANCHED OUT OF.

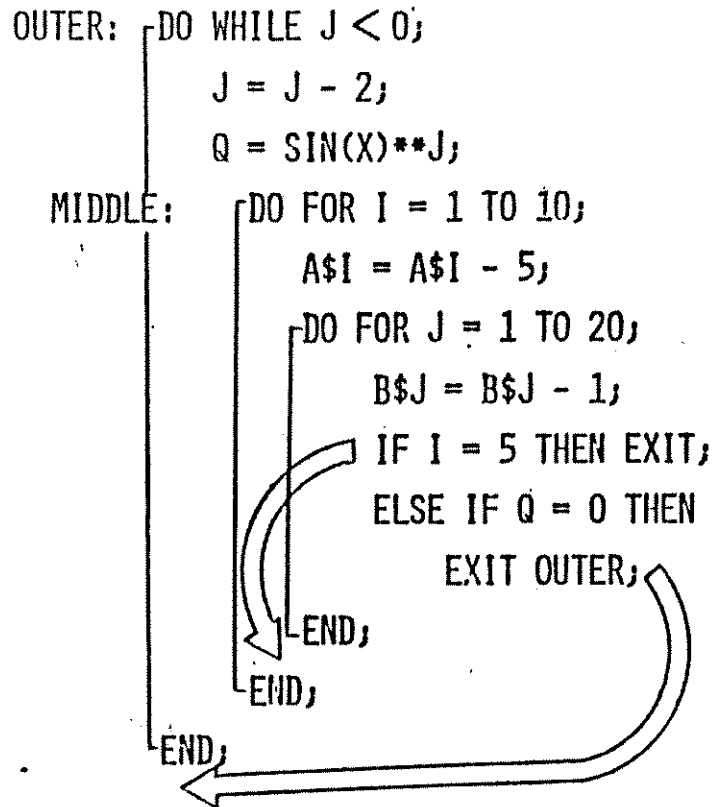
FORM 2: EXIT *label*;

- CAUSES BRANCH OUT OF THE ENCLOSING STATEMENT GROUP THAT POSSESSES THE LABEL *label* ON ITS "DO" STATEMENT.
- EXECUTION IS DIRECTED TO THE FIRST STATEMENT FOLLOWING THE "END" STATEMENT OF THAT GROUP.

4-25

STATEMENT GROUPS (CON'T.)

52
301



THIS SHOWS THE ADVANTAGES IN LABELING DO...END GROUPS. NOTE THAT THE EXIT STATEMENT ACTS LIKE A GO TO, BUT IS A MORE SECURE (AND STRUCTURED) CONSTRUCTION.

31
302

STATEMENT GROUPS (CON'T.)

REPEAT (MUST BE ENCLOSED WITHIN DO FOR, DO WHILE, OR DO UNTIL GROUP).

FORM 1: REPEAT;

- MUST BE ENCLOSED WITHIN A DO FOR, DO WHILE, OR DO UNTIL GROUP (I.E., A REPETITIVE GROUP).
- CAUSES IMMEDIATE BRANCH TO THE BEGINNING OF THE INNERMOST REPETITIVE GROUP.

FORM 2: REPEAT *label*;

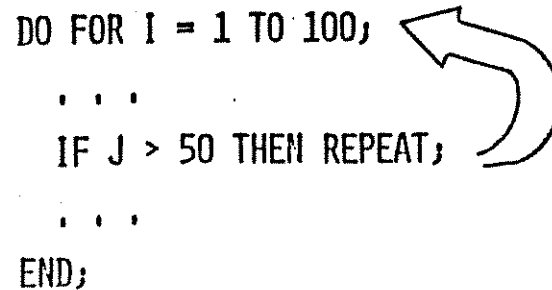
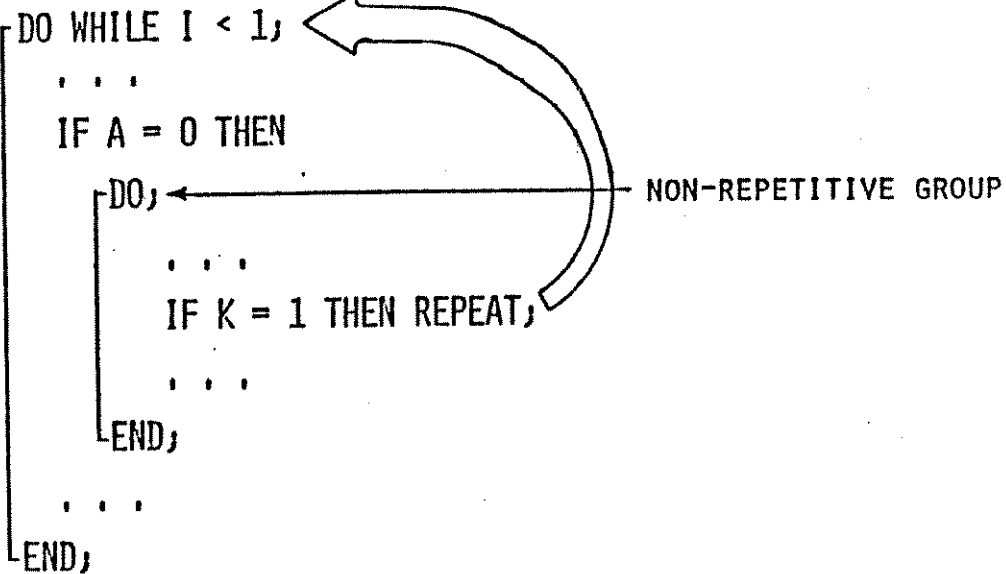
- MUST BE ENCLOSED WITHIN A REPETITIVE GROUP THAT HAS LABEL *label* ON ITS "DO" STATEMENT.
- CAUSES IMMEDIATE BRANCH TO THE BEGINNING OF SAID REPETITIVE GROUP.

STATEMENT GROUPS (CON'T.)

EXAMPLES

32
303

① REPEAT



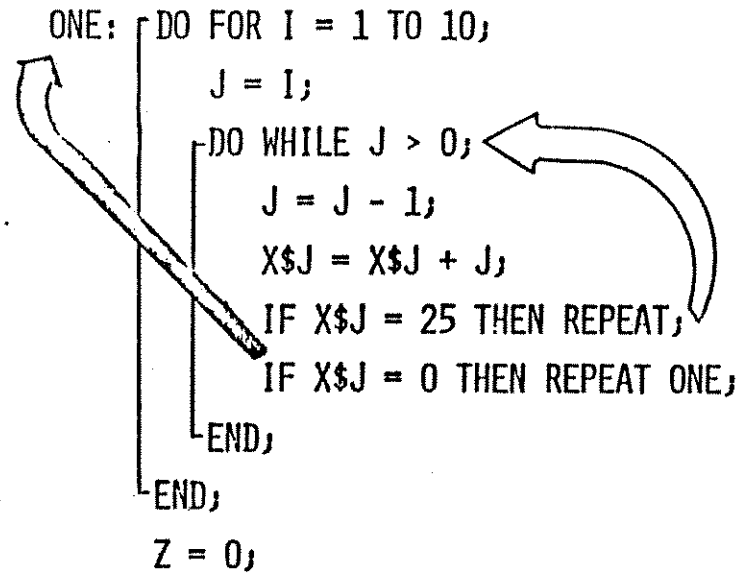
4-1-52

STATEMENT GROUP (CON'T.)

53
304

② REPEAT *label*

```
ONE: DO FOR I = 1 TO 10;  
      J = I;  
      DO WHILE J > 0;  
        J = J - 1;  
        X$J = X$J + J;  
        IF X$J = 25 THEN REPEAT;  
        IF X$J = 0 THEN REPEAT ONE;  
      END;  
    END;  
  Z = 0;
```



4-3

PROCEDURES AND FUNCTIONS

S/
305-

PROCEDURES AND FUNCTIONS ARE THE TWO MOST COMMON CODE BLOCKS. IF PROCEDURES/FUNCTIONS ARE COMPILED SEPARATELY THEY ARE CALLED COMSUBs (COMMON SUBROUTINES). OTHERWISE, THEY ARE NESTED IN A PROGRAM.

- THE MAXIMUM NESTING DEPTH OF CODE BLOCKS IS 16.
- THE MAXIMUM NUMBER OF BLOCKS (CODE AND DATA) IN A COMPILATION IS 256 (INCLUDING EXTERNAL TEMPLATES).
- PROCEDURES/FUNCTIONS MAY NEST WITHIN PROGRAM, TASK, UPDATE, PROCEDURE, AND FUNCTION BLOCKS (I.E., ALMOST ANYTHING).
- A PROGRAM BLOCK CANNOT NEST WITHIN A PROGRAM BLOCK.

THIS IS NOT NESTING

```
P1: EXTERNAL PROGRAM;  
  CLOSE P1;  
D VERSION 6  
P2: PROGRAM  
  DECLARE NN NAME PROGRAM  
    INITIAL(NAME(P1));  
  . . .  
  CLOSE P2;
```

THIS IS

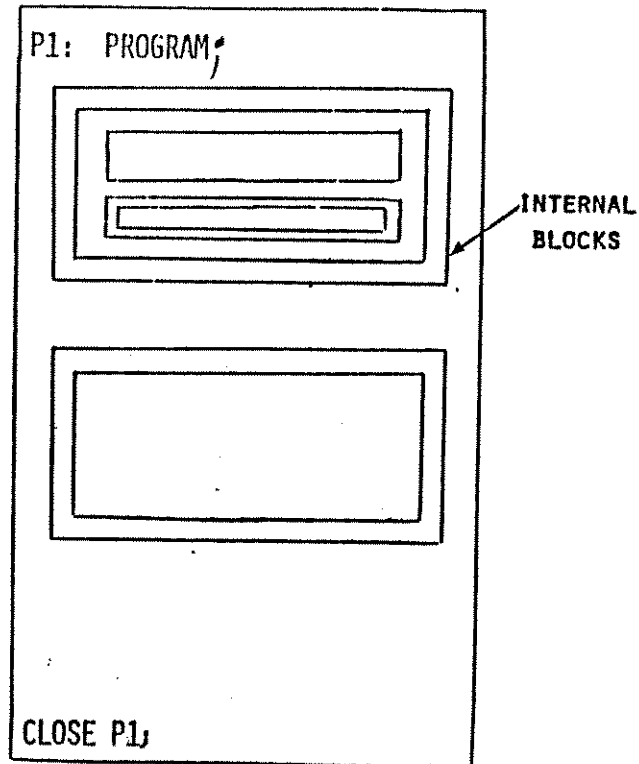
```
P1: PROGRAM;  
  . . .  
  P2: PROGRAM;  
    . . .  
    CLOSE P2;  
  . . .  
  CLOSE P1;
```

4-39

PROCEDURES AND FUNCTIONS (CON'T.)

52
306

THE MAXIMUM NEST DEPTH HERE IS 4.



PARAMETERS

PROGRAM BLOCKS CANNOT HAVE PARAMETERS -- ONLY PROCEDURE AND FUNCTION BLOCKS MAY, (THE OTHER BLOCKS NOT COVERED SO FAR -- UPDATE BLOCKS AND TASK BLOCKS -- ARE ALSO PARAMETERLESS.)

PARAMETERS ARE EITHER INPUT OR ASSIGN PARAMETERS. OF THESE, ONLY A PROCEDURE CAN HAVE AN ASSIGN PARAMETER.

PROCEDURES AND FUNCTIONS (CON'T.)

53
308

FUNCTION BLOCK

label: FUNCTION(i_1, i_2, \dots, i_n) *attributes*;

MANDATORY
IDENTIFIER

(OPTIONAL)
INPUT PARAMETERS

THE TYPE AND
PRECISION OF THE
DATA RETURNED BY
THE FUNCTION

NOTE: IF (*attributes*) IS OMITTED, THE FUNCTION IS PRESUMED TO RETURN A SINGLE PRECISION SCALAR (I.E., THE SAME DEFAULTS AS IN A DECLARE STATEMENT).

PROCEDURE AND FUNCTION BLOCKS MUST HAVE A CLOSE STATEMENT AS THE FINAL STATEMENT. THE CLOSE STATEMENT IS AN EXECUTABLE STATEMENT OF THE FORM:

CLOSE;
OR CLOSE *label*;

IF (*label*) IS PRESENT, IT MUST MATCH THE LABEL ON THE PROCEDURE/
FUNCTION BLOCK HEADER.

PROCEDURES AND FUNCTIONS (CON'T.)

51
309

A PROCEDURE IS EXITED EITHER BY FALLING INTO (~~OR BRANCHING TO~~) THE CLOSE STATEMENT, OR BY USE OF A RETURN STATEMENT. A FUNCTION MUST EXIT VIA A RETURN (exp) STATEMENT, I.E., NOT ONLY A RETURN, BUT ONE THAT RETURNS A VALUE OF THE PROPER DATA TYPE AND PRECISION (IMPLICIT CONVERSIONS ALLOWED).

-
- PARAMETER DECLARATIONS SHOULD ALWAYS PRECEDE LOCAL DATA DECLARATIONS. IN THE CASE OF COMSUBS THIS IS REQUIRED.

INPUT PARAMETERS - PASSED BY "VALUE" IN THE CASE OF INTEGERS AND SCALARS. ALL OTHER DATA PASSED BY "NAME" (I.E., REFERENCE).

ASSIGN PARAMETERS - ALWAYS PASSED BY "NAME".

THE CONCEPT OF A STACK, WHICH WILL BE DISCUSSED IN DETAIL LATER, IS A KEY PART OF THE MACHINERY FOR PARAMETER PASSAGE.

51-332

PROCEDURES AND FUNCTIONS (CON'T.)

S2
310

EXAMPLE ①

```

      INPUT          ASSIGN
      PARMS         PARMS
ALPHA: PROCEDURE(I, S1, V) ASSIGN(S2, W);
  DECLARE INTEGER, I;
  DECLARE SCALAR, S1, S2 DOUBLE;
  DECLARE V VECTOR, W MATRIX;
  DECLARE VECTOR, X, Y;
  DECLARE SCALAR, T, U;
  DECLARE MATRIX, A, B, C;
  IF S2 = 0 THEN
    DO,
      S2 = 0,
      W = S1 V V + I2 X Y,
    END;
  ELSE
    DO,
      S2 = 1,
      W = S1 V X + I2 V Y,
    END;
  IF I = 0 THEN
    W = W + A B + I C,
  CLOSE ALPHA;

```

PARAMETER DECLARATIONS

LOCAL DATA DECLARATIONS

AN ASSIGN PARM CAN BE USED AS INPUT AS WELL AS OUTPUT.

CAUSES A RETURN TO THE CALLER.

PROCEDURES AND FUNCTIONS (CON'T.)

53
311

EXAMPLE (2)

```
      INPUT      FUNCTION WILL RETURN A 3x3
      PARM      DOUBLE PRECISION MATRIX
BETA: FUNCTION(M) MATRIX DOUBLE;
      DECLARE M MATRIX DOUBLE;          } PARAMETER DECLARATION
      DECLARE M1 MATRIX DOUBLE         }
      INITIAL(3#1, 3#2, 3#3);           } LOCAL DATA DECLARATIONS
      DECLARE M2 MATRIX DOUBLE
      INITIAL(3#2, 3#1, 3#3);
      RETURN  $\overset{*}{M}$   $\overset{*}{M1}$  +  $\overset{*}{M2}$   $\overset{*}{M2}$ ;
CLOSE BETA; ← WILL NEVER BE EXECUTED
                BECAUSE OF RETURN STMT.
```

NOTE: A RUN-TIME ERROR WILL OCCUR IF THE CLOSE STATEMENT IS EVER REACHED IN A FUNCTION.

PROCEDURES ARE ENTERED VIA A CALL STATEMENT THAT MAY OR MAY NOT HAVE INPUT OR ASSIGN ARGUMENTS -- DEPENDING ON WHETHER THE PROCEDURE HAS INPUT OR ASSIGN PARAMETERS.

```
                ASSIGN
                ARGUMENTS
CALL ALPHA(J, X, V1) ASSIGN(T, W);
      INPUT
      ARGUMENTS
```

PROCEDURES AND FUNCTIONS (CON'T.)

81
312

FUNCTIONS ARE ENTERED BY INVOCATION, I.E., BY USING THE NAME OF THE FUNCTION AS THOUGH IT WERE A DATA TYPE. IF THE FUNCTION HAS INPUT PARAMETERS, THEN THE INVOCATION MUST SPECIFY CORRESPONDING INPUT ARGUMENTS.

$M3 = \text{BETA}(M2);$

FUNCTION BETA TAKES MATRIX M2 AS AN INPUT ARGUMENT AND RETURNS A NEW MATRIX THAT IS THEN ASSIGNED TO M3.

IF THE FUNCTION HAS NO PARAMETERS, THEN THE PARENTHEZIZED LIST IS OMITTED.

- MORE FREEDOM (E.G., MISMATCH OF DATA TYPES AND/OR PRECISION) IS ALLOWED FOR INPUT ARGUMENTS \longleftrightarrow INPUT PARAMETERS THAN IS ALLOWED FOR ASSIGN ARGUMENTS \longleftrightarrow ASSIGN PARAMETERS.

PROCEDURES AND FUNCTIONS (CON'T.)

INPUT PARAMETERS

52
313

REMEMBER THAT PARAMETERS ARE DECLARED DATA ITEMS, BUT ARGUMENTS
(INPUT, AT LEAST) CAN BE EXPRESSIONS:

```
ALPHA: PROCEDURE(I, S);  
  DECLARE I INTEGER, S SCALAR;  
  . . .  
  CLOSE ALPHA;  
CALL ALPHA( $K^3 + 3 + \text{SIN}(7)$ ,  $\text{LOG}(\text{COSH}(4)) + 3.14159$ );
```

VALUE WILL BE COMPUTED
AND PHYSICALLY ASSIGNED
TO I.

VALUE WILL BE COMPUTED
AND PHYSICALLY ASSIGNED
TO S.

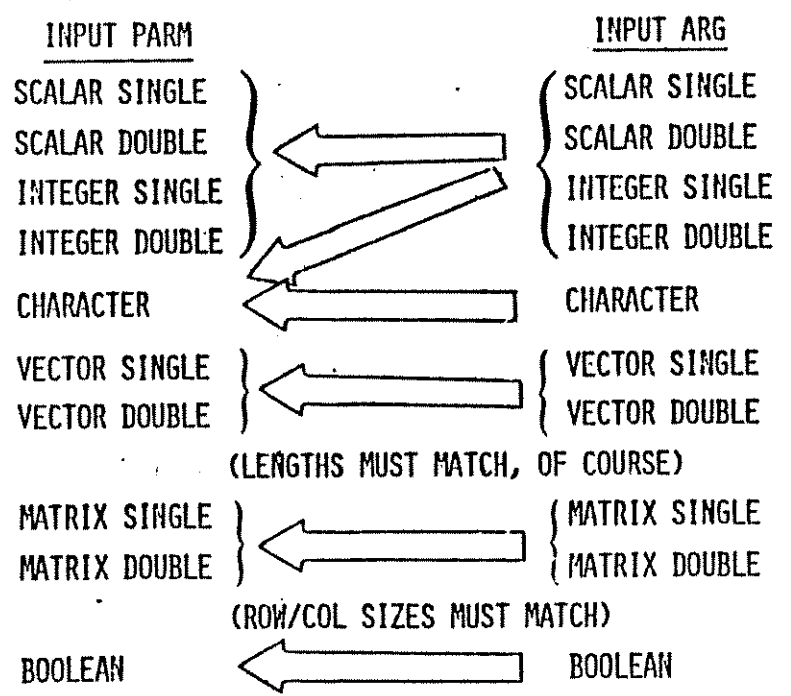
IF INPUT ARGUMENTS ARE EXPRESSIONS THAT RESULT IN NON-INTEGERS OR SCALAR
RESULTS, HAL/S PUTS THE EVALUATED EXPRESSION IN A TEMPORARY LOCATION
AND PASSES THE ADDRESS OF THIS TEMPORARY LOCATION TO THE PROCEDURE.

53
3/4

PROCEDURES AND FUNCTIONS (CON'T.)

IMPLICIT CONVERSIONS (INPUT PARAMETERS)

[NO IMPLICIT CONVERSIONS LEGAL FOR ASSIGN PARAMETERS]



NOTE THAT THE LEGAL CONVERSIONS FOR INPUT ARG  INPUT PARM ARE THE SAME AS FOR ASSIGNMENT STATEMENTS.

PROCEDURES AND FUNCTIONS (CON'T.)

315

ASSIGN PARAMETERS

- ASSIGN ARGUMENTS MUST BE HAL/S DATA ITEMS -- THEY CANNOT BE EXPRESSIONS.
- ASSIGN ARGUMENTS MUST MATCH THE CORRESPONDING PARAMETER IN TYPE AND PRECISION.
- MATRIX/VECTOR DIMENSIONS MUST MATCH EXACTLY.
- IF THE ASSIGN ARGUMENT IS AN ARRAY IT MUST EXACTLY MATCH THE NUMBER OF DIMENSIONS (AND RANGES) OF THE ASSIGN PARAMETER.
- AN ASSIGN ARGUMENT OF VECTOR OR MATRIX TYPE CAN BE COMPONENT SUBSCRIPTED -- PROVIDED THE EFFECT IS TO REDUCE THE VECTOR/MATRIX TO A SINGLE SCALAR.
- AN ARRAY CAN BE SUBSCRIPTED -- BUT ONLY IF ALL ARRAYNESS IS SUBSCRIPTED AWAY, I.E., THE ARRAY IS REDUCED TO A SINGLE ELEMENT.

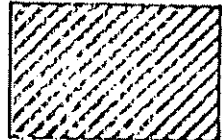
PROCEDURES AND FUNCTIONS (CON'T.)

52
3/6

EXAMPLES

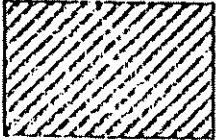
(FROM PROGRAMMER'S
GUIDE)

FUNCTION INTEGER ()
ONE: PROCEDURE



CLOSE;

TWO: PROCEDURE(A,B) ASSIGN(C);
DECLARE A MATRIX(3,3);
DECLARE B INTEGER;
DECLARE C INTEGER;



CLOSE;

DECLARE M1 MATRIX(3,3),
M2 MATRIX(3,3) DOUBLE,
M3 MATRIX(4,4),
S SCALAR,
I INTEGER,
ID INTEGER DOUBLE;

4-47

53
317

PROCEDURES AND FUNCTIONS (CON'T.)

LEGAL:

S = S + ONE;

S = S + M
1, ONE

Note: subscripts may be integer expressions of any kind.

M2 = TWO(M2,S) + M2;

M2 is converted to single precision during transmission.

M2 = TWO(M2,I);

I is converted to scalar type during transmission.

ILLEGAL:

M2 = TWO(M3,1.5);

row and column dimensions of M3 do not match those of parameter A.

M2 = TWO(M1,'ARGUMENT' || I);

transmission of character type argument to scalar parameter B incurs an illegal type conversion.

4-46

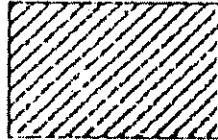
PROCEDURES AND FUNCTIONS (CON'T.)

52
318

EXAMPLES

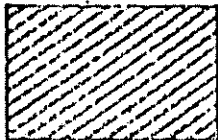
(FROM PROGRAMMER'S
GUIDE)

ONE: FUNCTION INTEGER;



CLOSE;

TWO: FUNCTION(A,B) MATRIX(4,4) DOUBLE;
DECLARE A MATRIX(4,4);
DECLARE B SCALAR;



CLOSE;

DECLARE M1 MATRIX(4,4),
M2 MATRIX(4,4) DOUBLE,
M3 MATRIX(3,3),
S SCALAR,
I INTEGER;

4-4.3

33
319

PROCEDURES AND FUNCTIONS (CON'T.)

CALL ONE;
CALL ONE(I); ← illegal: ONE possesses no parameters.

CALL TWO(M2^T, S+1) ASSIGN(I);
values may be passed in and out of TWO through I.
type conversion required here.
precision conversion required here.

CALL TWO(M3, ID) ASSIGN(S);
type conversion illegal for assign arguments.
precision conversion required.
dimension mismatch: parameter is a 3 x 3 matrix.

CALL TWO(M1, I) ASSIGN(I);
appearance in both places is legal.

4-97

PROCEDURES AND FUNCTIONS (CON'T.)

PROCEDURE QUIRKS

THIS IS ILLEGAL:

```
ALPHA: PROCEDURE(A) ASSIGN(A);  
      DECLARE A;  
      . . .  
      CLOSE ALPHA;
```

BUT THIS IS OK:

```
ALPHA: PROCEDURE(A) ASSIGN(B);  
      DECLARE A, B;  
      . . .  
      CLOSE ALPHA;  
      CALL ALPHA(Q) ASSIGN(Q);
```

IN THIS LATTER CASE, NO HIDDEN DANGERS EXIST IF Q IS AN INTEGER OR SCALAR OR A BIT (BOOLEAN) -- SINCE Q WILL BE PASSED BY VALUE ON THE INPUT SIDE AND ASSIGNED VIA REFERENCE ON THE ASSIGN SIDE. FOR OTHER DATA, HOWEVER, MODIFICATION OF THE ASSIGN PARAMETER CAN RESULT IN MODIFICATION OF THE INPUT PARAMETER!

PROCEDURES AND FUNCTIONS (CON'T.)

THIS PROBLEM WILL NOT OCCUR IF A "TEMPORARY" COPY HAD TO BE MADE FOR THE INPUT ARGUMENT, E.G., BECAUSE A PRECISION CONVERSION WAS NECESSARY.

EXAMPLE

```
ALPHA: PROCEDURE(V) ASSIGN(W);  
       DECLARE VECTOR DOUBLE, V, W;  
        $\bar{W} = 2\bar{V}$ ;  
        $\bar{X} = \bar{V} + \text{VECTOR}(\$(@DOUBLE))(1,1,2)$ ;  
       . . .  
       CLOSE ALPHA;
```

TROUBLE

```
{ DECLARE VECTOR DOUBLE, U;  
  . . .  
  CALL ALPHA( $\bar{U}$ ) ASSIGN( $\bar{U}$ );
```

THIS WOULD NOT BE A PROBLEM IF, WITHIN ALPHA, V WERE DECLARED VECTOR SINGLE INSTEAD!

PROCEDURES AND FUNCTIONS (CON'T.)

5/
322

FUNCTION RETURN

A FUNCTION IS EXITED VIA

RETURN (exp);

WHERE (exp) IS AN EXPRESSION WHICH MATCHES THE DECLARED TYPE OF THE FUNCTION -- EXCEPT THAT THE STANDARD IMPLICIT CONVERSIONS ARE

ALLOWED:

SCALAR	←	INTEGER
INTEGER	←	SCALAR
CHARACTER	←	INTEGER
	←	SCALAR

NOTE THAT PRECISIONS NEED NOT MATCH SINCE HAL/S WILL GENERATE CODE NECESSARY TO ACCOMPLISH PRECISION MATCHING.

NOTE: REMEMBER THAT WE CANNOT SAY

BETA: FUNCTION(I, J) ARRAY(6) SCALAR;

I.E., NO ARRAY DECLARATION IS POSSIBLE. THEREFORE, A FUNCTION CAN NEVER RETURN AN ARRAY!

PROCEDURES AND FUNCTIONS (CON'T.)

52
323

EXAMPLE

BETA: FUNCTION(I, M, V) SCALAR;

DECLARE I INTEGER,

M MATRIX DOUBLE,

V VECTOR;

⋮

RETURN I**2 + 6;

← WILL BE CONVERTED TO A SINGLE
PRECISION SCALAR

⋮

RETURN M\$(3,2);

← WILL BE CONVERTED FROM DOUBLE TO
SINGLE

⋮

RETURN 'I = ' || I;

⋮
← ERROR -- CHARACTER STRINGS DO NOT
CONVERT TO SCALAR IMPLICITLY.

CLOSE BETA;

4-8-2

PROCEDURES AND FUNCTIONS (CON'T.)

31
324

CHARACTER STRINGS AS PARAMETERS:

ALL INPUT AND ASSIGN PARAMETERS OF CHARACTER TYPE
MUST BE DECLARED AS:

CHARACTER(*)

I.E., SPECIFYING AN INDEFINITE LENGTH. THIS AVOIDS
TRUNCATION PROBLEMS DUE TO DYNAMICALLY VARYING SIZES
OF STRINGS.

EXAMPLES

- ① ALPHA: PROCEDURE(C1) ASSIGN(C2),
 DECLARE CHARACTER(*), C1, C2;
 ...
- ② ALPHA: PROCEDURE ASSIGN(C),
 DECLARE C CHARACTER(7), ← ILLEGAL
 ...
- ③ BETA: FUNCTION CHARACTER(8),
- ④ BETA: FUNCTION CHARACTER(*), ← ILLEGAL

NOTE THAT FUNCTIONS OF CHARACTER TYPE MUST SPECIFY THEIR MAXIMUM
LENGTH JUST AS A STANDARD DECLARE HAS TO!

PROCEDURES AND FUNCTIONS (CON'T.)

52
326

ONE DIMENSIONAL ARRAYS ARE MORE FLEXIBLE -- IF THE PARAMETER IS DECLARED AS:

ARRAY(n)

WHERE

$$2 \leq n \leq 32767$$

THEN THE CORRESPONDING ARGUMENT MUST ALSO BE ARRAY(n), I.E., MUST MATCH EXACTLY AS IN THE 2-D AND 3-D CASES. IF:

ARRAY(*)

IS SPECIFIED, HOWEVER, THEN THE ARGUMENTS CAN BE 1-D ARRAYS OF ANY LEGAL LENGTH. THIS ALLOWS PROCEDURES TO OPERATE ON ARRAYS OF VARIABLE SIZE.

EXAMPLES:

① ALPHA: PROCEDURE(A);
 DECLARE A ARRAY(6) INTEGER;
 . . .
 CLOSE ALPHA;

ALPHA CAN ONLY BE CALLED WITH AN ARRAY(6) INPUT ARGUMENT:

DECLARE B ARRAY(6) INTEGER;
 . . .
 CALL ALPHA(B);

PROCEDURES AND FUNCTIONS (CON'T.)

② ALPHA: PROCEDURE(A);
 DECLARE A ARRAY(*) INTEGER;
 . . .
 CLOSE ALPHA;
 . . .

 DECLARE C ARRAY(6) INTEGER,
 D ARRAY(30000) INTEGER DOUBLE,
 E ARRAY(4) SCALAR;

CALL ALPHA(C);
CALL ALPHA(D);
CALL ALPHA(E);

} THESE ARE ALL LEGAL!

WITHIN THE PROCEDURE, THE ACTUAL LENGTH OF THE ARRAY(*)
ITEMS CAN BE FOUND BY MEANS OF THE BUILT-IN FUNCTION
 SIZE(α)

PROCEDURES AND FUNCTIONS

EXAMPLE

```
ALPHA: PROCEDURE(A) ASSIGN(B);  
  DECLARE ARRAY(*) SCALAR, A, B;  
  DECLARE I INTEGER;  
  . . .  
  DO FOR I = 1 TO SIZE(A);  
    B$I = SINH(A$I);  
  END;  
  . . .
```

```
  B = SINH(A);  
  . . .  
CLOSE ALPHA;
```

← AN EXAMPLE OF AN ARRAYED ASSIGNMENT
(WILL BE DISCUSSED LATER)

IMPORTANT POINT: THE COMPILER
CAN ALSO OBTAIN THE SIZES OF ARRAY(*)
AT RUN-TIME.

C. BUILT-IN FUNCTIONS

31
329

HAL/S typically supports the following set of built-in functions. Minor variations may arise between implementations.

ARITHMETIC FUNCTIONS							
<ul style="list-style-type: none"> arguments may be INTEGER or SCALAR types in functions with one argument, result type matches argument type (except as specifically noted) in functions with two arguments, unless specifically specified, result type is scalar if either or both arguments are scalar; otherwise the result type is integer arrayed arguments cause multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match 							
Name, Arguments	Comments						
ABS(α)	$ \alpha $						
CEILING(α)	smallest integer $\geq \alpha$						
DIV(α, β)	integer division α/β (arguments rounded to integers)						
FLOOR(α)	largest integer $\leq \alpha$						
MIDVAL(α, β, γ)	the value of the argument which is algebraically between the other two. If two or more arguments are equal, the multiple value is returned. Result is always scalar.						
MOD(α, β)	$\alpha \text{ MOD } \beta$						
ODD(α)	<table> <tr> <td>TRUE</td> <td>1 if α odd</td> <td rowspan="2">} result is</td> </tr> <tr> <td>FALSE</td> <td>0 if α even</td> <td>BOOLEAN</td> </tr> </table>	TRUE	1 if α odd	} result is	FALSE	0 if α even	BOOLEAN
TRUE	1 if α odd	} result is					
FALSE	0 if α even		BOOLEAN				
REMAINDER(α, β)	signed remainder of integer division α/β (argument rounded to integer)						

140

52
329A

HAL/S-FC COMPILER SYSTEM
SPECIFICATION

IR-95-5

1 March 1976

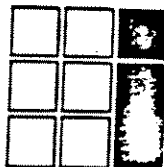
INTERMETRICS APPROVAL

Daniel J. Lickly

Daniel J. Lickly
Head, HAL Language Compiler Department

Dr. F. H. Martin

Dr. F. H. Martin
Shuttle Program Manager



INTERMETRICS

52
329
B

DMDVAL

HAL/S-PC LIBRARY ROUTINE DESCRIPTION

Source Member Name: DMDVAL Size of Code Area: 20 Bw
 Stack Requirement: 18 Bw Data CSECT Size: 0 Bw
 Intrinsic Procedure
 Other Library Modules Referenced: None

ENTRY POINT DESCRIPTIONS

Primary Entry Name: DMDVAL

Function: Finds mid value of three double precision scalar arguments.

Invoked by:
 Compiler emitted code for HAL/S construct of the form:
 MIDVAL(A,B,C) where A,B,C are double precision scalars.
 Other Library Modules:

Execution Time (microseconds): 41.4

Input Arguments:

Type	Precision	How Passed	Units
scalar	DP	F0	
scalar	DP	F2	
scalar	DP	F4	

Output Results:

Type	Precision	How Passed	Units
scalar	DP	F0	

Errors Detected:

Error #	Cause	Fixup

Comments:
 Registers Unsafe Across Call: F0,F1,F2,F3,F4,F5.

```

Algorithm: IF A = B THEN RETURN A;
           IF A < B THEN DO;
             IF B <= C THEN RETURN B;
             ELSE IF A <= C THEN RETURN C;
             ELSE RETURN A;
           END;
           ELSE DO;
             IF C <= B THEN RETURN B;
             ELSE IF C <= A THEN RETURN C;
             ELSE RETURN A;
           END;
  
```

ARITHMETIC FUNCTIONS (CONTINUED)	
Name, Arguments	Comments
ROUND (α)	nearest integral value to α
SIGN (α)	+1 $\alpha > 0$ -1 $\alpha < 0$
SIGNUM (α)	+1 $\alpha > 0$ 0 $\alpha = 0$ -1 $\alpha < 0$
TRUNCATE (α)	largest integer $< \alpha $ times SIGNUM (integer ($\bar{\alpha}$))

Version 11-01-7
 81
 331

ALGEBRAIC FUNCTIONS

- arguments may be integer or scalar types - conversion to scalar occurs with integer arguments
- result type is always scalar
- arrayed arguments cause multiple invocations of the function, one for each array element
- angular values are supplied or delivered in radians.

Name, Arguments	Comments
ARCCOS (α)	$\cos^{-1} \alpha \quad \alpha \leq 1$
ARCCOSH (α)	$\cosh^{-1} \alpha \quad \alpha \geq 1$
ARCSIN (α)	$\sin^{-1} \alpha, \quad \alpha \leq 1$
ARCSINH (α)	$\sinh^{-1} \alpha$
ARCTAN2 (α, β)	$-\pi < \tan^{-1}(\alpha/\beta) \leq \pi$ Proper Quadrant if: $\left. \begin{array}{l} \alpha = k \sin \theta \\ \beta = k \cos \theta \end{array} \right\} k > 0$
ARCTAN (α)	$\tan^{-1} \alpha$
ARCTANH (α)	$\tanh^{-1} \alpha \quad \alpha < 1$
COS (α)	$\cos \alpha$
COSH (α)	$\cosh \alpha$
EXP (α)	e^{α}
LOG (α)	$\log_e \alpha, \quad \alpha > 0$
SIN (α)	$\sin \alpha$
SINH (α)	$\sinh \alpha$
SQRT (α)	$\sqrt{\alpha}, \quad \alpha \geq 0$
TAN (α)	$\tan \alpha$
TANH (α)	$\tanh \alpha$

102

51
332

VECTOR-MATRIX FUNCTIONS

- arguments are vector or matrix types as indicated
- result types are as implied by mathematical operation
- arrayed arguments cause multiple invocations of the function, one for each array element

Name, Arguments	Comments
ABVAL(α)	length of vector α
DET(α)	determinant of square matrix α
INVERSE(α)	inverse of nonsingular square matrix α
TRACE(α)	sum of diagonal elements of square matrix α
TRANSPOSE(α)	transpose of matrix α
UNIT(α)	unit vector in same direction as vector α

51
333

136

MISCELLANEOUS FUNCTIONS		
<ul style="list-style-type: none"> arguments are as indicated; if none are indicated the function has no arguments result type is as indicated 		
Name, Arguments	Result Type	Comments
CLOCKTIME	scalar	returns time of day
DATE	integer	returns date (implementation dependent format)
ERRGRP	integer	returns group number of last error detected, or zero
ERRNUM	integer	returns number of last error detected, or zero
PRI0	integer	returns priority of process calling function
RANDOM	scalar	returns random number from rectangular distribution over range 0-1
RANDOMG	scalar	returns random number from Gaussian distribution mean zero, variance one.
RUNTIME	scalar	returns Real Time Executive clock time (Section 8.)
NEXTIME (<label>)	scalar	<p><label> is the name of a program or task. The value returned is determined as follows:</p> <ol style="list-style-type: none"> If the specified process was scheduled with the REPEAT EVERY option and has begun at least one cycle of execution, then the value is the time the next cycle will begin. If the specified process was scheduled with the IN or AT phrase, and has not yet begun execution, then the value is the time it will begin execution. Otherwise, the value is equal to the current time (RUNTIME function).

116

51-
334

MISCELLANEOUS FUNCTIONS (CONTINUED)		
Name, Arguments	Result Type	Comments
SHL(α, β)	Same as α	<p>α may be integer or bit type. β must be integer type.</p> <p>If α is integer type, the result is an integer whose internal binary representation is that of α shifted left by β bit locations. The signed nature of the integer α is taken into account in an implementation dependent manner which depends upon the number system and word size of the target computer.</p> <p>If α is bit type, the result is a bit string containing the value of α shifted left by β bit locations. α is treated as an unsigned logical quantity. The size of the result is implementation dependent.</p> <p>Arrayed arguments produce multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match.</p>
SHR(α, β)	Same as α	<p>α may be integer or bit type. β must be integer type.</p> <p>Results are as defined for the SHL function except that all shifting occurs to the right.</p> <p>Arrayed arguments produce multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match.</p>

C-5

51
335

CHARACTER FUNCTIONS

- first argument is character type - second argument is as indicated (any argument indicated as character type may also be integer or scalar, whereupon conversion to character type is implicitly assumed)
- result type is as indicated
- arrayed arguments produce multiple invocations of the function, one for each array element - arraynesses of arrayed arguments must match

Name, Arguments	Result Type	Comments
INDEX(α, β)	integer	β is character type - if string β appears in string α , index pointing to the first character of β is returned; otherwise zero is returned
LENGTH(α)	integer	returns length of character string
LJUST(α, β)	character	β is integer type - string α is expanded to length β by padding on the right with blanks $\beta \geq \text{length}(\alpha)$
RJUST(α, β)	character	β is integer type - string α is expanded to length β by padding on the left with blanks $\beta > \text{length}(\alpha)$
TRIM(α)	character	leading and trailing blanks are stripped from α

31
336

BIT FUNCTIONS		
<ul style="list-style-type: none"> arguments are bit type result is bit type arrayed arguments produce multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match 		
Name, Arguments	Result Type	Comments
XOR(α, β)	bit	Result is Exclusive OR of α and β . Length of result is length of longer argument. Shorter argument is left padded with binary zeros to length of longer argument.

ARRAY FUNCTIONS	
<ul style="list-style-type: none"> arguments are n-dimensional arrays where n is arbitrary arguments are integer or scalar type result type matches argument type and is unarrayed 	
Name, Parameters	Comments
MAX(α)	maximum of all elements of α
MIN(α)	minimum of all elements of α
PROD(α)	product of all elements of α
SUM(α)	sum of all elements of α

SI
337

SIZE FUNCTION	
Name, Argument	Comments
SIZE(α)	<p>One of the following must hold:</p> <ul style="list-style-type: none">• α is an unsubscripted arrayed variable with a one-dimensional array specification - function returns length of array.• α is an unsubscripted major structure with a multiple specification - function returns number of classes.• α is an unsubscripted structure terminal with a one-dimensional array specification - function returns length of array. <p>Result is of integer type</p>

E

51
338

BIT STRINGS

REMEMBER THAT BOOLEAN = BIT(1)

Ⓐ BIT STRING LITERALS

BIN(n)'bbbb...b' OR BIN'bbbb...b'
OCT(n)'oooo...o' OR OCT'oooo...o'
HEX(n)'hhhh...h' OR HEX'hhhh...h'
DEC(n)'dddd...d' OR DEC'dddd...d'

- THE REPETITION COUNT (n) IS OPTIONAL
- THE RESULTING BIT LITERAL MUST HAVE BETWEEN 1 AND 32 BITS.

EXAMPLES

	<u>LENGTH</u>
BIN'1011' = 1011	4
BIN(3)'101' = 101101101	9
HEX(2)'FFE' = 1111111111011111111110	24
OCT'714' = 111001100	9
DEC'25' = 11001	5

S/
339

BIT STRINGS (CON'T.)

(B) BIT DECLARATION

DECLARE *name* BIT(*n*);

WHERE $1 \leq n \leq 32$

AGAIN, NOTE THAT:

DECLARE B BIT(1);

AND

DECLARE B BOOLEAN;

ARE EQUIVALENT. BIT DECLARATIONS CAN BE COMBINED WITH OTHER
DECLARED DATA IN COMPOUND AND FACTORED DECLARES, E.G.,

DECLARE BIT(4), B1, B2, B3;

DECLARE I INTEGER, B1 BOOLEAN,

B2 BIT(32), S SCALAR DOUBLE,

B3 BIT(6);

5 2

5/
340

BIT STRINGS (CON'T.)

Ⓒ BIT STRING INITIALIZATION

BIT STRINGS ARE INITIALIZED VIA AN INITIAL/CONSTANT LIST
CONTAINING BIT STRING LITERALS:

```
DECLARE B1 BIT(16) INITIAL(HEX'33F'),  
        B2 BIT(1) CONSTANT(FALSE),  
        B3 BIT(8) CONSTANT(BIN'11100111'),  
        B4 ARRAY(3) BIT(2) INITIAL(BIN'10', BIN'00', BIN'11');
```

LITERALS ARE PADDED ON THE LEFT WITH ZEROS OR TRUNCATED ON THE LEFT AS REQUIRED.

```
· DECLARE B1 BIT(8) INITIAL(BIN'1'),  
        B2 BIT(4) INITIAL(HEX'FF');
```

RESULTS IN:

```
B1 = 00000001  
B2 = 1111          (4 BITS LOST)
```

BIT STRINGS (CON'T.)

S/341

Ⓓ BIT STRING SUBSCRIPTING

- COMPONENT SUBSCRIPTING -

(UNARRAYED BITS)

- TO SELECT THE ITH BIT FROM A BIT STRING (RESULT IS A BIT(I)):

BIT_STRING\$I

WHERE

$$1 \leq I \leq L$$

(L ≡ LENGTH OF BIT_STRING)

'I' CAN BE VARIABLE, I.E., AN EXPRESSION.

- TO SELECT I BITS FROM A BIT STRING STARTING AT THE JTH (RESULT IS A BIT(I)):

BIT_STRING\$(I AT J)

WHERE

$$1 \leq I \leq L \quad \text{AND IS KNOWN AT COMPILE TIME}$$

AND $1 \leq J \leq L - I + 1$ AND MAY BE VARIABLE (AN EXPRESSION).

32
342

BIT STRINGS (CON'T.)

- TO SELECT A SUBSTRING STARTING WITH THE ITH BIT AND ENDING WITH THE JTH;

BIT_STRING\$(I TO J)

WHERE

I AND J ARE KNOWN AT COMPILE TIME, AND

$$1 \leq I < J \leq L$$

NOTE (1): THE ESSENCE OF THE RULES FOR SUBSCRIPTING IS THAT THE RESULTANT BIT STRING MUST HAVE A LENGTH KNOWN AT COMPILE TIME (AND OF COURSE LIE WITHIN 1 AND 32).

NOTE (2): SINCE BOOLEANS ARE EQUIVALENT TO BIT(1), THEY MAY BE COMPONENT SUBSCRIPTED -- ALTHOUGH THIS IS SENSELESS. THIS EXPLAINS, HOWEVER, THE FACT THAT ARRAY SUBSCRIPTS ON BOOLEAN ARRAYS MUST HAVE A TRAILING COLON:

BOOL_TABLE\$(1:)

BIT STRINGS (CON'T.)

51
343

- ARRAYED BIT STRINGS -

BIT STRINGS (AND BOOLEANS) MAY, OF COURSE, BE ORGANIZED INTO ARRAYS OF 1, 2, OR 3 DIMENSIONS.

ASSUME:

B IS AN ARRAY(4) OF BIT(5) STRINGS:

$B \equiv (10110_2, 00000_2, 01100_2, 11111_2)$

THEN

$B\$(2:) \equiv 00000_2$

A BIT(5)

$B\$(2:3) \equiv 0_2$

A BIT(1)

$B\$(1:2 \text{ AT } 3) \equiv 11_2$

A BIT(2)

$B\$(2 \text{ AT } 2: 3 \text{ AT } 3) \equiv$

$(000_2, 100_2)$

ARRAY(2) BIT(3)

$B\$(*:1) \equiv (1_2, 0_2, 0_2, 1_2)$

ARRAY(4) BIT(1)

5-1

51
344

BIT STRINGS (CON'T.)

Ⓔ BIT STRING OPERATIONS

LEGAL OPERATIONS: (SAME AS BOOLEANS WITH THE ADDITION OF CATENATION).

& AND	}	INTERSECTION
 OR	}	CONJUNCTION
¬ NOT	}	COMPLEMENT
 CAT	}	CATENATION

COMPLEMENT - INVERTS THE LOGICAL VALUE OF EVERY BIT IN THE STRING.

IF B = 101101011

THEN ¬B = 010010100

BIT STRINGS (CON'T.)

S2
345

CONJUNCTION -- OR's ALL CORRESPONDING BITS TOGETHER
(BINARY OPERATION).

IF B1 \equiv 11101011

B2 \equiv 10011100

THEN B1/B2 \equiv 11111111

NOTE: IF THE TWO BIT STRINGS ARE OF UNEQUAL LENGTH, THE
SHORTER IS PADDED ON THE LEFT WITH ZEROS BEFORE ORING.
THE RESULTANT BIT STRING HAS THE LENGTH OF THE LONGER
STRING.

IF B1 \equiv 101

B2 \equiv 111000

B1/B2 \equiv 111101

(A BIT(6))

5-4

52
346

BIT STRINGS (CON'T.)

INTERSECTION - AND'S ALL CORRESPONDING BITS TOGETHER
(BINARY OPERATION).

IF B1 = 1101001
B2 = 0110010
THEN B1 AND B2 = 0100000

NOTE: IF THE TWO BIT STRINGS ARE OF UNEQUAL LENGTH, THE
SHORTER IS PADDED ON THE LEFT WITH ZEROS BEFORE ANDING.
THE RESULTANT BIT STRING HAS THE LENGTH OF THE LONGER
STRING.

IF B1 = 101
B2 = 110110
THEN B1 AND B2 = 000100 (BIT(6))

BIT STRINGS (CON'T.)

52
347

CATENATION - TWO BIT STRINGS (BOOLEANS) CAN BE CHAINED
(CATENATED) TOGETHER TO FORM A SINGLE (LONGER)
BIT STRING. THE SECOND STRING IS APPENDED TO THE
END OF THE FIRST. IF THE RESULTANT BIT STRING
EXCEEDS 32, THEN THE LEFTMOST EXCESS BITS ARE
TRUNCATED.

IF B1 \equiv TRUE \equiv BIN'1'

B2 \equiv 10110

THEN B1 CAT B2 \equiv 110110 BIT(6)

IF B1 \equiv 11011

B2 \equiv 10101

THEN B1||B2 \equiv 1101110101 BIT(10)

IF B1 \equiv 10101010101010101 BIT(17)

B2 \equiv 110110110110110110 BIT(18)

B1||B2 \equiv 1010101010101010110110110110110110

THROWN
AWAY



B1||B2 \equiv 01010101010101110110110110110110

A BIT(32)

SI
348

BIT STRINGS (CON'T.)

Ⓕ PRECEDENCE

HI				
↑	¬, NOT	1	COMPLEMENT	
	, CAT	2	CATENATION	
	&, AND	3	INTERSECTION	
↓	, OR	4	CONJUNCTION	
LO				

SEQUENCES OF OPERATIONS OF THE SAME PRECEDENCE ARE
EVALUATED LEFT TO RIGHT.

EXAMPLE:

IF (B1||¬B2|B3 & B4|B5|B6)=ON
 ↑ ↑ ↑ ↑ ↑ ↑
 ② ① ④ ③ ⑥ ⑤

I.E., EQUIVALENT TO:

IF ((B1||¬B2)|(B3&B4)| (B5|B6))=ON
(WHICH IS EASIER TO READ)

BIT STRINGS (CON'T.)

SI.
349

⑥ BIT STRING ASSIGNMENTS

L_STRING = R_STRING

- NEITHER, EITHER, OR BOTH MAY BE SUBSCRIPTED.
- IF L_STRING > R_STRING IN LENGTH, THEN R_STRING IS LEFT PADDED WITH ZEROES PRIOR TO ASSIGNMENT
- IF L_STRING < R_STRING IN LENGTH, THEN R_STRING IS TRUNCATED FROM THE LEFT AS NEEDED.

EXAMPLES:

LET B1 BE A BIT(8)

B2 BE A BIT(6) = 110110

THEN B1 = B2 RESULTS IN

B1 = 00110110

IF B3 IS A BIT(10) = 1010111101

THEN B1 = B3 RESULTS IN

B1 = 10111101

5-1

52
350

BIT STRINGS (CON'T.)

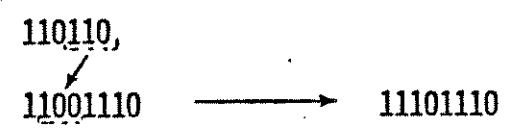
IF B1 = 11001110 BIT(8)

B2 = 110110 BIT(6)

THEN

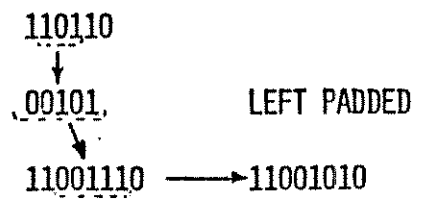
A) B1\$(3 AT 2) = B2 RESULTS
IN B1 = 11101110

I.E.,



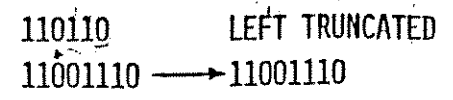
B) B1\$(5 AT 3) = B2\$(2 TO 4) RESULTS
IN B1 = 11001010

I.E.,



C) B1\$3 = B2 RESULTS
IN B1 = 11001110

I.E.,



BIT STRINGS (CON'T.)

51
351

Ⓜ BIT STRINGS IN CONDITIONAL STATEMENTS

- IF BIT STRINGS ARE COMPONENT SUBSCRIPTED DOWN TO A SINGLE BIT, THEN THEY MAY BE MIXED WITH BOOLEANS TO FORM BOOLEAN EXPRESSIONS.

EXAMPLES:

B1 BIT(4) ≡ 1101

B2 BIT(8) ≡ 10110100

B3 BOOLEAN ≡ TRUE

B4 BOOLEAN ≡ FALSE

IF B1 THEN X = 0; ILLEGAL -- B1 IS NOT A BOOLEAN.

IF B1\$1 THEN X = 0;
TRUE WILL BE EXECUTED

IF (B1&B2\$2) THEN X = 0; ILLEGAL -- B1 IS NOT A BOOLEAN.

IF (B1\$2&B2\$3&B3|B4) THEN X = 0;
TRUE FALSE WILL BE EXECUTED

S2
352

BIT STRINGS (CON'T.)

- BIT STRINGS MAY BE COMPARED FOR = OR \neg = (CLASS II OPERATORS) TO FORM RELATIONAL EXPRESSIONS.

E.G.,

DO WHILE B1 = B2;
IF B2 = BIN'101' THEN DO;

IF THE BIT OPERANDS ARE OF UNEQUAL LENGTH THEN, OF COURSE, THE SHORTER IS LEFT-PADDED WITH THE REQUISITE NUMBER OF ZEROES PRIOR TO THE COMPARISON.

EXAMPLES:

B1 = 101

B2 = 11

B3 = 1100101

B4 = 00101

THEN B1 = B4 IS TRUE

B1 = B2 IS FALSE

B1 \neg B2 IS TRUE

B3 = B4 IS FALSE

B3\$(3 TO 7) = B4 IS TRUE

(B2||B4) = B3 IS TRUE

BIT STRING COMPARISONS

TRUE \equiv ON \equiv 0000 ... 0001

FALSE \equiv OFF \equiv 0000 ... 0000

33
353

SUPPOSE:

DECLARE B1 BIT(8) INITIAL(BIN'11011101');

DECLARE B2 BIT(4) INITIAL(BIN'1010');

- (A) IF B1 THEN DO; (ILLEGAL)
- (B) IF B1 = TRUE THEN DO;
FALSE BECAUSE '11011101' \neq '00000001'
- (C) IF B2 = FALSE THEN DO;
FALSE BECAUSE '1010' \neq '0000'
- (D) IF B1\$8 THEN DO;
TRUE
- (E) IF B2\$(2 AT 2) = TRUE THEN DO;
TRUE BECAUSE '01' = '01'
- (F) IF B1\$(3 AT 4) = BIN'111' THEN DO;
TRUE

5-15,1

51
354

BIT STRINGS (CON'T.)

SUMMARY

A CONDITIONAL EXPRESSION, E.G.

IF (cond exp) THEN ...

DO WHILE (cond exp);

DO UNTIL (cond exp);

IS MADE UP OF COMBINED RELATIONAL EXPRESSIONS, E.G.

$(A < 0) \& (B = \text{BIN}'101')$ IS < 2

OR COMBINED BOOLEAN EXPRESSIONS, E.G.

$(B1 | (B2 \& \neg B3)) \& (B2 | B3)$

WHERE B1, ... B3 ARE BOOLEANS

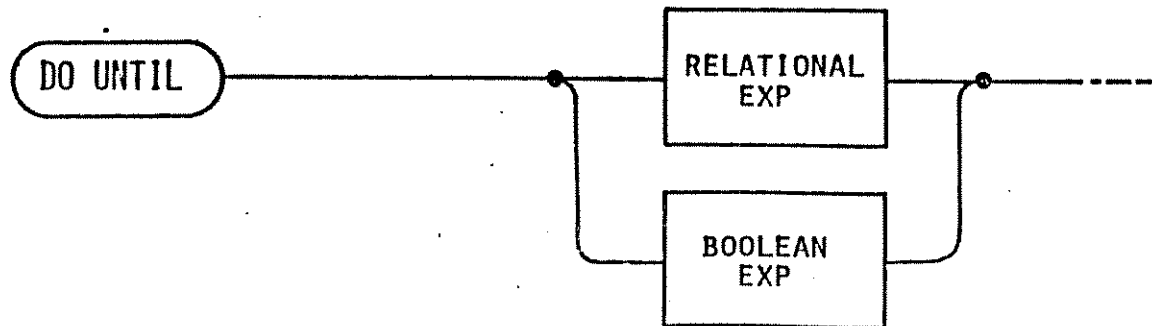
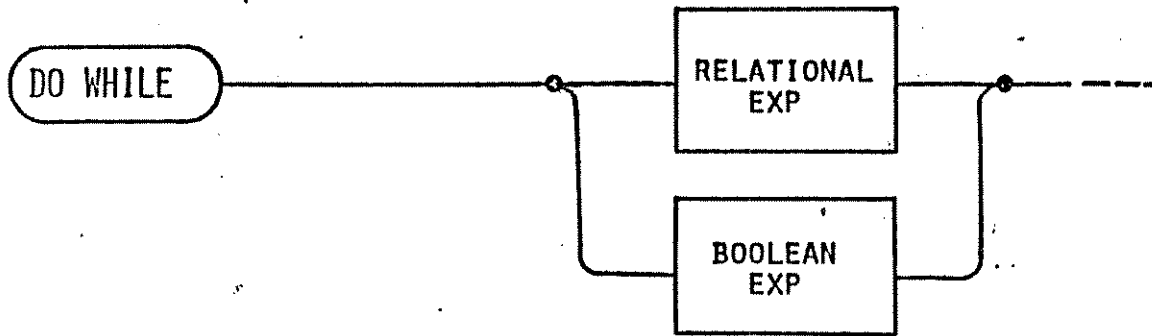
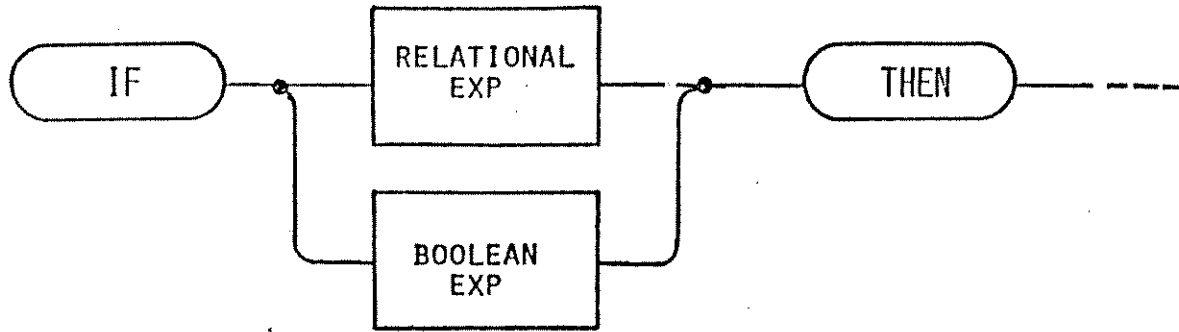
NOTE: A BIT STRING THAT IS SUBSCRIPTED DOWN TO 1
BIT CAN BE USED AS A BOOLEAN.

BUT, A CONDITIONAL EXPRESSION CANNOT BE A MIXTURE:

IF (B1 & B2) | (M = N) ...
IS ILLEGAL!!

RELATIONAL EXPRESSION
BIT EXPRESSION

CLARIFICATION



CONDITIONAL EXPRESSIONS (CON'T.)

CONDITIONAL EXPRESSION IS EITHER A RELATIONAL EXPRESSION OR A BOOLEAN EXPRESSION:

RELATIONAL: $(A < 3) \mid (\overset{*}{M} = \overset{*}{N}) \ \& \ (S = T^{**2})$

BOOLEAN: $B1 \ \& \ (\neg B2 \mid B3) \ \& \ B4 \mid B5$

CONDITIONAL EXPRESSIONS (CON'T.)

52
357

KEY POINT: A BIT(1) IS EXACTLY EQUIVALENT TO A BOOLEAN AND
MAY BE USED IN A BOOLEAN EXPRESSION,

EXAMPLE 1: (MIXING BIT(1) AND BOOLEAN)

```
DECLARE BA BIT(1), BB BOOLEAN,  
        BC BOOLEAN;  
IF BA1(¬BB&BC) THEN ...
```

EXAMPLE 2: (MAKING A BIT(1) BY SUBSCRIPTING)

```
DECLARE QBITS BIT(8), BB BOOLEAN,  
        BC BOOLEAN;  
IF QBITS$3|(BB&BC) THEN ...
```

EXAMPLE 3: (ILLEGAL USAGES)

```
IF QBITS THEN ... (THIS WILL NOT WORK EVEN THOUGH THE  
PROGRAMMER'S GUIDE IMPLIES THAT IT WILL  
EVALUATE TO TRUE IF THE LOWEST BIT IS A  
1!!)
```

EXAMPLE 4: (FIXUP OF EXAMPLE 3)

```
IF QBITS = TRUE ... (NOW IT IS A RELATIONAL EXP)
```

5-16.3

CONDITIONAL EXPRESSIONS (CONT'D.)

```
1 M↑ BOOL:
1 M↑ PROGRAM;
2 M↑   DECLARE BOOLEAN INITIAL(FALSE),
2 M↑     BA, BB, BC, BD, BE, BF;
3 M↑   DECLARE INITIAL(BIN'0'),
3 M↑     B1 BIT(1),
3 M↑     B2 BIT(2),
3 M↑     B3 BIT(3),
3 M↑     B4 BIT(4),
3 M↑     B5 BIT(5),
3 M↑     B6 BIT(6),
3 M↑     B7 BIT(7),
3 M↑     B8 BIT(8),
3 M↑     B9 BIT(9),
3 M↑     B10 BIT(10),
3 M↑     B11 BIT(11),
3 M↑     B12 BIT(12),
3 M↑     B13 BIT(13),
3 M↑     B14 BIT(14),
3 M↑     B15 BIT(15),
3 M↑     B16 BIT(16),
3 M↑     B17 BIT(17);
4 M↑   DECLARE QQ BIT(5) INITIAL(BIN'111110101010101010101');
   C↑
   E↑
5 M↑   IF BA & BB ↑ BC & (\BD ↑ BE) THEN
6 M↑     DO;
7 M↑     END;
```

5-16.4

CONDITIONAL EXPRESSIONS (CONT'D.)

```

      E↑
  8 M↑   IF (B1 = B2) ↑ (B3 \= B4) & (\B5 = B6) THEN
  9 M↑       DO;
 10 M↑       END;
      C↑
      E↑
 11 M↑   IF (BA = BB) ↑ (BC = BD) & (BA \= BE) THEN
 12 M↑       DO;
 13 M↑       END;
      C↑
      E↑
 14 M↑   IF (BA \= BB) ↑ (BC \= BD) & (B6 \= B8) THEN
 15 M↑       DO;
 16 M↑       END;
      C↑
      E↑
 17 M↑   IF BA THEN
 18 M↑       DO;
 19 M↑       END;
      C↑
      E↑
 20 M↑   IF B4 THEN
**** OB1   ERROR #1 OF SEVERITY 1. *****
**** BIT EXPRESSION IN IF CLAUSE MUST BE BOOLEAN
 21 M↑       DO;
 22 M↑       END;

```

5-16.5

```

C↑
E↑
23 M↑ IF B4 THEN
S↑ 1
24 M↑ DO;
HAL/S FC- 9. 21
TMT INTERMETRICS, INC.
SOURCE
25 M↑ END;
C↑
E↑
26 M↑ IF (B6 ↑ B7 ) THEN
S↑ 2 3
27 M↑ DO;
28 M↑ END;
C↑
E↑
29 M↑ IF BA ↑ B1 & B2 ↑ BB THEN
**** OB1 ERROR #2 OF SEVERITY 1. ****
**** BIT EXPRESSION IN IF CLAUSE MUST BE BOOLEAN
**** LAST ERROR WAS DETECTED AT STATEMENT 20. ****
30 M↑ DO;
31 M↑ END;
C↑
E↑
32 M↑ IF (BA = TRUE) ↑ (BB = FALSE) ↑ (B4 = BIN'1011') THEN
33 M↑ DO;
34 M↑ END;
C↑
E↑
35 M↑ IF B5 = TRUE THEN
S↑ 2 AT 2
36 M↑ DO;
37 M↑ END;
C↑
E↑
38 M↑ IF B10 THEN
S↑ 4 TO 9
**** OB1 ERROR #3 OF SEVERITY 1. ****
**** BIT EXPRESSION IN IF CLAUSE MUST BE BOOLEAN
**** LAST ERROR WAS DETECTED AT STATEMENT 29. ****

```

5-16.6

BIT STRINGS (CON'T.)

51
358

① BIT STRING ARGUMENTS & PARAMETERS

- BIT STRINGS MAY BE INPUT PARAMETERS (ARGUMENTS) OF PROCEDURES AND FUNCTIONS AND ASSIGN PARAMETERS (ARGUMENTS) OF PROCEDURES.

EXAMPLE:

```
FLAGS: PROCEDURE(B1) ASSIGN(B2),  
      DECLARE B1 BIT(16),  
            B2 BIT(8),  
      . . .  
      CLOSE FLAGS;
```

RULES (INPUT PARAMETERS)

- BOTH PARAMETER AND ARGUMENT MUST BE OF BIT TYPE (NO IMPLICIT CONVERSIONS).
- THE ARGUMENT IS PADDED/TRUNCATED ON THE LEFT AS NECESSARY TO FIT THE INPUT PARAMETER.

5-17

BIT STRING (CON'T.)

RULES: (ASSIGN PARAMETERS)

- THE ASSIGN ARGUMENT MUST BE A DECLARED BIT DATA ITEM.
- ARGUMENT AND PARAMETER LENGTHS MUST MATCH EXACTLY!
- NO SUBSCRIPTING (COMPONENT) IS ALLOWED ON THE ARGUMENT.

52
357

Examples:

Let the following data be declared:

```
| DECLARE B1 BIT(16),  
|           B2 BIT(3);
```

and let the following procedure be defined:

```
| SWITCHES: PROCEDURE(D2) ASSIGN(D1);  
|           DECLARE D1 BIT(3),  
|           D2 BIT(8);
```

• • •

```
| CLOSE SWITCHES;
```

Both legal and illegal invocations of this procedure are shown below:

```
| CALL SWITCHES (B1 | BIN'1001') ASSIGN(B2);
```

this 16-bit quantity truncated to 9 bits on passage

```
| CALL SWITCHES (B2) ASSIGN(B1);
```

illegal - length mismatch

this 3-bit quantity padded to 8 bits on passage

```
| CALL SWITCHES (BIN'1') ASSIGN(FALSE);
```

illegal - not a declared bit string data item.

5-18

BIT STRINGS (CON'T.)

82
360

① BIT STRING FUNCTIONS

Label: FUNCTION(l_1, l_2, \dots) BIT(n);

OPTIONAL INPUT
PARAMETERS

$1 \leq n \leq 32$

AN INVOCATION OF SUCH A BIT FUNCTION BEHAVES LIKE A BIT(n) DATA ITEM IN ALL CONTEXTS!

EXAMPLES:

① F: FUNCTION(B) BIT(3);

DECLARE B BIT(8);

...
RETURN B; ← LEFTMOST 5 BITS TRUNCATED

...
RETURN B\$4; ← RESULT PADDED (ON LEFT) TO 3 BITS

...
RETURN 6; ← ILLEGAL! BIT STRING QUANTITY NOT BEING RETURNED.

...
CLOSE F;

STRUCTURES

51
362

HAL/S ALLOWS TWO TYPES OF DATA ORGANIZATIONS: ARRAYS & STRUCTURES.

- ARRAYS: 1-, 2-, OR 3-DIMENSIONAL PATTERNS OF HOMOGENEOUS DATA ITEMS.

EXAMPLE:

```
DECLARE AX ARRAY(3,2,100) INTEGER INITIAL(5);
```

EACH ARRAY DIMENSION CAN RANGE FROM 2 TO 32,767.

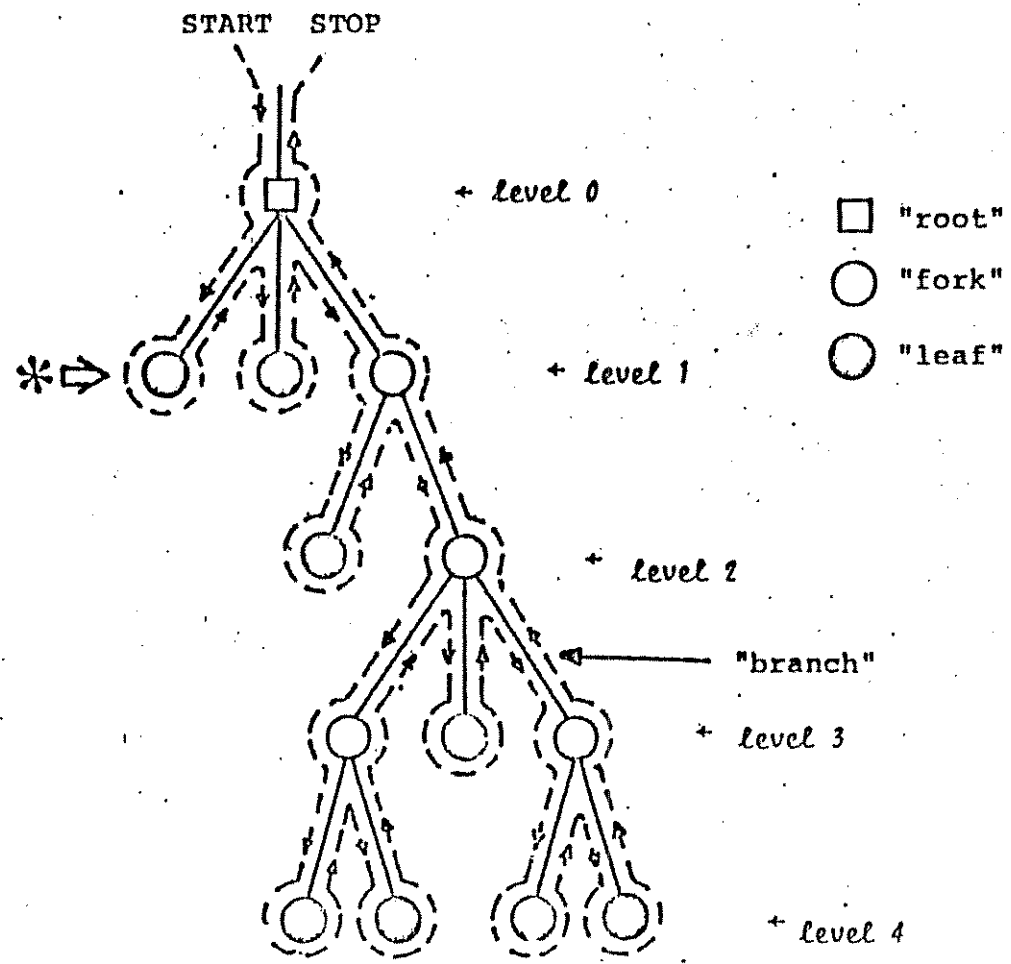
- STRUCTURES: HIERARCHICAL (TREE-LIKE) COLLECTIONS OF HOMOGENEOUS OR HETEROGENEOUS DATA ITEMS. A STRUCTURE IS MORE GENERAL THAN AN ARRAY SINCE IT CAN CONTAIN ARRAYS (AND EVEN OTHER STRUCTURES).

A STRUCTURE MUST REFERENCE A STRUCTURE TEMPLATE (THINK OF IT AS A DSECT IN 360 TERMINOLOGY) THAT WAS PREVIOUSLY DEFINED.

51
363

STRUCTURES (CON'T.)

A STRUCTURE TEMPLATE IS A TREE:



5-22

STRUCTURES (CON'T.)

53
364

NOTE: THE PROGRAMMER'S GUIDE MAY BE CONSULTED FOR AN ABSTRACT DESCRIPTION OF STRUCTURE TEMPLATES. HERE WE WILL DEVELOP THE CONCEPT USING CONCRETE EXAMPLES.

① STRUCTURE TEMPLATE:

Defn STRUCTURE *name*: *node*¹, *node*², *node*³, ... *node*ⁿ ;

TEMPLATE NAME

WHERE:

node^t ≡ *n name*

(MINOR STRUCTURES OR FORKS)

node^t ≡ *n name attributes*

(STRUCTURE TERMINAL)

NOTE: THE STRUCTURE LEVEL 'n' MUST SATISFY $1 \leq n \leq 5$

CURRENT LIMITATION

STRUCTURES (CON'T.)

52
365

EXAMPLE 1

STRUCTURE QQ: ← NOTE THE COLON!

- 1 A INTEGER,
- 1 B BOOLEAN,
- 1 M MATRIX(4,4) DOUBLE,
- 1 C BIT(6);

← NOTE THE SEMI-COLON!

4 STRUCTURE TERMINALS ALL AT LEVEL 1.

EXAMPLE 2

STRUCTURE RR:

- 1 Z1, ← A MINOR STRUCTURE HAS NO ATTRIBUTES
- 2 I INTEGER, }
- 2 J INTEGER, } ← EACH MINOR STRUCTURE HAS 2 TERMINALS
- 1 Z2,
- 2 I INTEGER, }
- 2 J INTEGER; }

2 MINOR STRUCTURES
OR FORKS

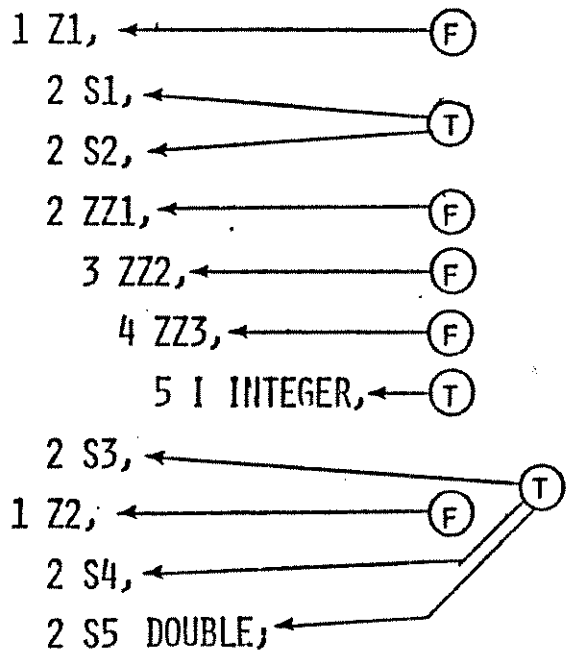
5.2

STRUCTURES (CON'T.)

S2
366

EXAMPLE 3

STRUCTURE SS:



(T) TERMINAL

(F) FORK (MINOR STRUCTURE)

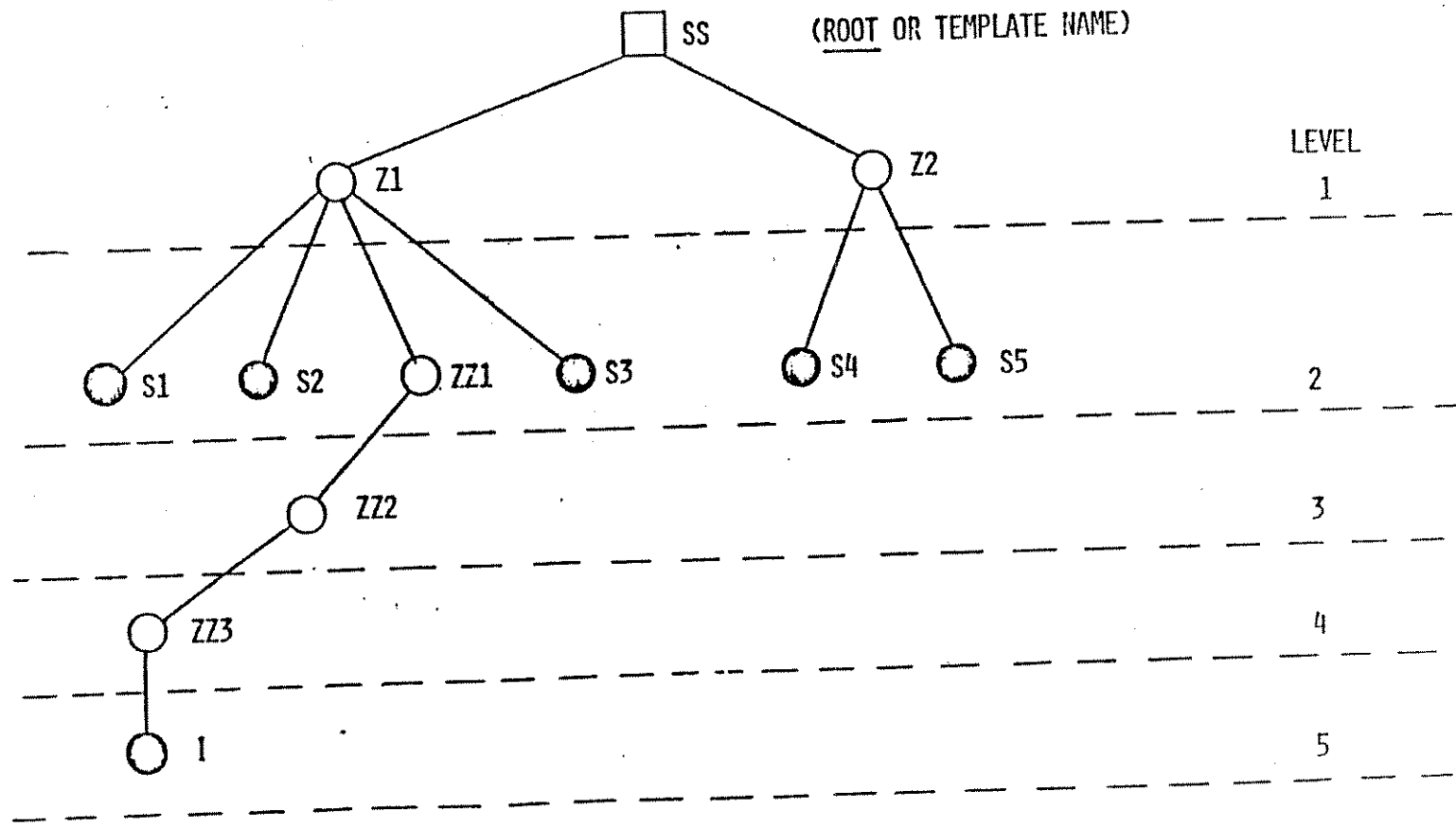
NOTE (1): FORK (MINOR STRUCTURE) IS EASILY SEEN BY OBSERVING THAT THE LEVEL # OF THE NEXT NODE GOES UP BY 1!

S-2.1-

53
367

STRUCTURES (CON'T.)

AS A TREE, EXAMPLE 3 TAKES THE FORM:



○ ≡ FORK (MINOR STRUCTURE) ● ≡ LEAF (TERMINAL)

526

STRUCTURES (CON'T.)

S3
368

NOTE (2): IF A TERMINAL HAS NO ATTRIBUTES IT IS CONSIDERED TO BE A SCALAR (JUST LIKE DECLARES).

NOTE (3): IF AN INTEGER, SCALAR, VECTOR, OR MATRIX TERMINAL HAS NO PRECISION SPECIFICATION, IT IS ASSUMED TO BE SINGLE (JUST LIKE DECLARES).

THE FOLLOWING DATA TYPES ARE LEGAL IN A STRUCTURE TEMPLATE:

INTEGER	BOOLEAN
SCALAR	BIT
VECTOR	CHARACTER
MATRIX	STRUCTURE

ALSO, NAME VARIABLES OF THESE TYPES ARE LEGAL, PLUS:

NAME EVENT
NAME PROGRAM
NAME TASK

WILL BE DISCUSSED
LATER.

S.27

31
369

STRUCTURES (CON'T.)

RESTRICTIONS:

- (1) TEMPLATE NAMES MUST BE UNIQUE WITHIN THE NAME SCOPE.
- (2) NAMES WITHIN THE TEMPLATE DO NOT NECESSARILY HAVE TO BE UNIQUE.
- (3) INITIAL/CONSTANT LISTS CANNOT BE PRESENT IN A TEMPLATE.
- (4) STATIC/AUTOMATIC CANNOT BE SPECIFIED IN A TEMPLATE.
- (5) CHARACTER(*), ARRAY(*), AND STRUCTURE(*) ARE ALSO PRECLUDED.
- (6) A STRUCTURE TERMINAL CANNOT HAVE COPIES.

WILL BE DISCUSSED
LATER

STRUCTURE TEMPLATES ARE ESSENTIALLY DECLARE STATEMENTS -- AS A RESULT THEY MUST BE PLACED WITHIN A DECLARE GROUP (AND PRIOR, OF COURSE, TO ANY STRUCTURE DECLARATIONS REQUIRING THE TEMPLATE).

STRUCTURES (CON'T.)

31
370

(B) STRUCTURE DECLARATION:

SIMPLE STRUCTURE

DECLARE *name* α -STRUCTURE;

MULTI-COPY STRUCTURE

DECLARE *name* α -STRUCTURE(N);

WHERE *name* IS THE STRUCTURE (OR MAJOR STRUCTURE) NAME AND α IS THE NAME OF A PREVIOUSLY-DEFINED TEMPLATE. N IS THE NUMBER OF COPIES DESIRED.

$$2 \leq N \leq 32,767$$

EXAMPLE

STRUCTURE Q:

1 QA SCALAR, OR JUST 1 QA

1 QB INTEGER,

1 QC BOOLEAN; HYPHEN

DECLARE Q Q-STRUCTURE(3);

DECLARE R Q-STRUCTURE;

DECLARE I INTEGER, S SCALAR, W Q-STRUCTURE;

5.29

STRUCTURES (CON'T.)

3/
371

EXAMPLE

```
CPL: EXTERNAL COMPOOL,  
    . . .  
STRUCTURE Q:  
    1 A,  
    1 B DOUBLE,  
    1 C INTEGER,  
    . . .  
CLOSE CPL,  
PROG: PROGRAM,  
    . . .  
STRUCTURE R:  
    1 I INTEGER DOUBLE,  
    1 V ARRAY(6,5) VECTOR(4),  
    1 D CHARACTER(80),  
    . . .  
DECLARE R R-STRUCTURE,  
DECLARE S Q-STRUCTURE(10),  
DECLARE T T-STRUCTURE,  
STRUCTURE T:  
    1 E INTEGER,
```

ILLEGAL - TEMPLATE MUST PRECEDE
STRUCTURE

STRUCTURES (CON'T.)

SI
372

© STRUCTURE INITIALIZATION

INITIAL/CONSTANT LISTS ARE ATTACHED TO THE STRUCTURE
DECLARATION -- NOT TO A TEMPLATE OR ITS INNARDS.

TERMINALS ARE INITIALIZED BY THE ORDER OF APPEARANCE
IN THE TEMPLATE.

- STRUCTURES WITH NO COPIES -

STRUCTURE Q:

1 A,
2 I INTEGER,
2 S SCALAR,
1 B,
2 J INTEGER,
2 K INTEGER,
2 L INTEGER,
2 M ARRAY(10) INTEGER;

DECLARE Z Q-STRUCTURE INITIAL(4, 6.95, 3#2, *)

PARTIAL INITIALIZATION

I ≡ 4 S ≡ 6.95 J ≡ K ≡ L ≡ 2

M IS UNINITIALIZED

SI

STRUCTURES (CON'T.)

5/
373

- STRUCTURES WITH COPIES -

TWO CHOICES:

- PUT ENOUGH LITERALS IN THE INITIAL/CONSTANT LIST TO INITIALIZE THE FIRST COPY -- ALL REMAINING COPIES WILL BE INITIALIZED LIKE THE FIRST AUTOMATICALLY.
- PUT ENOUGH LITERALS IN THE INITIAL/CONSTANT FOR ALL TERMINALS OF ALL COPIES. EACH COPY IS FULLY INITIALIZED BEFORE PASSING TO THE NEXT.

EXAMPLES

STRUCTURE Q:
1 I INTEGER,
1 S SCALAR,
, , ,

- (1) DECLARE Q Q-STRUCTURE(5) INITIAL(6, 81.45);
- (2) DECLARE R Q-STRUCTURE(3) INITIAL(1, 84.0, 2, -16.4, 4, 350.8);
- (3) DECLARE SS Q-STRUCTURE(20000) INITIAL(5, -40.6, 3, 119.2, *)

COPY 1

COPY 2

COPIES 3-20,000
UNINITIALIZED

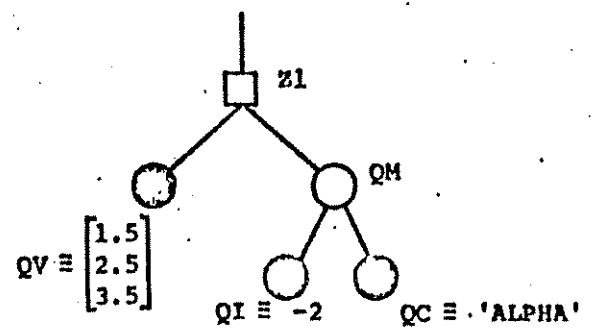
5-31

52
374

STRUCTURES (CON'T.)

INITIALIZATION EXAMPLE

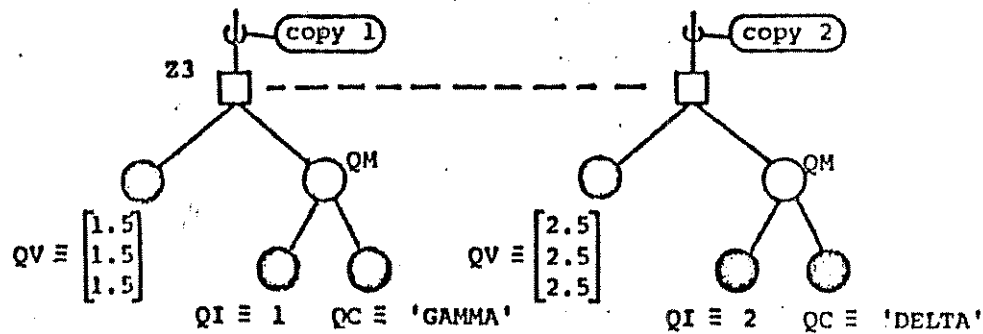
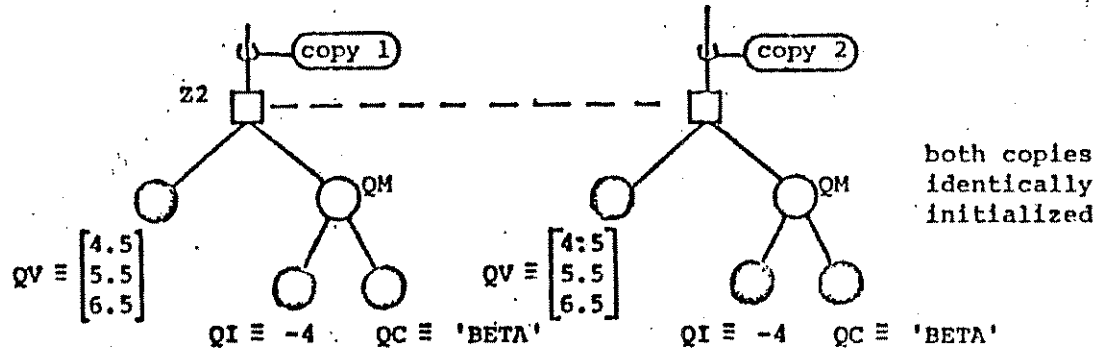
```
STRUCTURE Q;  
  1 QV VECTOR(3),  
  1 QM,  
  2 QI INTEGER,  
  2 QC CHARACTER(80);  
  
DECLARE Z1 Q-STRUCTURE INITIAL(1.5, 2.5, 3.5, -2, 'ALPHA');  
DECLARE Z2 Q-STRUCTURE(2) INITIAL(4.5, 5.5, 6.5, -4, 'BETA');  
DECLARE Z3 Q-STRUCTURE(2) INITIAL(3#1.5, 1, 'GAMMA',  
  3#2.5, 2, 'DELTA');
```



S-33

STRUCTURES (CON'T.)

53
375



5.34

S/
376

STRUCTURES (CON'T.)

(E) QUALIFIED REFERENCES

STRUCTURE Q:

1 A,
2 I INTEGER,
2 B,
3 J INTEGER,
3 S1 SCALAR,
3 C,
4 M MATRIX(4,4),
4 V VECTOR(4),
3 S2 SCALAR,
2 D,
3 CHARX CHARACTER(10),
3 BB BOOLEAN,
2 K INTEGER,
1 MM X-STRUCTURE,
1 VV VECTOR(3),

DECLARE R Q-STRUCTURE INITIAL(...);

(1) ENTIRE STRUCTURE IS REFERENCED VIA NAME 'R', I.E., THE MAJOR STRUCTURE NAME.

(2) A TERMINAL IS REFERENCED BY: $R.n_1, n_2, \dots, n_x, \text{terminal_name}$

MINOR STRUCTURES ENCOUNTERED

STRUCTURES (CON'T.)

52
377

I R.A.I
J R.A.B.J
S1 R.A.B.S1
M R.A.B.C.M
V R.A.B.C.V
S2 R.A.B.S2
CHARX. R.A.D.CHARX
BB R.A.D.BB
K R.A.K
MM R.MM
VV R.VV

R REFERENCES THE WHOLE STRUCTURE

R.A
R.A.B
R.A.B.C
R.A.D } REFERENCE SUB-STRUCTURES (MINOR STRUCTURES) OF
VARYING COMPLEXITY

STRUCTURES (CON'T.)

SI
378

① STRUCTURE NESTING:

```
1 M↑ PROG:
1 M↑ PROGRAM;
2 M↑   STRUCTURE A:
2 M↑     1 AI INTEGER,
2 M↑     1 A1,
2 M↑     2 AC CHARACTER(80),
2 M↑     2 AB BOOLEAN;
3 M↑   STRUCTURE B:
3 M↑     1 BS SCALAR,
3 M↑     1 B1,
3 M↑     2 BV VECTOR(3),
3 M↑     2 BA A-STRUCTURE; ← CANNOT HAVE COPIES!
4 M↑   STRUCTURE C:
4 M↑     1 BS SCALAR,
4 M↑     1 B1,
4 M↑     2 BV VECTOR(3),
4 M↑     2 BA,
4 M↑     3 AI INTEGER,
4 M↑     3 A1,
4 M↑     4 AC CHARACTER(80),
4 M↑     4 AB BOOLEAN;
5 M↑   DECLARE Q A-STRUCTURE;
6 M↑   DECLARE R B-STRUCTURE;
7 M↑   DECLARE S C-STRUCTURE;
8 M↑   DECLARE T C-STRUCTURE;
9 M↑   DECLARE U A-STRUCTURE;
C↑
```

5-35

STRUCTURES (CON'T.)

32
379

```

E↑      +      +
10 M↑   R = S;
**** AV4.      ERROR #1 OF SEVERITY 1. *****
**** TREE ORGANIZATIONS DO NOT MATCH ACROSS ASSIGNMENT
C↑
E↑      +      +
11 M↑   S = T;
C↑
E↑      +      +
12 M↑   R. B1. BA = Q;
C↑
E↑      +      +
13 M↑   S. B1. BA = U;
14 M↑   CLOSE;
```

STRUCTURES (CON'T.)

8/
380

THE QUALIFIED REFERENCE SYSTEM MUST BE USED FOR QUALIFIED STRUCTURES,
BUT IS OPTIONAL FOR UNQUALIFIED STRUCTURES.

DEFN. A STRUCTURE IS UNQUALIFIED IF IT HAS THE SAME NAME AS ITS
TEMPLATE -- OTHERWISE, IT IS A QUALIFIED STRUCTURE.

EXAMPLE

```
STRUCTURE Q:
  1 I INTEGER,
  1 S SCALAR,
  1 M MATRIX DOUBLE,
  1 T;
DECLARE Q Q-STRUCTURE;
DECLARE R Q-STRUCTURE;
THEN Q IS UNQUALIFIED.
THEN R IS QUALIFIED.
```

STRUCTURES (CON'T.)

52
381

IF A STRUCTURE IS UNQUALIFIED, THEN RATHER THAN USE THE QUALIFIED REFERENCE MECHANISM, TERMINALS AND MINOR STRUCTURES CAN SIMPLY BE REFERRED TO BY THEIR NAMES.

IN THE PRECEDING EXAMPLE:

Q		SHORT FORM (SINCE Q IS <u>UNQUALIFIED</u>)
I	I OR Q.I	
S	S OR Q.S	
M	M OR Q.M	
T	T OR Q.T	
R		
I	R.I	} MUST USE QUALIFICATIONS.
S	R.S	
M	R.M	
T	R.T	

UNQUALIFIED STRUCTURES ARE NICE ... BUT NOT EVERY STRUCTURE CAN BE UNQUALIFIED.

NOTE: IN TERMS OF CPU OR CORE, THERE IS NO DIFFERENCE BETWEEN A QUALIFIED OR AN UNQUALIFIED STRUCTURE.

STRUCTURES (CON'T.)

51
382

RULES FOR UNQUALIFIED STRUCTURES:

- (1) BOTH THE TEMPLATE AND THE STRUCTURE MUST BELONG TO THE SAME NAME-SCOPE:

LEGAL
P: PROGRAM;
STRUCTURE Q;
 1 A,
 1 B;
 . . .
 DECLARE Q Q-STRUCTURE;
 . . .
CLOSE P;

ILLEGAL
CPL: COMPOOL;
STRUCTURE Q;
 1 A,
 1 B;
 . . .
 CLOSE CPL;
 . . .
P: PROGRAM;
 DECLARE Q Q-STRUCTURE;

TEMPLATE IS IN
DIFFERENT NAME-
SCOPE

- (2) THE TEMPLATE CAN CONTAIN NO NESTED STRUCTURES, I.E., STRUCTURE TERMINALS,

FOR EXAMPLE, THE FOLLOWING IS ILLEGAL:

STRUCTURE Q:
 1 A,
 1 I INTEGER,
 1 M R-STRUCTURE, ← CANNOT HAVE NESTED STRUCTURE
 DECLARE Q Q-STRUCTURE(4);

STRUCTURES (CON'T.)

52
383

(3) ALL NAMES (IDENTIFIERS) WITHIN THE TEMPLATE MUST BE UNIQUE
IN THE NAME-SCOPE.

KEY IDEA: IDENTIFIERS WITHIN A TEMPLATE ARE NORMALLY INVISIBLE TO
THE OUTSIDE ENVIRONMENT. BY MAKING THE STRUCTURE UNQUALIFIED,
THESE HIDDEN NAMES SUDDENLY BECOME VISIBLE.

EXAMPLE

```
DECLARE I;  
STRUCTURE Q:  
  1 I INTEGER,  
  1 S,  
  1 M MATRIX;
```

SO FAR ... NO PROBLEM (THE TWO I'S CANNOT "SEE" EACH OTHER)

```
  DECLARE R Q-STRUCTURE(10);  
STILL NO PROBLEM...
```

```
  DECLARE Q Q-STRUCTURE;  
NOW WE WILL HAVE A MULTIPLY-DEFINED SYMBOL "I".
```


STRUCTURES (CON'T.)

53
384

IF THE STRUCTURE TEMPLATE IS NEVER INTENDED TO BE UNQUALIFIED,
THEN DUPLICATION OF IDENTIFIERS CAN OCCUR WITHIN THE TEMPLATE
AS LONG AS NO AMBIGUITY RESULTS WHEN USING THE QUALIFIED REFERENCE
SYSTEM.

EXAMPLE 1

STRUCTURE Q:

1 Q1,

2 QS,

1 Q2,

2 QS,

LEGAL DUPLICATE NAMES

DECLARE ZQ Q-STRUCTURE;

BECAUSE ZQ.Q1.QS CANNOT BE CONFUSED WITH ZQ.Q2.QS

NOTE THAT:

DECLARE Q Q-STRUCTURE

WOULD RESULT IN MULTIPLY-DEFINED QS.

C. 57

STRUCTURES (CON'T.)

51
385

EXAMPLE 2

STRUCTURE R:

```
1 R1, ← ILLEGAL DUPLICATE NAMES
2 RS SCALAR,
1 R1 CHARACTER(80);
DECLARE ZR R-STRUCTURE;
```

WHAT IS ZR.R1?

SUMMARY OF NAME UNIQUENESS

- (1) WITHIN THE SAME NAME-SCOPE, ALL TEMPLATE AND STRUCTURE NAMES MUST BE UNIQUE (UNLESS, OF COURSE, WE ARE DEFINING AN UNQUALIFIED STRUCTURE -- IN WHICH CASE A TEMPLATE NAME AND A MAJOR STRUCTURE NAME WILL MATCH)
- (2) CLEARLY, FOR A GIVEN TEMPLATE, THERE CAN BE AT MOST ONE UNQUALIFIED STRUCTURE WITH THAT NAME.
- (3) IF A TEMPLATE IS UNQUALIFIED, ALL IDENTIFIERS WITHIN IT MUST BE UNIQUE IN THE NAME-SCOPE.

STRUCTURES (CON'T.)

52
386

- (4) IF A TEMPLATE IS QUALIFIED, SOME DUPLICATION OF IDENTIFIERS CAN OCCUR IF NO AMBIGUITY RESULTS.

EXAMPLES

(1) DECLARE S;
STRUCTURE S:
1 ..,
MULTIPLE DEFN. OF S.

(2) STRUCTURE S:
1 A,
1 B;
STRUCTURE S:
1 A,
1 B;
MULTIPLE DEFN. OF S

(3) STRUCTURE S:
1 A,
1 B;
OK
DECLARE S S-STRUCTURE;

(4) DECLARE M MATRIX;
DECLARE M S-STRUCTURE;
MULTIPLE DEFN. OF M

6-1

STRUCTURES (CON'T.)

53
387

(5) STRUCTURE S:

1 A,
2 I,
2 J,
1 B,
2 I,
2 J;

OK AS LONG AS WE DO NOT SAY
DECLARE S S-STRUCTURE

(6) DECLARE I INTEGER;

STRUCTURE S:

1 I INTEGER,
1 J INTEGER;

↓
DITTO

MISCELLANEOUS COMMENTS

- (1) A TEMPLATE DECLARATION DOES NOT SET ASIDE CORE MEMORY -- ONLY THE STRUCTURE DECLARATION DOES THAT.
- (2) QUALIFIED AND UNQUALIFIED STRUCTURES ARE EQUIVALENT IN TERMS OF CORE AND CPU.
- (3) THERE IS NO INHERENT INEFFICIENCY IN COLLECTING RELATED DATA INTO A STRUCTURE.

STRUCTURES (CON'T.)

(4) IN TERMS OF CORE AND CPU EFFICIENCY WE CAN WRITE:

—————→ LESS EFFICIENT

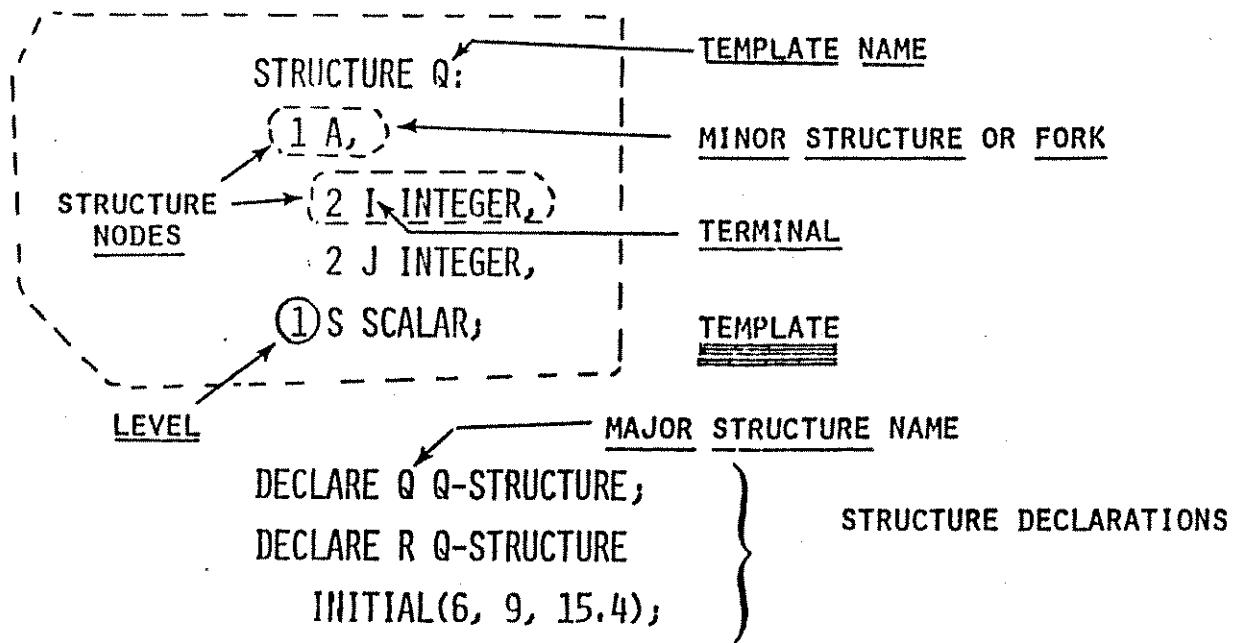
SIMPLE VARIABLE \geq ARRAY \geq STRUCTURE

THIS MEANS THAT A SIMPLE VARIABLE GENERALLY MIGHT BE MORE EFFICIENT THAN AN ARRAY, AND AN ARRAY IN TURN MIGHT BE MORE EFFICIENT THAN A STRUCTURE -- IN MANY IMPORTANT CASES, HOWEVER, EQUALITY HOLDS (MORE ON THIS LATER...).

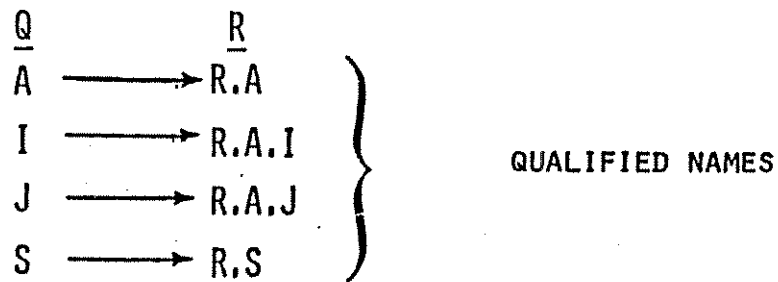
STRUCTURES (CON'T.)

SI
389

SUMMARY OF TERMINOLOGY



Q ≡ UNQUALIFIED STRUCTURE
R ≡ QUALIFIED STRUCTURE



STRUCTURES (CON'T.)

S/
390

A. STRUCTURE SUBSCRIPTING

• COPY SUBSCRIPTING ONLY

SUPPOSE WE HAVE A MULTI-COPY STRUCTURE:

DECLARE Q Q-STRUCTURE(L);

(1) TO SELECT THE ITH COPY

Q\$I OR Q\$(I);

WHERE I IS AN INTEGER EXPRESSION (SCALAR IF YOU WANT) THAT HAD BETTER LIE IN THE RANGE

$$1 \leq I \leq L$$

AT RUN-TIME.

NOTE: SEMICOLON IS ONLY MANDATORY WHEN TERMINAL
SUBSCRIPTING IS TO BE DONE ALSO.

STRUCTURES (CON'T.)

S2
391

- (2) TO SELECT A SUBSET OF I COPIES STARTING WITH THE JTH;

Q\$(I AT J) OR Q\$(I AT J_j)

WHERE I IS AN INTEGER LITERAL (I.E., VALUE KNOWN AT
COMPILE-TIME) AND J IS AN INTEGER EXPRESSION WITH:

$$1 \leq J \leq L - I + 1$$

- (3) TO SELECT A SUBSET OF COPIES STARTING FROM THE ITH AND
ENDING WITH THE JTH;

Q\$(I TO J) OR Q\$(I TO J_j)

WHERE BOTH I AND J ARE INTEGER LITERALS AND:

$$1 \leq I \leq J \leq L$$

STRUCTURES (CON'T.)

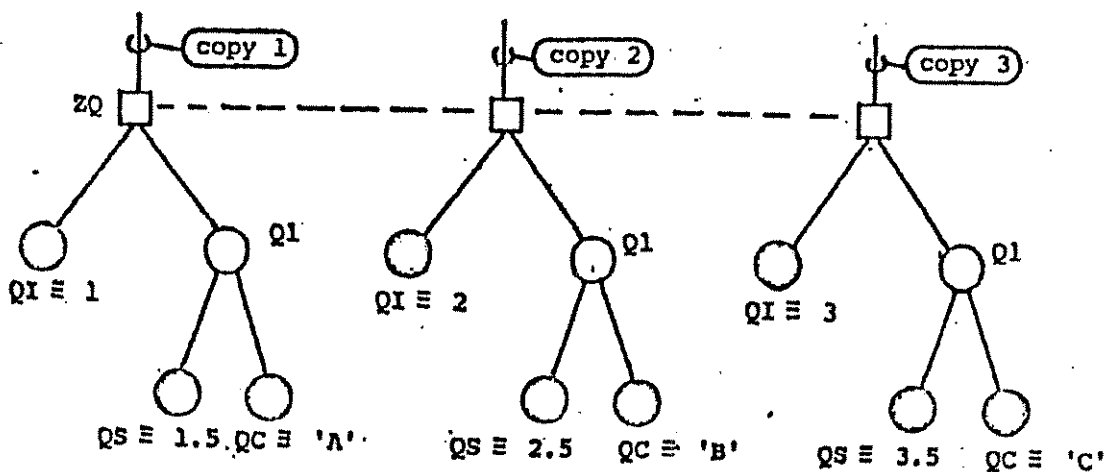
52
392

COPY SUBSCRIPTING EXAMPLES:

GIVEN

```
STRUCTURE Q:  
  1 QI INTEGER,  
  1 Q1,  
  2 QS SCALAR,  
  2 QC CHARACTER(80);  
  ⋮  
DECLARE ZQ Q-STRUCTURE(3);
```

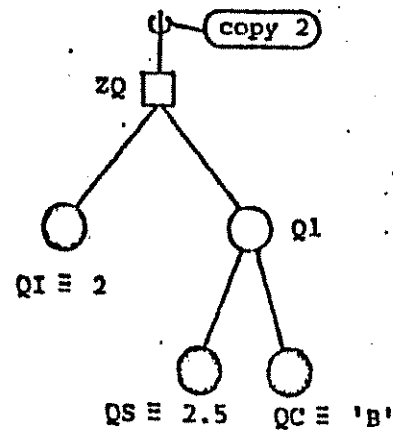
WITH THE FOLLOWING VALUES:



53
393

STRUCTURES (CON'T.)

THEN ZQ_2 , SELECTS COPY 2 WITH VALUES:



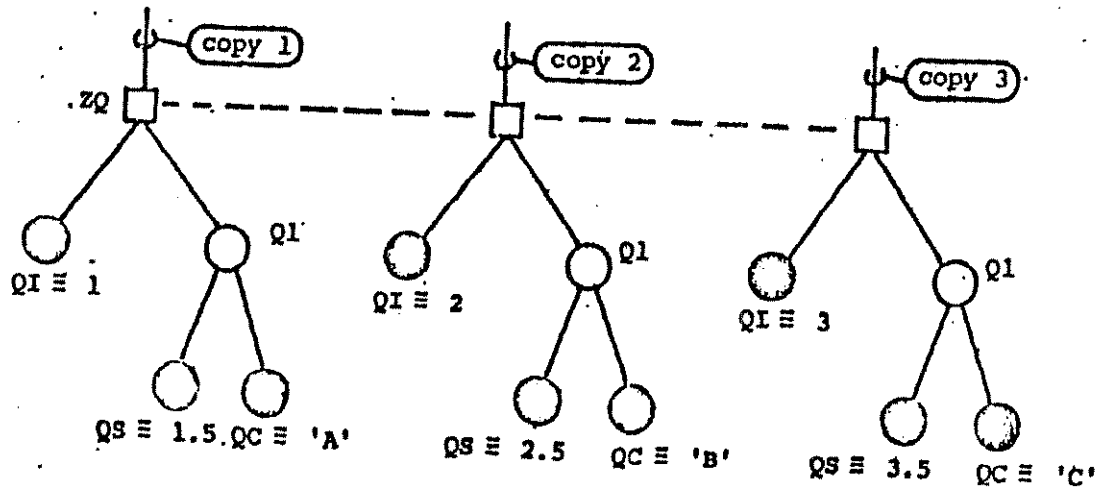
C₂-16.

STRUCTURES (CON'T.)

GIVEN

```
STRUCTURE Q:  
  1 QI INTEGER,  
  1 QI,  
  2 QS SCALAR,  
  2 QC CHARACTER(80);  
:  
: DECLARE ZQ Q-STRUCTURE(3);
```

WITH THE FOLLOWING VALUES:



STRUCTURES (CON'T.)

GIVEN

STRUCTURE Q:

1 QI INTEGER,

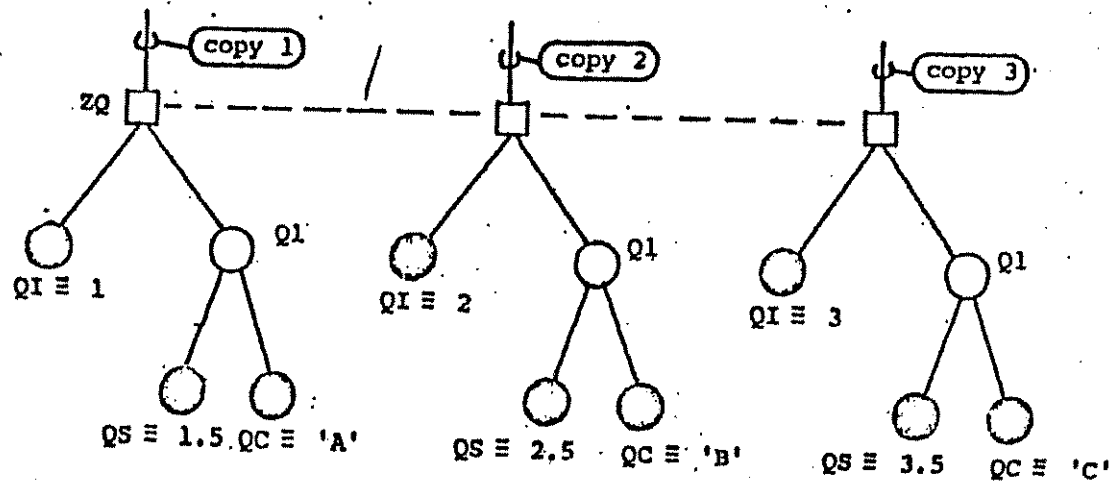
1 Q1,

2 QS SCALAR,

2 QC CHARACTER(80);

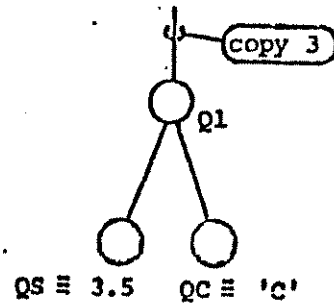
DECLARE ZQ Q-STRUCTURE(3);

WITH THE FOLLOWING VALUES:



STRUCTURES (CON'T.)

QZ.Q1₃ SELECTS:



STRUCTURES (CON'T.)

51
398

• SUBSCRIBING OF STRUCTURE TERMINALS

- (1) IF THE STRUCTURE HAS NO COPIES THEN THE TERMINAL IS SUBSCRIBED EXACTLY AS THOUGH IT WERE NOT IN A STRUCTURE.

STRUCTURE Q:

1 B1 BIT(8),
1 M MATRIX(4,4),
1 C CHARACTER(8),
1 V VECTOR;

DECLARE Q Q-STRUCTURE, ← NO COPIES

THEN:

B1\$4	OR	Q.B1\$4
B1\$(3 AT 3)		Q.B1\$(3 AT 3)
M\$(2,4)		Q.M\$(2,4)
M\$(*,3)		Q.M\$(*,3)
C\$5		Q.C\$5
C\$(4 TO 6)		Q.C\$(4 TO 6)
V\$3		Q.V\$3
V\$(2 AT 1)		Q.V\$(2 AT 1)

STRUCTURES (CON'T.)

52
379

STRUCTURE Q:
1 I ARRAY(10) INTEGER,
1 B ARRAY(6) BOOLEAN,
1 M ARRAY(2,4,6) MATRIX,
1 B4 ARRAY(8) BIT(4);
DECLARE R Q-STRUCTURE;

THEN:

R.I\$5 R.I\$(2 AT 3) ETC.
R.B\$(3:) R.B\$(3 TO 5:) ETC.
R.M\$(*,*, *: 2,3)
R.M\$(1,3,5: *,1)
R.M\$(2,*,6:)
R.M\$(*,*,3: 1 TO 2, 2 AT 1)
R.B4\$(5:)
R.B4\$(5:3)
R.B4\$(2 TO 5: 3 AT 1)
. . .

THEREFORE, THERE IS NOTHING NEW TO SAY ABOUT SUBSCRIBING
IF THE STRUCTURE HAS NO COPIES.

3/
400

(2) IF THE STRUCTURE HAS COPIES THEN WE HAVE THREE MODES OF TERMINAL SUBSCRIPTING:

(A) STRUCTURE (COPY) SUBSCRIPTING ONLY
⇒ NEED TRAILING ";" UNLESS TERMINAL IS AN UNARRAYED INTEGER OR SCALAR.

EXAMPLE

```
STRUCTURE Q:
  1 I INTEGER,
  1 II ARRAY(5) INTEGER,
  1 B BOOLEAN,
  1 M MATRIX;
DECLARE Q Q-STRUCTURE(50);
```

THEN:
I\$5 GETS I FROM 5TH COPY
II\$(5,) GETS II ARRAY FROM 5TH COPY
II\$(10 AT 4,) BEHAVES LIKE A 2-DIMENSIONAL ARRAY

SEMI-COLON NEEDED
BECAUSE BOOLEAN ≡ BIT(1) B\$(3,) GETS BOOLEAN FROM 3RD COPY
 M\$(6,) GETS MATRIX FROM 6TH COPY

(...)

STRUCTURES (CON'T.)

52
401

ⓑ STRUCTURE AND TERMINAL SUBSCRIPTING

⇒ COPY SUBSCRIPT; TERMINAL SUBSCRIPTS

EXAMPLE

STRUCTURE Q:

1 II ARRAY(2,3) INTEGER,
1 BB ARRAY(6) BIT(9),
1 MM ARRAY(5,10,15) MATRIX(4,5),
1 N ARRAY(10) INTEGER,
1 I INTEGER;

DECLARE R Q-STRUCTURE(20000);

THEN:

R.II\$(4;2,2)
R.II\$(100 AT 307; 2,*)
R.BB\$(60; 4: 5 TO 8)
R.BB\$(31;5:)
R.MM\$(5 AT 9; 5,*,7: 3,*)
R.MM\$(27; *,*,*: *,5)
R.N\$(6;10)
R.I\$(50)

STRUCTURES (CON'T.)

53
402

© TERMINAL SUBSCRIPTING ONLY



*; TERMINAL SUBSCRIPTS

EXAMPLE

STRUCTURE Q:

1 II ARRAY(10) INTEGER,
1 M ARRAY(10,5,6) MATRIX;

STRUCTURE QQ:

1 Z1 Q-STRUCTURE,
1 J ARRAY(5) BIT(16),
1 A,
2 BOOL ARRAY(10) BOOLEAN,
2 C ARRAY(5) CHARACTER(80);

DECLARE P QQ-STRUCTURE(20);

THEN:

P.Z1.II\$(*; 6 TO 10)

P.Z1.M\$(*; 1 TO 3, 2 TO 5, 4: *,3)

P.Z1.M\$(*; *,*,*; 2,3)

P.J\$(*; 4 AT 1: 3 AT 1)

P.A.BOOL\$(*; 6:)

P.A.C\$(*; 3 TO 5: 10 AT 24)

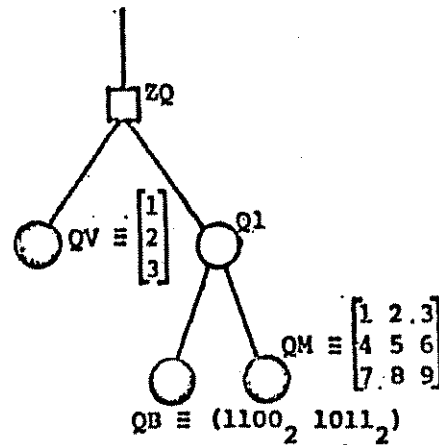
STRUCTURES (CON'T.)

EXAMPLES OF TERMINAL SUBSCRIPTING

GIVEN

```
| STRUCTURE Q:  
| 1 QV VECTOR(3),  
| 1 Q1,  
| 2 QB ARRAY(2) BIT(4),  
| 2 QM MATRIX(3,3);  
|  
|  
|  
| DECLARE ZQ Q-STRUCTURE;
```

WITH THE FOLLOWING VALUES:



STRUCTURES (CON'T.)

THEN:

$$ZQ, QV_1 \equiv 1$$

$$ZQ, Q1, QB_{*3} \equiv (O_2 \ 1_2)$$

AND $ZQ, Q1, QM_{2 \text{ TO } 3, 2 \text{ TO } 3} \equiv \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix}$

STRUCTURES (CON'T.)

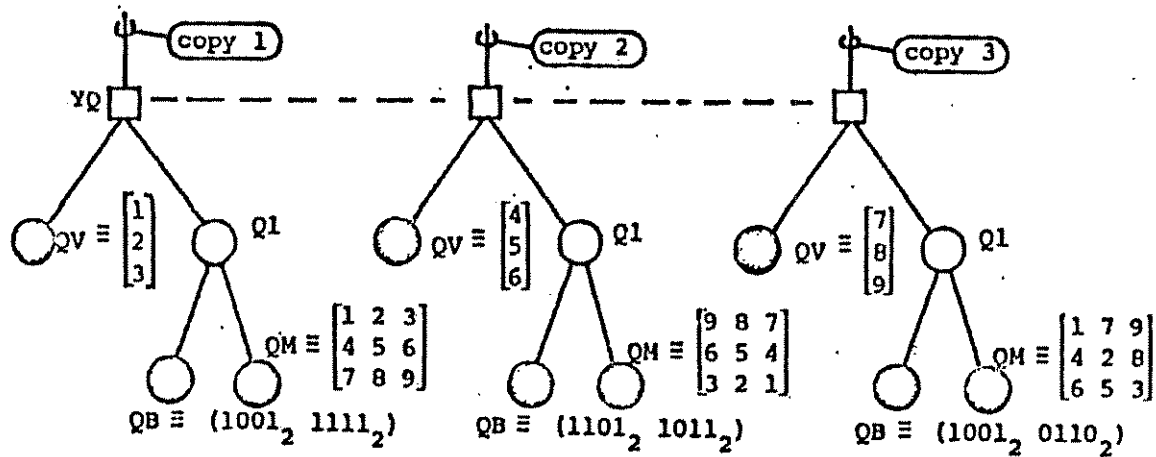
FURTHER, GIVEN

STRUCTURE Q:

1 QV VECTOR(3),
 1 Q1,
 2 QB ARRAY(2) BIT(4),
 2 QM MATRIX(3,3)

⋮
 DECLARE YQ Q-STRUCTURE(3);

WITH THE FOLLOWING VALUES:



STRUCTURES (CON'T.)

then

$$YQ.QV_{*13} \equiv (\overset{\text{copy 1}}{3} \quad \overset{\text{copy 3}}{6} \quad \underset{\text{copy 2}}{9}) \quad \text{result is scalar type}$$

$$YQ.Q1.QB_{2,*:1 \text{ TO } 2} \equiv (11_2 \ 10_2)$$

└ array property unmodified

$$YQ.Q1.QM_{2,3,3} \equiv 1$$

STRUCTURES (CON'T.)

51
407

TREE (TEMPLATE) EQUIVALENCE

TWO STRUCTURES (OR MINOR STRUCTURES) ARE EQUIVALENT (TREE-EQUIVALENT) IF:

- (1) THE ACTUAL "SHAPES" OF THE TREES (ORGANIZATION OF NODES) ARE THE SAME, AND
- (2) CORRESPONDING NODES AGREE EXACTLY IN ATTRIBUTES.

Ⓐ OBVIOUSLY, TWO STRUCTURES ARE EQUIVALENT IF THEY USE THE SAME TEMPLATE:

```
|  STRUCTURE Q:  
|  1 Q1 INTEGER,  
|  1 Q1,  
|  2 QS SCALAR,  
|  2 QC CHARACTER(80);  
|  DECLARE ZQ1 Q-STRUCTURE,  
|          ZQ2 Q-STRUCTURE(20);
```

ZQ1 AND ZQ2 ARE TREE-EQUIVALENT, (NOTWITHSTANDING THE MISMATCH IN NUMBER OF COPIES).

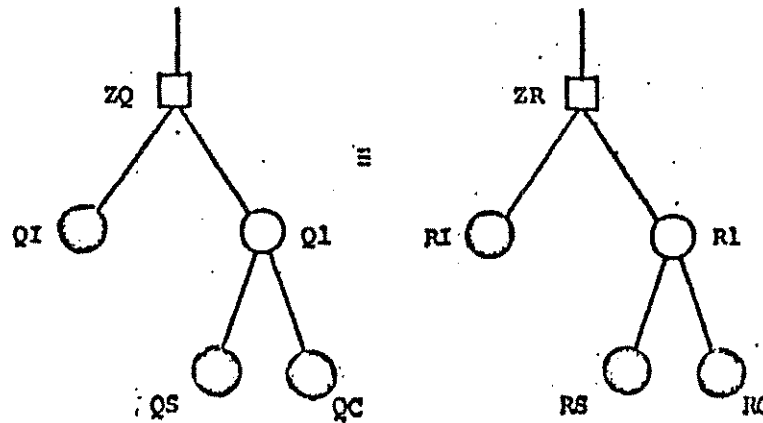
STRUCTURES (CON'T.)

52
408

- Ⓑ STRUCTURES ARE ALSO EQUIVALENT IF THEIR TEMPLATES DIFFER ONLY IN THE IDENTIFIERS:

```
STRUCTURE Q:  
  1 QI INTEGER,  
  1 Q1,  
  2 QS SCALAR,  
  2 QC CHARACTER(80);  
DECLARE ZQ Q-STRUCTURE;  
.  
.  
.  
STRUCTURE R:  
  1 RI INTEGER,  
  1 R1,  
  2 RS SCALAR,  
  2 RC CHARACTER(80);  
DECLARE ZR R-STRUCTURE;
```

The tree shapes of ZR and ZQ are the same:



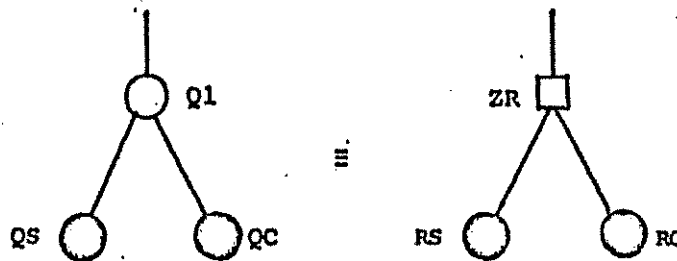
STRUCTURES (CON'T.)

53
409

- ③ MINOR STRUCTURES CAN BE EQUIVALENT EVEN IF THE MAJOR STRUCTURES ARE NOT.

```
STRUCTURE Q:  
  1 QI INTEGER,  
  1 Q1,  
  2 QS SCALAR,  
  2 QC CHARACTER(80);  
DECLARE ZQ Q-STRUCTURE;  
.  
.  
STRUCTURE R:  
  1 RS SCALAR,  
  1 RC CHARACTER(80);  
DECLARE ZR R-STRUCTURE;
```

The tree shapes of ZQ and ZR clearly are not the same. However, the tree shapes of ZQ.Q1 and ZR are the same:



53
410

STRUCTURES (CON'T.)

ADDITIONALLY, HOWEVER, CORRESPONDING NODES MUST AGREE IN ALL ATTRIBUTES,
E.G. DATA TYPE, PRECISION, ARRAYNESS

TYPE	MATCHING REQUIREMENTS
BIT STRING	number of bits (BOOLEAN is equivalent to BIT(1))
CHARACTER	maximum declared length
INTEGER	precision
SCALAR	precision
VECTOR	precision, length
MATRIX	precision, row and column dimensions
STRUCTURE	specified structure template

Examples:

```
STRUCTURE Q:  
  1 Q1 INTEGER,  
  1 Q1,  
  2 QM MATRIX (3,3),  
  2 QC CHARACTER (80);  
DECLARE ZQ Q-STRUCTURE;  
:  
:  
STRUCTURE R:  
  1 R1 INTEGER DOUBLE,  
  1 R1,  
  2 RM MATRIX (3,3),  
  2 RC CHARACTER (80);  
DECLARE ZR R-STRUCTURE;
```

ZQ AND ZR ARE NOT TREE EQUIVALENT

BUT, ZQ.Q1 AND ZR.R1 ARE TREE EQUIVALENT.

STRUCTURES (CON'T.)

FOR TREE EQUIVALENCE, CORRESPONDING TERMINALS OF STRUCTURE TYPE MUST
IN TURN USE TREE-EQUIVALENT TEMPLATES:

EXAMPLE

STRUCTURE P:
 1 S SCALAR,
 1 T SCALAR DOUBLE;
STRUCTURE Q:
 1 U SCALAR,
 1 V SCALAR DOUBLE;
STRUCTURE PP:
 1 A,
 1 B P-STRUCTURE;
STRUCTURE QQ:
 1 F,
 1 G Q-STRUCTURE;
DECLARE X PP-STRUCTURE;
DECLARE Y QQ-STRUCTURE;

THEN X IS EQUIVALENT TO Y.

31
411

STRUCTURES (CON'T.)

51
412

A NESTED STRUCTURE IS NOT THE SAME AS A SIMILAR MINOR STRUCTURE.

```
STRUCTURE Q:  
  1 QS SCALAR,  
  1 QC CHARACTER(80);  
STRUCTURE R:  
  1 RI INTEGER,  
  1 RQ Q-STRUCTURE;  
DECLARE ZR R-STRUCTURE;  
STRUCTURE S:  
  1 SI INTEGER,  
  1 S1,  
  2 SS SCALAR,  
  2 SC CHARACTER(80);  
DECLARE ZS S-STRUCTURE;
```

FUNCTIONALLY THE SAME,
BUT NOT EQUIVALENT

ZS AND ZR ARE NOT EQUIVALENT!

IF WE REDEFINE S WE ARE OK:

```
STRUCTURE S:  
  1 SI INTEGER,  
  1 SQ Q-STRUCTURE;
```

STRUCTURES (CON'T.)

SI
413

STRUCTURE ASSIGNMENTS

SYMBOLIC FORM: $L = R$

- (1) L IS RECEIVING STRUCTURE DATA ITEM WITH POSSIBLE STRUCTURE SUBSCRIPTING.
- (2) R IS EITHER A (POSSIBLY SUBSCRIPTED) STRUCTURE DATA ITEM, OR A STRUCTURE FUNCTION.
- (3) IF L AND R ARE MAJOR OR MINOR STRUCTURES, THEY MUST BE TREE-EQUIVALENT.

STRUCTURES (CON'T.)

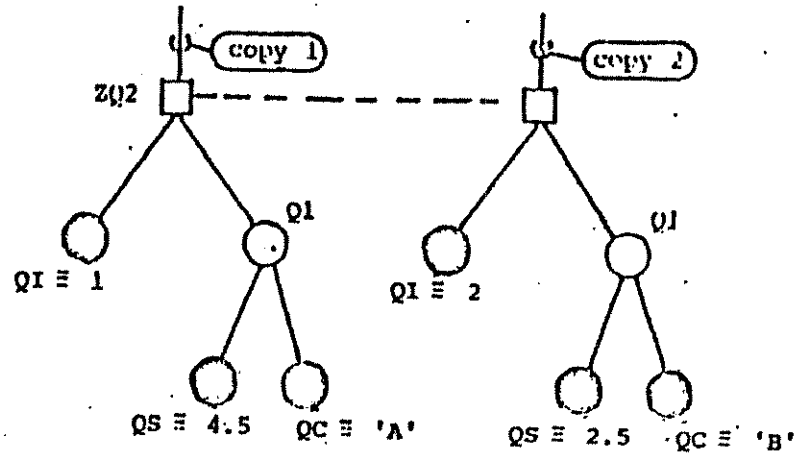
52
414

EXAMPLES

Given:

```
STRUCTURE Q:  
  1 QI INTEGER,  
  1 QI,  
  2 QS SCALAR,  
  2 QC CHARACTER(80);  
DECLARE ZQ1 Q-STRUCTURE;  
DECLARE ZQ2 Q-STRUCTURE(2);
```

where ZQ2 has the values:



37

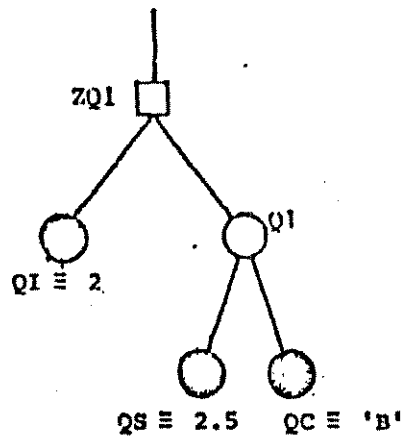
STRUCTURES (CON'T.)

S3
415

then

$$\left. \begin{array}{l} \text{S} \\ \cdot \end{array} \right\} \begin{array}{l} ZQ1 = ZQ2 ; \\ \quad \quad \quad 2 \end{array}$$

results in ZQ1 having the values:

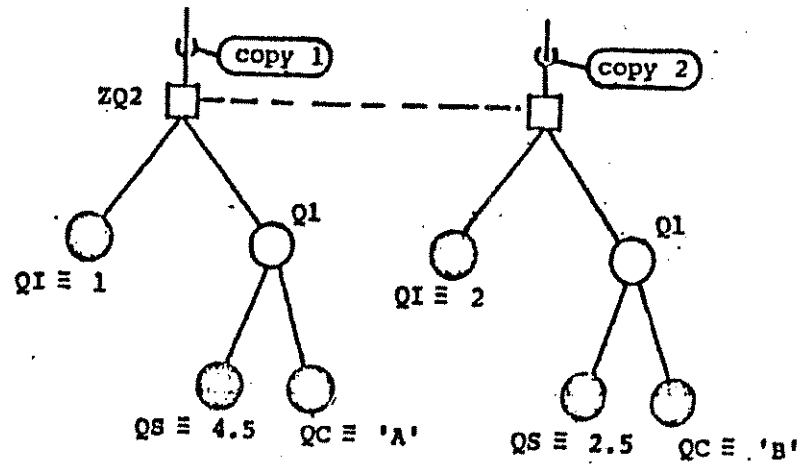


STRUCTURES (CON'T.)

Given:

```
STRUCTURE Q:  
  1 QI INTEGER,  
  1 Q1,  
  2 QS SCALAR,  
  2 QC CHARACTER(80);  
DECLARE ZQ1 Q-STRUCTURE;  
DECLARE ZQ2 Q-STRUCTURE(2);
```

where ZQ2 has the values:



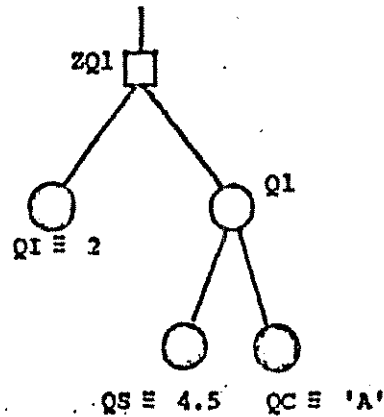
STRUCTURES (CON'T.)

53
417

and if then the following is executed

$$\begin{array}{|l} \text{ZQ1.Q1} = \text{ZQ2.Q1} ; \\ \hline \text{S} \end{array}$$

the values of ZQ1 are modified to:



MULTIPLE ASSIGNMENTS

A NUMBER OF STRUCTURE DATA ITEMS (MAJOR OR MINOR STRUCTURES)
MAY BE ASSIGNED THE SAME VALUES BY MEANS OF A MULTIPLE
ASSIGNMENT:

$$L^1, L^2, L^3 \dots L^n = R^*$$

WHERE:

$$L^1 \dots L^n$$

ARE STRUCTURE DATA ITEMS, AND EACH L^i ARE TREE-EQUIVALENT TO R .

NOTE: NO PARTICULAR ORDER OF ASSIGNMENT CAN BE GUARANTEED!

EXAMPLE:

STRUCTURE Q:

1 I INTEGER,

1 S;

DECLARE Q Q-STRUCTURE(100);

...

Q\$3, Q\$9, Q\$10 = Q\$47;

* R MAY ALSO BE A STRUCTURE FUNCTION.

52
418

STRUCTURES IN RELATIONAL EXPRESSIONS

52
419

- ① STRUCTURE COMPARISONS CAN BE MADE IN RELATIONAL EXPRESSIONS, WHICH IN TURN MAY BE USED IN

IF ...
DO WHILE ...
AND DO UNTIL ... CONSTRUCTS.

- ② ONLY CLASS II COMPARATIVE OPERATIONS MAY BE EMPLOYED ON STRUCTURE DATA ITEMS, I.E., MAJOR AND MINOR STRUCTURES. CLASS II OPERATIONS ARE:

= \neg =

E.G.,

IF $L = R$ THEN DO;
DO WHILE $L \neg = R$;

RULES:

- 1) L AND R ARE EITHER STRUCTURE DATA ITEMS OR STRUCTURE FUNCTIONS.
- 2) L AND R MUST BE TREE-EQUIVALENT.

STRUCTURES IN RELATIONAL EXPRESSIONS (CON'T.)

53
420

- ③ TWO STRUCTURES ARE EQUAL \iff ALL CORRESPONDING TERMINALS HAVE EQUAL VALUES.

EXAMPLES:

STRUCTURE Q;

 1 I INTEGER,

 1 S SCALAR;

DECLARE Q-STRUCTURE INITIAL(6, 18.0), L, R;

 . . .

IF L = R THEN DO;

 . . . TRUE

END;

L.I = 4;

DO WHILE L \neq R,

 . . . TRUE

END;

51
421

STRUCTURE ARGUMENTS AND PARAMETERS

INPUT PARAMETERS (PROCEDURES/FUNCTIONS)

A STRUCTURE DATA ITEM (MAJOR STRUCTURE) CAN BE AN INPUT PARAMETER OF A PROCEDURE OR A FUNCTION. THE TEMPLATE DECLARATION FOR THE STRUCTURE MUST PRECEDE THE STRUCTURE DECLARATION.

THE CORRESPONDING INPUT ARGUMENT CAN BE A MAJOR STRUCTURE, MINOR STRUCTURE OR STRUCTURE FUNCTION -- AND MUST BE TREE-EQUIVALENT!

ASSIGN PARAMETERS (PROCEDURES ONLY)

AN ASSIGN ARGUMENT CAN BE A MAJOR OR MINOR STRUCTURE -- AND MUST BE TREE-EQUIVALENT TO THE CORRESPONDING PARAMETER.

TERMINALS AND MINOR STRUCTURES CAN POSSESS NO "COPYNESS" -- IF THEY ARE CONTAINED IN A MULTI-COPY STRUCTURE, THEN SUBSCRIPTING MUST BE USED TO ISOLATE A SINGLE COPY.


MAJOR STRUCTURES WITH COPIES CAN BE LEFT UNSUBSCRIPTED (FULL COPYNESS) -- OR MUST BE SUBSCRIPTED TO A SINGLE COPY!!

714

STRUCTURE ARGUMENTS AND PARAMETERS (CON'T.)

52
422

EXAMPLE 1: POSITION OF TEMPLATE

```
ANALYZE: PROCEDURE(S1) ASSIGN(S2);  
  STRUCTURE S:  
    1 SI INTEGER,  
    1 SN,  
    2 SS SCALAR,  
    2 SC CHARACTER(80);  
  DECLARE S1 S-STRUCTURE,  
          S2 S-STRUCTURE;  
  
   } executable code  
  
CLOSE ANALYZE;
```

RULE:

PARAMETER STRUCTURE TEMPLATES BEFORE PARAMETER
DECLARATIONS BEFORE LOCAL DATA DECLARATIONS.

7-5

STRUCTURE ARGUMENTS AND PARAMETERS (CON'T.)

EXAMPLE 2: TEMPLATE CAN BE IN AN OUTER SCOPE (E.G. COMPOOL)

33
423

```
CPL: EXTERNAL COMPOOL,  
      STRUCTURE Q:  
        1 I INTEGER,  
        1 V VECTOR,  
        . . .  
      CLOSE CPL,  
COMSUB: PROCEDURE(STRUC),  
        DECLARE STRUC Q-STRUCTURE,  
        . . .  
      CLOSE COMSUB,
```


STRUCTURE ARGUMENTS AND PARAMETERS (CON'T.)

52
427

EXAMPLE 3: LEGAL AND ILLEGAL INVOCATIONS

```
STRUCTURE Q:  
  1 QI INTEGER,  
  1 Q1,  
  2 QS SCALAR,  
  2 QC CHARACTER(80);  
STRUCTURE R:  
  1 RS SCALAR,  
  1 RC CHARACTER(80);  
DECLARE ZQ Q-STRUCTURE,  
        ZR R-STRUCTURE,  
        YQ Q-STRUCTURE(10);
```

• • •

```
TREE: PROCEDURE(D1) ASSIGN(D2);  
      DECLARE D1 R-STRUCTURE,  
             D2 Q-STRUCTURE;
```



} procedure body

```
CLOSE TREE;
```

STRUCTURE ARGUMENTS AND PARAMETERS (CON'T.)

53
425-

THEN:

```
CALL TREE(ZR) ASSIGN(ZQ);  
CALL TREE(ZR) ASSIGN(YQ );  
S  
CALL TREE(ZQ.Q1) ASSIGN(ZQ);  
CALL TREE(ZR) ASSIGN(ZR);
```

4
└── illegal - no tree-
equivalence

STRUCTURE ARGUMENTS AND PARAMETERS (CON'T.)

52
426

EXAMPLE 4: LEGAL AND ILLEGAL INVOCATIONS

```
1 M↑ TESTB:
1 M↑ PROGRAM)
2 M↑   STRUCTURE Q:
2 M↑     1 A,
2 M↑       2 I INTEGER,
2 M↑       2 S SCALAR,
2 M↑     1 B,
2 M↑       2 J INTEGER,
2 M↑       2 T SCALAR;
3 M↑   DECLARE Q Q-STRUCTURE(10) INITIAL(1, 2, 3, 4);
4 M↑   DECLARE R Q-STRUCTURE(5) INITIAL(5, 6, 7, 8);
5 M↑   DECLARE P Q-STRUCTURE;

6 M↑ PROC1:
6 M↑   PROCEDURE ASSIGN(U, V, W);
7 M↑     DECLARE U INTEGER;
8 M↑     DECLARE V Q-STRUCTURE(5);
9 M↑     STRUCTURE X:
9 M↑       1 K INTEGER,
9 M↑       1 SS SCALAR;
10 M↑     DECLARE W X-STRUCTURE(10);
11 M↑   CLOSE;

      E↑
12 M↑   CALL PROC1 ASSIGN(I , [R], [A]);
      S↑
      3
**** FS2   ERROR #1 OF SEVERITY 1. ****
**** THE STRUCTURE COPIES OF ASSIGN ARGUMENT A MUST BE SUBSCRIBED
**** AWAY,
```

53
427

STRUCTURE ARGUMENTS AND PARAMETERS (CON'T.)

```
      E↑
13 N↑   CALL PROC1 ASSIGN(R.A.I , [Q]      , [A]);
      S↑           7      5 AT 2
**** SR3      ERROR #2 OF SEVERITY 1. ****
**** INDEX VALUE IN SUBSCRIPT OF R.A.I IS GREATER THAN THE LEGAL MAXIMUM
**** SV1      ERROR #3 OF SEVERITY 1. ****
**** SUBSCRIPTING OF Q IS ILLEGAL IN CONTEXT OF USE AS AN ASSIGN ARGUMENT
**** FS2      ERROR #4 OF SEVERITY 1. ****
**** THE STRUCTURE COPIES OF ASSIGN ARGUMENT Q MUST BE SUBSCRIPTED
**** AWAY
**** FS2      ERROR #5 OF SEVERITY 1. ****
**** THE STRUCTURE COPIES OF ASSIGN ARGUMENT A MUST BE SUBSCRIPTED
**** AWAY
**** LAST ERROR WAS DETECTED AT STATEMENT 12. ****
```

STRUCTURE ARGUMENTS AND PARAMETERS (CON'T.)

5/
428

STRUCTURE(*)

JUST AS IN THE CASE OF 1-DIMENSIONAL ARRAYS, A STRUCTURE INPUT OR ASSIGN PARAMETER IS ALLOWED TO HAVE A VARIABLE NUMBER OF COPIES, I.E., THE NUMBER OF COPIES IS PASSED TO THE PROCEDURE IN THE STACK AT THE TIME OF INVOCATION.

THE BUILT-IN FUNCTION SIZE MAY LIKEWISE BE USED FOR SUCH UNKNOWN-COPYNESS STRUCTURES TO OBTAIN THE ACTUAL NUMBER OF COPIES.

STRUCTURE ARGUMENTS AND PARAMETERS (CON'T.)

32
429

EXAMPLE 1

```
. . .  
DECLARE R Q-STRUCTURE(10);  
DECLARE S Q-STRUCTURE(50);
```

```
. . .  
PROC: PROCEDURE ASSIGN(D);  
  DECLARE D Q-STRUCTURE(*);
```

```
. . .  
  DO I = 1 TO SIZE(D);
```

```
. . .  
  END;
```

```
. . .  
  CLOSE PROC;
```

```
. . .  
  CALL PROC ASSIGN(R);  
  CALL PROC ASSIGN(S);
```

STRUCTURE ARGUMENTS AND PARAMETERS (CON'T.)

52
430

EXAMPLE 2: ARRAY PROCESSING FEATURE CAN LIKEWISE BE USED

```
STRUCTURE Q;  
  1 SCAL1,  
  1 SCAL2,  
  1 VECTX VECTOR;  
DECLARE Q Q-STRUCTURE(50);  
DECLARE R Q-STRUCTURE(20);  
  . . .  
PROC:  PROCEDURE ASSIGN(E);  
  DECLARE E Q-STRUCTURE(*);  
  . . .  
  E.SCAL1, E.SCAL2 = 0; ←———— ARRAYED MULTIPLE ASSIGNMENT STMT  
  . . .  
CLOSE PROC;  
  . . .  
CALL PROC ASSIGN(Q);  
  . . .  
CALL PROC ASSIGN(R);
```

STRUCTURE FUNCTIONS

5/
431

HAL/S ALLOWS FUNCTIONS OF STRUCTURE TYPE WHICH MAY BE SUBSTITUTED FOR DECLARED STRUCTURES IN MANY OF THE PREVIOUSLY DEFINED STRUCTURE OPERATIONS.

FORM:

```
label: FUNCTION( $i^1$ ,  $i^2$ , ...)  $\alpha$ -STRUCTURE;  
    . . .  
    RETURN structure;  
CLOSE;
```

NOTES:

- (1) A STRUCTURE FUNCTION CANNOT HAVE COPYNESS (REMEMBER THAT ORDINARY FUNCTIONS CANNOT HAVE AN ARRAY DECLARATION).
- (2) THE TEMPLATE α MUST BE DEFINED IN AN OUTER SCOPE PRIOR TO THE FUNCTION HEADER.
- (3) THE INPUT PARAMETER LIST IS OMITTED ENTIRELY IF THERE ARE NO PARAMETERS.

STRUCTURE FUNCTIONS (CON'T.)

53
433

AS IS THE CASE WITH ALL FUNCTIONS, A STRUCTURE FUNCTION
MUST ACTUALLY RETURN A STRUCTURE OF THE REQUISITE TYPE:

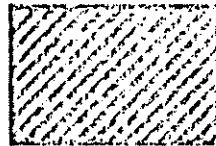
```
STRUCTURE S:
  1 SS SCALAR,
  1 SC CHARACTER(80);
STRUCTURE Q:
  1 QI INTEGER,
  1 Q1 S-STRUCTURE;
:
:
TREE: FUNCTION(D1) S-STRUCTURE;
  DECLARE D1 Q-STRUCTURE;
  :
  :
  RETURN D1.Q1;
  :
  :
  RETURN D1;
  :
  :
CLOSE TREE;
      _____ illegal - lack of
                        tree-equivalence
```

NOTE: REMEMBER THAT AGGREGATE DATA (VECTORS, MATRICES, CHARACTER
STRINGS, ARRAYS, AND STRUCTURES) ARE NEVER PHYSICALLY
PASSED TO OR FROM A PROCEDURE OR FUNCTION. AGGREGATES
ARE ALWAYS PASSED BY NAME (REFERENCE) -- NEVER BY VALUE!

STRUCTURE FUNCTIONS (CON'T.)

A STRUCTURE FUNCTION IS INVOKED BY EMPLOYING ITS NAME IN A REFERENCE CONTEXT. IT SHOULD BE CLEAR, HOWEVER, THAT A STRUCTURE FUNCTION IS NOT REALLY A STRUCTURE -- I.E., MINOR STRUCTURES OR TERMINALS OF IT CANNOT BE REFERENCED.

```
STRUCTURE Q:  
  1 Q1 INTEGER,  
  1 Q1,  
  2 QS SCALAR,  
  2 QC CHARACTER(80);  
DECLARE ZQ Q-STRUCTURE;  
TREE: FUNCTION Q-STRUCTURE;
```



} function body

```
CLOSE TREE;
```

```
.  
.  
.
```

```
ZQ = TREE;  
ZQ.Q1 = TREE.Q1;
```

```
legal invocation  
illegal invocation
```

ALSO,

```
IF ZQ = TREE THEN DO;
```

ARRAY PROCESSING

S/
435

EXAMPLE

STRUCTURE Q:

```
1 I INTEGER,  
1 A ARRAY(4,9) SCALAR,  
1 M ARRAY(6,10,4) MATRIX,  
1 C ARRAY(6) CHARACTER(9);  
DECLARE Q Q-STRUCTURE(16);  
DECLARE J ARRAY(20) INTEGER;  
DECLARE B ARRAY(16,4) SCALAR;  
DECLARE N ARRAY(16,10,4) MATRIX;  
DECLARE D ARRAY(6) CHARACTER(9);
```

7-19

52
436

ARRAY PROCESSING (CON'T.)

	α	TYPE
Q	+ {1:16}	MAJOR STRUC
I	+ {1:16}	INTEGER
A	+ {3:16,4,9}	SCALAR
M	+ {4:16,6,10,4}	MATRIX
C	+ {2:16,6}	CHARACTER
J	+ {1:20}	INTEGER
B	+ {2:16,4}	SCALAR
N	+ {3:16,10,4}	MATRIX
D	+ {1:6}	CHARACTER

7-20

ARRAY PROCESSING (CON'T.)

53
437

ARRAYNESS IS A FUNCTION OF SUBSCRIPTING!

$$A^{\alpha}(*; *, 3) \rightarrow \{2:16, 4\}$$
$$M(*; 3, *, *) \rightarrow \{3:16, 10, 4\}$$

TWO OPERANDS HAVE MATCHING ARRAYNESS IF THE N-TUPLES ARE IDENTICAL:

$$A(*; *, 3) \equiv B \equiv \{2:16, 4\}$$
$$M(*; 3, *, *) \equiv N \equiv \{3:16, 10, 4\}$$

B. ARRAYED EXPRESSIONS

AN ARRAYED EXPRESSION IS AN ORDINARY EXPRESSION IN WHICH THE OPERANDS HAVE ARRAYNESS. AN ARRAYED EXPRESSION CAN BE ASSIGNED TO (OR COMPARED AGAINST) AN ARRAYED DATA ITEM.

1-2-1

ARRAY PROCESSING (CON'T.)

SI
438

EXAMPLE

DECLARE ARRAY(50), S, T;

DECLARE I INTEGER;

...

DO FOR I = 1 TO 50;

S\$I = T\$I; ← ORDINARY ASSIGNMENT

END;

...

S = T; ← ARRAYED ASSIGNMENT STATEMENT

- IN AN ARRAYED EXPRESSION, EACH OPERAND MAY OR MAY NOT HAVE ARRAYNESS -- BUT ALL OPERANDS THAT HAVE ARRAYNESS MUST MATCH IN ARRAYNESS.

ARRAY PROCESSING (CON'T.)

52
431

EXAMPLE:

DECLARE ARRAY(3,6) INTEGER, I, J;

DECLARE K ARRAY(4,10,4) INTEGER;

DECLARE S ARRAY(3,6) SCALAR;

I + J IS LEGAL (ARRAYNESSES ARE {2:3,6})

I + J + 6 IS LEGAL (ARRAYNESSES ARE {2:3,6})

I + K IS ILLEGAL (ARRAYNESS OF K IS {3:4,10,4})

I + K\$(3 AT 1, 6 AT 2,3) IS LEGAL

(K HAS BEEN REDUCED TO ARRAYNESS {2:3,6} BY SUBSCRIPTING)

I + J + S IS LEGAL (RESULT WILL BE AN ARRAY(3,6) OF SCALARS)

I J S IS LEGAL (RESULT WILL BE AN ARRAY(3,6) OF SCALARS)

1.2.3

USER-DEFINED FUNCTIONS INVOLVED IN AN ARRAYED EXPRESSION
WILL BE INVOKED ONLY ONCE UNLESS AN ARRAYED ARGUMENT IS BEING
PASSED TO A FUNCTION WHICH ACCEPTS UNARRAYED ARGUMENTS.

52
440

```
DECLARE ARRAY(10), I, J;
DECLARE K;
F1: FUNCTION(ARG);
    DECLARE ARG;
    . . .
CLOSE F1;
F2: FUNCTION(ARG);
    DECLARE ARG ARRAY(10);
    . . .
CLOSE F2;
    . . .
. . . I + J + F1(K);
    ↑
    F1 INVOKED ONCE
. . . I + J + F2(I);
    ↑
    F2 INVOKED ONCE
. . . I + J + F1(I);
    ↑
    F1 INVOKED 10 TIMES
. . . I + J + F2(K);
    ↑
    ERROR-PARAMETER MISMATCH
```

ARRAY PROCESSING (CON'T.)

53
441

BUILT-IN FUNCTIONS, E.G., SQRT, SIN, ..., ARE BETTER BEHAVED --
THE COMPILER KNOWS THAT THEY CANNOT INFLUENCE "EXTERNAL" DATA.

- A BUILT-IN FUNCTION WITH NO ARGUMENTS OR WITH ARGUMENTS
THAT ARE NOT ARRAYED MAY ULTIMATELY BE EVALUATED ONLY ONCE
IN AN ARRAYED EXPRESSION

EXAMPLE

DECLARE C SCALAR INITIAL(5);

DECLARE ARRAY(20) SCALAR, S, T;

...

S = T + SIN(C);

CURRENTLY WILL BE EVALUATED 20 TIMES



ULTIMATELY THE EFFECT MAY BE ...

COMPILER-TEMPORARY = SIN(C)

S = T + COMPILER-TEMPORARY

AGAIN, IF SIN HAD BEEN A USER FUNCTION SUCH OPTIMIZATION IS NOT
POSSIBLE.

ARRAY PROCESSING (CON'T.)

S1
442

- IF THE BUILT-IN FUNCTION HAS ARRAYED ARGUMENTS (AND THE ARRAYNESSES OF ALL ARGUMENTS MATCH THE ARRAYNESS OF THE EXPRESSION -- AN ERROR WOULD OTHERWISE RESULT) THEN THE FUNCTION IS INVOKED ONCE PER ELEMENTAL EVALUATION. ON EACH EVALUATION THE FUNCTION OPERATES ON SUCCESSIVE ELEMENTS OF THE ARRAYED INPUT ARGUMENTS.

EXAMPLE

```
DECLARE ARRAY(20), S, T, U,
```

```
S = ARCTAN2(T,U),
```

```
THIS IS EQUIVALENT TO:
```

```
DO FOR I = 1 TO 20,
```

```
S$I = ARCTAN2(T$I, U$I);
```

```
END;
```

ARRAY PROCESSING (CON'T.)

S2
443

IMPORTANT NOTE:

IF THE BUILT-IN FUNCTION NORMALLY TAKES AN ARRAYED ARGUMENT THEN THE FUNCTION ACTS ON THE WHOLE ARGUMENT EACH CYCLE THROUGH THE ARRAYED EXPRESSION.

SUCH BUILT-IN FUNCTIONS ARE:

(α IS A 1-, 2-, OR 3-DIMENSIONAL ARRAY OF INTEGERS OR SCALARS)

MAX(α)	MAXIMUM (GREATEST) ELEMENT OF THE ARRAY
MIN(α)	MINIMUM (LEAST) ELEMENT OF THE ARRAY
PROD(α)	PRODUCT OF ALL ELEMENTS OF THE ARRAY
SUM(α)	SUM OF ALL ELEMENTS OF THE ARRAY

53
444

ARRAY PROCESSING (CON'T.)

EXAMPLE

DECLARE ARRAY(20), S, T, U;

S = T + SUM(U);

ARRAYNESS OF U NEED NOT MATCH S & T.

THIS IS EQUIVALENT TO:

DO FOR I = 1 TO 20;

S\$I = T\$I + SUM(U);

END;

ACTS LIKE A CONSTANT

C. ARRAYED ASSIGNMENTS

AN ARRAYED ASSIGNMENT IS OF ONE OF THE FOLLOWING TWO FORMS:

- ① ARRAYED DATA ITEM = UNARRAYED EXPRESSION
- ② ARRAYED* DATA ITEM = ARRAYED* EXPRESSION

* - IN THIS CASE, OF COURSE, ALL ARRAYNESSES MUST MATCH.

7-50



3

ARRAY PROCESSING (CON'T.)

52
448

IN CASE ①, ALL ELEMENTS OF THE ARRAY ON THE LEFT-HAND-SIDE OF THE ASSIGNMENT STATEMENT ARE IDENTICALLY SET EQUAL TO THE RIGHT-HAND EXPRESSION:

DECLARE ARRAY(10), A, B;

...

A = 0;

ALL 10 ELEMENTS OF A ARE ZEROED

A = SUM(B);

ALL 10 ELEMENTS OF A ARE SET EQUAL TO THE SUM OF ALL ELEMENTS OF B

A = COS(15);

ALL 10 ELEMENTS OF A ARE SET EQUAL TO THE SCALAR COS(15)

IN CASE ②, SUCCESSIVE ELEMENTS OF THE LEFT-HAND-SIDE ARRAYED DATA ITEM ARE SET TO CORRESPONDING EVALUATIONS OF THE RIGHT-HAND-SIDE ARRAYED EXPRESSION:

ARRAY PROCESSING (CON'T.)

53
446

```
DECLARE ARRAY(10) INTEGER,  
    M, N, P;  
DECLARE ARRAY(5,10), S, T, U;  
    . . .  
P = M + N;
```

IS EQUIVALENT TO:

```
DO FOR I = 1 TO 10,  
    P$I = M$I + N$I,  
END,  
    U = S T,
```

IS EQUIVALENT TO:

```
DO FOR I = 1 TO 5,  
    DO FOR J = 1 TO 10,  
        U$(I,J) = S$(I,J) T$(I,J),  
    END,  
END;
```

ARRAY PROCESSING (CON'T.)

51
447

```
1 M↑ TESTD:
1 M↑ PROGRAM;
2 M↑   STRUCTURE Q:
2 M↑     1 II ARRAY(2, 3) INTEGER,
2 M↑     1 BB ARRAY(6) BIT(9),
2 M↑     1 MM ARRAY(5, 10, 15) MATRIX(4, 5),
2 M↑     1 N ARRAY(10) INTEGER,
2 M↑     1 I INTEGER;
3 M↑   DECLARE Q Q-STRUCTURE(30) INITIAL(6#1, 6#BIN'1', 750#(20#0), 10#4, 5);
4 M↑   STRUCTURE S:
4 M↑     1 C ARRAY(10) VECTOR(5);
5 M↑   DECLARE S S-STRUCTURE(5);
6 M↑   DECLARE T ARRAY(5, 10) VECTOR(5);
7 M↑   DECLARE U ARRAY(5, 10, 5) SCALAR;
  C↑
  E↑
  -
  -
8 M↑   [[C]] = [[MM]]
  S↑
  E↑
  -
  -
  5 AT 9, 5, *, 7: 3, *
9 M↑   [T] = [[MM]]
  S↑
  E↑
  -
  -
  5 AT 9, 5, *, 7: 3, *
10 M↑  [U] = [[MM]]
  S↑
  -
  -
  5 AT 9, 5, *, 7: 3, *
*** AA1      ERROR #1 OF SEVERITY 1. *****
*** ARRAYNESS OF LEFT HAND SIDE OF ASSIGNMENT DOES NOT MATCH THAT OF RIGHT HAND SIDE
*** AV1      ERROR #2 OF SEPERITY 1. *****
*** TYPE OF U IS ILLEGAL FOR ASSIIONMENT FROM GIVEN RIGHT-HAND SIDE.
```


ARRAY PROCESSING (CON'T.)

```

      E↑
      8 N↑ DO FOR TEMPORARY I = 1 TO SIZE(E1)
      9 N↑   E. SCAL1 , E. SCAL2 = 0)
      S↑   I           I
10 N↑   END;

      ST#8 EQU *
          L 7,12(0)
          STH 7,17(0)
          LFXI 7,1
      LBL#6 EQU *
          STH 7,16(0)
          CH 7,17(0)
          BC 1, **15
      ST#9 EQU *
          NIH 7,707(1)
          LR 6,7
          SRA 6,1
          SRA 7,1
          SER 0,0
          LH 2,10(0)
          STE 0,0(7,2)
          STE 0,2(6,2)
      ST#10 EQU *
      LBL#8 EQU *
          LFXI 7,1
          AH 7,16(0)
          BC 7, *-16
      LBL#7 EQU *

```

E+2
I
LBL#7
H'10'
E
I
LBL#6

} 20 HW

ARRAY PROCESSING (CON'T.)

```

E↑
11 M↑      DO FOR TEMPORARY I = 1 TO SIZE(E1);
12 M↑      E. SCAL1 = 0;
S↑
13 M↑      I
E. SCAL2 = 0;
S↑
14 M↑      I
END;
    
```

ST#11	EQU	*					
	L	7,12(0)	}	22 HW		E+2	
	STH	7,17(0)					
	LFXI	7,1					
LBL#9	EQU	*					I
	STH	7,16(0)					
	CH	7,17(0)					
	BC	1,*+17					LBL#10
ST#12	EQU	*					H'10'
	MIH	7,707(1)					
	SRA	7,1					
	SER	0,0					E
	LH	2,10(0)					
	STE	0,0(7,2)					
ST#13	EQU	*					I
	LH	7,16(0)					H'10'
	MIH	7,707(1)					
	SRA	7,1					
	STE	0,2(7,2)					
ST#14	EQU	*					
LBL#11	EQU	*					
	LFXI	7,1					
	AH	7,16(0)			I		
	BC	7,*-18			LBL#9		
LBL#10	EQU	*					

```

15 M↑      CLOSE PROC;
    
```

```

ST#15     EQU     *
LBL#4     EQU     *
          LN      0(0)
          BCR     7,4
    
```

7-23

ARRAY PROCESSING (CON'T.)

```
      E↑
16  M↑ CALL PROC ASSIGN([Q]);
      E↑
17  M↑ CALL PROC ASSIGN([R]);
```

```
      ST#16 EQU *
          LA 5,-8(1) Q
          LHI 6,50
          BAL 4,0(3) R2TESTC
      ST#17 EQU *
          LA 5,492(1) R
          LHI 6,20
          BAL 4,0(3) R2TESTC
```

7-34



ARRAY PROCESSING (CON'T.)

FURTHER ARRAYED ASSIGNMENT EXAMPLES

Given:

```
DECLARE INTEGER,  
I1 ARRAY(2,3),  
I2,  
I3 ARRAY(2,3),  
I4 ARRAY(4);
```

then

```
I1 = I2;      (ARRAYED DATA ITEM = UNARRAYED EXPRESSION)
```

is an arrayed assignment in which all elements of I1 are assigned the value of I2.

```
I1 = I3;      (ARRAYED DATA ITEM = ARRAYED EXPRESSION)
```

assigns each element of I3 to the corresponding element of I1.

```
I1 = I4;
```

is illegal because the arrayness of the receiving data item is {2:2,3} while that of the right hand side is {1:4}.

```
I2 = I1;
```

is illegal because the right hand side has arrayness while the receiving data item has none.

ARRAY PROCESSING (CON'T.)

NOTE THAT THE FOLLOWING ARE ALSO POSSIBLE:

$$I1 = I1 + I3;$$

$$I1 = I1 I3;$$

$$I1 = I3**2;$$

$$I1 = I3/I2;$$

ARRAY PROCESSING (CON'T.)

Further given:

```
STRUCTURE Q:  
  1 QI INTEGER,  
  1 Q1,  
  2 QS ARRAY(4) SCALAR,  
  2 QC CHARACTER(80);  
DECLARE ZQ1 Q-STRUCTURE(2);  
DECLARE ZQ2 Q-STRUCTURE(2);  
DECLARE S ARRAY(2,4) SCALAR;
```

the following assignments are legal:

```
ZQ1 = ZQ2;  
ZQ1.Q1 = ZQ2.Q1;  
ZQ1.Q1.QS = ZQ2.Q1.QS;  
ZQ1.Q1.QS = S;
```

ARRAY PROCESSING (CON'T.)

D. MULTIPLE ASSIGNMENTS

MULTIPLE ASSIGNMENTS HAVE BEEN DISCUSSED PREVIOUSLY.
TO EXTEND THEM TO ALLOW ARRAYED BEHAVIOR WE NEED THE
FOLLOWING ADDITIONAL RULE:

IF ONE RECEIVING DATA ITEM POSSESSES
ARRAYNESS, THEN ALL MUST POSSESS
MATCHING ARRAYNESS.

ARRAY PROCESSING (CON'T.)

EXAMPLES OF MULTIPLE ARRAYED ASSIGNMENTS:

Given:

```
DECLARE INTEGER,  
I1 ARRAY(2,3),  
I2,  
I3 ARRAY(4),  
I4 ARRAY(2,3);
```

then

```
I1, I4 = I2;
```

is legal since the arrayness of I1 and I4 match.

However, both of the following are illegal:

```
I1, I2 = I2;  
I1, I3 = I4;
```


ARRAY PROCESSING (CON'T.)

```
DECLARE I ARRAY(3) INTEGER,  
        M MATRIX(2,2),  
        MA ARRAY(3) MATRIX(2,2),  
        MB ARRAY(2) MATRIX(2,2);
```

Let $M \equiv \begin{bmatrix} 1.75 & 0.25 \\ 0.75 & 1.25 \end{bmatrix}$ and $I \equiv \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}$

then $M_{I,*} \equiv \begin{pmatrix} M_{2,*} \\ M_{1,*} \\ M_{1,*} \end{pmatrix} \equiv \begin{pmatrix} [1.75 \ 0.25] \\ [1.75 \ 0.25] \\ [1.75 \ 0.25] \end{pmatrix}$

- a linear 3-array of 2-vectors: subscripting has reduced M from a matrix to a row-vector, but since I is arrayed, the entire operand has an effective arrayness even though M itself has not.

ARRAY PROCESSING (CON'T.)

Given:

```
DECLARE ARRAY(2,3),  
V VECTOR(3),  
I INTEGER;
```

$$\text{with } V \equiv \begin{pmatrix} \begin{bmatrix} 1.5 \\ 2.5 \\ 3.5 \end{bmatrix} & \begin{bmatrix} 4.5 \\ 5.5 \\ 6.5 \end{bmatrix} & \begin{bmatrix} 7.5 \\ 8.5 \\ 9.5 \end{bmatrix} \\ \begin{bmatrix} -0.5 \\ -1.5 \\ -2.5 \end{bmatrix} & \begin{bmatrix} -3.5 \\ -4.5 \\ -5.5 \end{bmatrix} & \begin{bmatrix} -6.5 \\ -7.5 \\ -8.5 \end{bmatrix} \end{pmatrix}$$

$$\text{and } I \equiv \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$$

then

$$V_{*,*,I} \longleftarrow V\$(*,*:I)$$

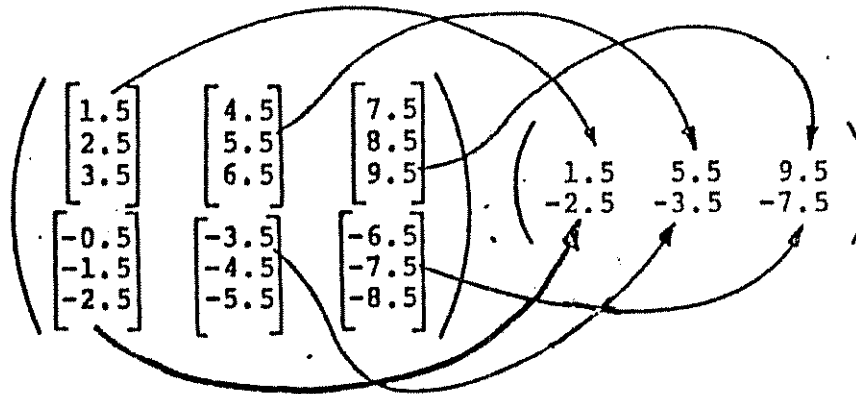
is equivalent to

$$V_{i,j:I_{i,j}} \text{ for } 1 \leq i \leq 2, 1 \leq j \leq 3$$

NOTE THAT THE ARRAYNESS OF
V\$(*,*:I) MATCHES THE
ARRAYNESS OF I

ARRAY PROCESSING (CON'T.)

The arrayed vector subscript I selects an array of scalars from the vector array V as shown below:



In assignment context, the following values of V would be changed:

$$\begin{pmatrix} V_{1,1:1} & V_{1,2:2} & V_{1,3:3} \\ V_{2,1:3} & V_{2,2:1} & V_{2,3:2} \end{pmatrix}$$

ARRAY PROCESSING (CON'T.)

E. ARRAYED SUBSCRIPTING

Q. WHAT HAPPENS WHEN A SUBSCRIPT IS ITSELF AN ARRAY
OR HAS ARRAYNESS?

A. THAT IS A GOOD QUESTION.

SOME RULES:

- (1) IF BOTH THE OPERAND (I.E., DATA ITEM) AND THE SUBSCRIPT
HAVE ARRAYNESS THEN THE ARRAYNESSES MUST MATCH! IN
THIS CASE, THE WHOLE OPERAND IS CONSIDERED TO HAVE THAT
SAME ARRAYNESS.
- (2) IF ONLY THE SUBSCRIPT IS ARRAYED, THEN THE OPERAND
ITSELF BECOMES ARRAYED, I.E., IT INHERITS THE ARRAYNESS
OF ITS SUBSCRIPT.

ARRAY PROCESSING (CON'T.)

NOTES:

- (A) EXPONENTS CAN HAVE ARRAYNESS ALSO.
- (B) SUBSCRIPTS, OF COURSE, CAN IN TURN HAVE SUBSCRIPTS AD INFINITUM. ONE OR MORE OF THESE CAN BE ARRAYED -- PROVIDED ALL ARRAYNESSES MATCH.

NOW,

- (1) IF THE SUBSCRIPTED OPERAND IS PART OF AN ARRAYED EXPRESSION (E.G., IN AN ARRAYED ASSIGNMENT) THEN THE ARRAYED SUBSCRIPT(S) ARE EVALUATED ONCE PER ELEMENTAL EVALUATION OF THE EXPRESSION. NOTE THAT ALL ARRAYNESSES MUST MATCH.
- (2) IF THE SUBSCRIPTED OPERAND IS A RECEIVING DATA ITEM IN AN ASSIGNMENT (LEFT-HAND SIDE) THEN THE ARRAYED SUBSCRIPT IS EVALUATED ONCE DURING EACH ELEMENTAL ASSIGNMENT. AGAIN, ARRAYNESSES MUST MATCH.

ARRAY PROCESSING (CON'T.)

IF

DECLARE I ARRAY(2,3) INTEGER

INITIAL(1,2,3,3,1,2);

SO THAT

$$I = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$$

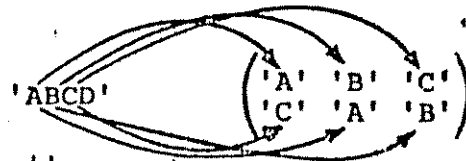
Note that an arrayed subscript can actually generate arrayness in an unarrayed data item. For example, if

C is an unarrayed character string with C ≡ 'ABCD'

then

C_I has the arrayness of I.

The values are selected as follows:



Then C || C_I would be an arrayed expression with values:

$$\begin{pmatrix} 'ABCDA' & 'ABCDB' & 'ABCDC' \\ 'ABDC' & 'ABCDA' & 'ABCDB' \end{pmatrix}$$

ARRAY PROCESSING (CON'T.)

In an assignment context, the following values of C would be changed:

$$\begin{pmatrix} C_1 & C_2 & C_3 \\ C_3 & C_1 & C_2 \end{pmatrix}$$

Note that values of C_1 , C_2 and C_3 would be each changed by two elemental assignments. The results of this assignment are therefore likely to be implementation dependent.

ARRAY PROCESSING (CON'T.)

$$\text{Let } MA \equiv \begin{pmatrix} [1.0 & 0.0] \\ [3.0 & 2.0] \\ [4.0 & 7.0] \\ [6.0 & 5.0] \\ [8.0 & 3.0] \\ [4.0 & 9.0] \end{pmatrix} \quad \begin{array}{l} \textcircled{1} I_1 = 2 \\ \textcircled{2} I_2 = 1 \\ \textcircled{3} I_3 = 1 \end{array}$$

$$\text{Then } MA_{*,1,*} \equiv \begin{pmatrix} M_{1,2,*} \\ M_{2,1,*} \\ M_{3,1,*} \end{pmatrix} \equiv \begin{pmatrix} [3.0 & 2.0] \\ [4.0 & 7.0] \\ [8.0 & 3.0] \end{pmatrix}$$

is also a linear 3-array of 2-vectors: now however MA and I both have arrayness (which correctly match). Three parallel subscript evaluations are effectively performed using corresponding array elements of MA and I each time.

Note $MB_{*,1,*}$ is illegal since the arrayness of MB does not match the arrayness of I.

However $MB_{*,I_1 \text{ TO } 2,*}$ is legal since array subscripting has been used on I to force arrayness matching.

$$\text{If } MB \equiv \begin{pmatrix} [0.5 & 0.5] \\ [0.1 & 0.3] \\ [0.2 & 0.7] \\ [0.4 & 0.8] \end{pmatrix} \quad \begin{array}{l} \textcircled{1} I_1 = 2 \\ \textcircled{2} I_2 = 1 \end{array}$$

$$\text{then } MB_{*,I_1 \text{ TO } 2,*} \equiv \begin{pmatrix} MB_{1,2,*} \\ MB_{2,1,*} \end{pmatrix} \equiv \begin{pmatrix} [0.1 & 0.3] \\ [0.2 & 0.7] \end{pmatrix}$$

ARRAY PROCESSING (CON'T.)

3/
448

F. ARRAYED COMPARISONS

ARRAYED OPERANDS MAY BE USED IN RELATIONAL EXPRESSIONS --
BUT IF ONE OR BOTH OPERANDS OF A COMPARISON HAVE ARRAYNESS,
THEN ONLY THE CLASS II COMPARATIVE OPERATORS MAY BE USED,
I.E., = AND \neq .

ADDITIONALLY, OF COURSE, THE ARRAYNESSES MUST MATCH.

FURTHERMORE, A BOOLEAN EXPRESSION IN A CONDITIONAL CLAUSE
CAN BE ARRAYED:

```
DECLARE BOOL ARRAY(10) BOOLEAN;
```

```
...
```

```
IF [BOOL] THEN DO;
```

IN THIS CASE THE THEN CLAUSE WILL BE EXECUTED IF AND ONLY IF
ALL 10 BOOLEANS ARE TRUE.

ARRAY PROCESSING (CON'T.)

32
449

EXAMPLES

```
DECLARE I ARRAY(4) INTEGER
        INITIAL(1,2,0,3),
        J ARRAY(4) INTEGER
        INITIAL(1,2,0,3),
        K ARRAY(4) INTEGER
        INITIAL(3,1,2,6),
        L ARRAY(2) INTEGER
        INITIAL(0,3),
```

- (1) IF I = J THEN
 TRUE
- (2) IF J = K THEN
 FALSE
- (3) IF J \neq K THEN
 TRUE

ARRAY PROCESSING (CONT'D.)

```
1  NT  S1.
2  NT  PROGRAM.
3  NT  DECLARE ARRAY(4) BOOLEAN INITIAL(2#OFF, 2#ON), A;
4  NT  DECLARE ARRAY(3) INTEGER,
5  NT  B INITIAL(2),
6  NT  C CONSTANT(2, 3, 1),
7  NT  D CONSTANT(2),
8  NT  E INITIAL(2),
9  NT  B1;
10 NT  ET
11 NT  IF [A] THEN
12 NT  WRITE(6) 'A';
13 NT  ET
14 NT  IF [A] THEN
15 NT  ST 2 AT 1:
16 NT  WRITE(6) 'A$(2 AT 1:)'
17 NT  ET
18 NT  IF [A] THEN
19 NT  ST 2 AT 3:
20 NT  WRITE(6) 'A$(2 AT 3:)'
21 NT  [B1] = D
22 NT  ST [B]
23 NT  [B1] = D
24 NT  ST [C]
25 NT  [B1] = E
26 NT  ST [B]
27 NT  [B1] = E
28 NT  ST [D]
29 NT  ET [B]
30 NT  [B1] = [D]
31 NT  ST [C]
32 NT  [B1] = [D]
33 NT  ST [B]
34 NT  ET [B]
35 NT  [B1] = [E]
36 NT  ST [D]
```

ARRAY PROCESSING (CONT'D.)

```
16  NP      (0)
17  NP      (B1) = (E)
18  NP      (E) = SIZE(B1)
```

```
19  NP  F: FUNCTION(X),
20  NP      DECLARE X ARRAY(*) INTEGER,
21  NP      RETURN SIZE(X),
22  NP      CLOSE
```

```
23  NP      (E) = F(B1),
24  NP      CLOSE
```

ARRAY PROCESSING (CON'T.)

31
451

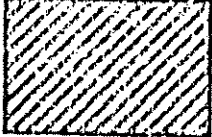
G. REVIEW OF INDEFINITELY ARRAYED PARAMETERS

INDEFINITELY ARRAYED PARAMETERS

THE PARAMETERS OF FUNCTIONS AND THE INPUT OR ASSIGN PARAMETERS OF PROCEDURES MAY BE DECLARED TO BE INDEFINITE ARRAYS. THE FORM OF ARRAY SPECIFICATION IS:

ARRAY(*)

Examples:

```
TWICE: PROCEDURE(A) ASSIGN(B);  
      DECLARE A ARRAY(*) VECTOR(3);  
      DECLARE B ARRAY(*) BIT(16);  
       } procedure body  
CLOSE TWICE;
```

REMEMBER THAT THE SIZE FUNCTION CAN BE USED TO FIND THE ACTUAL SIZE OF THE ARRAY AT RUN TIME.

7-513

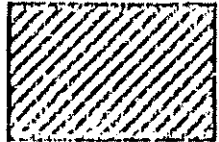
THE NUMBER OF MULTIPLE COPIES IN A STRUCTURE PARAMETER
MAY ALSO BE MADE INDEFINITE USING THE FOLLOWING FORM:

81
452

STRUCTURE(*)

Example:

```

FUN: FUNCTION(C) SCALAR;
  STRUCTURE Q:
    1 QI INTEGER,
    1 QS SCALAR;
  DECLARE C Q-STRUCTURE(*);
   } function body
CLOSE FUN;

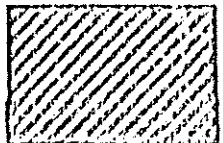
```

HERE, TOO, THE SIZE FUNCTION CAN BE USED.

Note that the ability to define an indefinite array does
not extend to an arrayed structure terminal.

Example:

```

BAD: FUNCTION(C) SCALAR;
  STRUCTURE Q:
    1 QI INTEGER,
    1 QS ARRAY(*) SCALAR; + illegal
  DECLARE C Q-STRUCTURE;
   } function body
CLOSE BAD;

```

7-54

8

81
453

ARRAY PROCESSING (CON'T.)

H. ARRAYED PROCEDURE ARGUMENTS

BOTH INPUT AND ASSIGN ARGUMENTS OF A PROCEDURE INVOCATION MAY POSSESS ARRAYNESS, HOWEVER, THE CORRESPONDING PARAMETERS OF THE PROCEDURE MUST BE ARRAYED ALSO -- AND HAVE THE SAME ARRAYNESS.

IF THE PARAMETER IS ARRAY(*) THEN ONE SHOULD PASS IN A 1-DIMENSIONAL ARRAY.

• INPUT ARGUMENTS

THE INPUT PARAMETER CAN BE VIEWED AS BEING ASSIGNED INTO FROM THE INPUT ARGUMENT. ARRAYNESSES MUST MATCH, BUT FLEXIBILITY IS ALLOWED IN THAT THE INPUT ARGUMENT MAY BE A NON-CONTIGUOUS ARRAY AND/OR OF DIFFERING PRECISION. IT MAY EVEN BE OF DIFFERING DATA TYPE IF AN APPROPRIATE IMPLICIT CONVERSION CAPABILITY EXISTS.

ARRAY PROCESSING (CON'T.)

52
454

EXAMPLE:

STRUCTURE Q:

1 I INTEGER,

1 S SCALAR DOUBLE,

1 M MATRIX;

.DECLARE Q Q-STRUCTURE(100);

. . .

{ ALPHA: PROCEDURE(T);

 DECLARE T ARRAY(50) SCALAR;

 . . .

 CLOSE ALPHA;

 . . .

CALL ALPHA(I\$(50 AT 27));

↑
HAS DATA TYPE INTEGER (IMPLICITLY CONVERTIBLE
TO SCALAR) AND ARRAYNESS (1:50)

ARRAY PROCESSING (CON'T.)

51
453

• ASSIGN ARGUMENTS

RULES (SIMILAR TO THOSE FOR NON-ARRAYED ASSIGN PARAMETERS)

1. The arrayness of the argument must match that of the corresponding parameter.
2. If the parameter is an indefinite array, arrayness matching is ensured if the corresponding argument is a 1-dimensional array.
3. If the argument is part of a structure which has multiple copies, structure subscripting must be used to limit the number of copies in the argument to one.
4. If array subscripting is present it must be such as to select one array element only.
5. If component subscripting is present, where necessary array subscripting must be used to limit the number of array elements in the argument to one.

ARRAY PROCESSING (CON'T.)

NOTE: THE MORE STRINGENT RULES FOR ASSIGN PARMS/ARGS
RESULT FROM THE FOLLOWING FACTS:

- 1) WHETHER INPUT OR ASSIGN, ARRAYS
ARE PASSED BY POINTER (REFERENCE).
ON THE INPUT SIDE THIS MAY BE A
POINTER TO AN ARRAY TEMPORARY
CREATED BY THE COMPILER TO MAKE
DATA CONTIGUOUS, CHANGE PRECISION,
ETC.
- 2) ON THE ASSIGN SIDE, THE PROCEDURE
MODIFIES THE ORIGINAL DATA DIRECTLY.

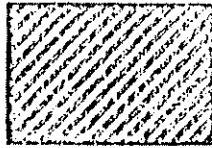
ARRAY PROCESSING (CON'T.)

EXAMPLE 1:

```

ONE: PROCEDURE (A) ASSIGN (B);
      DECLARE A ARRAY (2,3) SCALAR,
              B ARRAY (4) BIT (16);

```



} procedure body

```

CLOSE ONE;

```

and the following data declarations:

```

DECLARE P1 ARRAY (2,3) SCALAR,
        P2 ARRAY (2,5) SCALAR,
        P3 ARRAY (4) BIT (16),
        P4 SCALAR,
        P5 ARRAY (2,5) BIT (16),
        P6 BIT (16);

```

then some legal and illegal invocations of the procedure are as follows:

```

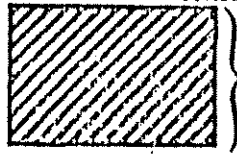
S CALL ONE (P1) ASSIGN (P3);
S CALL ONE (P2 *,1 TO 3) ASSIGN (P3);
S CALL ONE (P2 + P1 - P4) ASSIGN (P6);
  *,3 TO 5 illegal - not arrayed
S CALL ONE (P4) ASSIGN (P5 1,1 TO 4);
  legal arrayness but
  illegal subscript
  illegal - not arrayed

```

ARRAY PROCESSING (CON'T.)

EXAMPLE 2:

```
TWO: PROCEDURE(A) ASSIGN(B);  
      DECLARE A ARRAY(*) SCALAR,  
              B ARRAY(*) BIT(16);
```



} procedure body

```
CLOSE TWO;
```

```
S CALL TWO(P1 ) ASSIGN(P3);  
   1,*  
S CALL TWO(P2 ) ASSIGN(P3);  
   1,*  
CALL TWO(P1) ASSIGN(P6);
```

Illegal - wrong number
of array dimensions

Illegal ~ not arrayed

7-100
3



3

ARRAY PROCESSING (CON'T.)

51
459

I. ARRAYED FUNCTION ARGUMENTS

RULES

- Ⓐ IF A PARAMETER IS ARRAYED, THEN THE CORRESPONDING ARGUMENT MUST BE IDENTICALLY ARRAYED.
- Ⓑ IF THE PARAMETER IS UNARRAYED IT IS POSSIBLE FOR THE ARGUMENT TO HAVE ARRAYNESS -- IN THIS CASE, PROVIDED ALL ARRAYNESSES MATCH UP IN THE REST OF THE STATEMENT, THE FUNCTION WILL BE INVOKED ONCE PER ELEMENTAL EVALUATION.

EXAMPLE 1

```
ALPHA: FUNCTION(R);  
    DECLARE R ARRAY(10);  
    RETURN SUM(R);  
CLOSE;
```

IF THE FUNCTION EXPECTS AN ARRAY THEN ...
IT MUST GET ONE!

```
...  
DECLARE S, T;  
T = ALPHA(S);
```

ILLEGAL -- S IS NOT AN ARRAY(10)

ARRAY PROCESSING (CON'T.)

52
460

EXAMPLE 2

```
ALPHA: FUNCTION(R);  
  DECLARE R;  
  RETURN R**2 COS(R);  
CLOSE;  
...  
DECLARE ARRAY(10), S, T, U;  
DECLARE V;  
...
```

U MUST HAVE SAME ARRAYNESS AS S AND T

- (A) $[S] = [T] + \text{ALPHA}([U]);$
- (B) $[S] = [T] + \text{ALPHA}(V);$

(A) EQUIVALENT TO

```
DO FOR I = 1 TO 10;  
  S$I = T$I + ALPHA(U$I)  
END;
```

(B) EQUIVALENT TO

```
DO FOR I = 1 TO 10;  
  S$I = T$I + ALPHA(V);  
END;
```

RAVELING & UNRAVELING

(NATURAL SEQUENCE)

EXCERPT FROM LANGUAGE SPEC.

There are several kinds of operation in the HAL/S language which require operands with multiple components, array elements, and structure copies to be unraveled into a linear string of data elements. The reverse process of "raveling" a linear string of data elements into components, array elements, and structure copies also occurs. Two major occurrences of these processes are in I/O (see Section 10), and in conversion functions (see Section 6.5).

The standard order in which this unraveling and raveling takes place is called the "natural sequence". By applying the following rules in the order they are stated, the natural sequence of unraveling is obtained. By applying the rules in reverse order, and replacing "unraveled" by "raveled", the natural sequence for raveling is obtained.

RULES FOR MAJOR AND MINOR STRUCTURE:

1. If the operand is a major structure with multiple copies, each copy is unraveled in turn, in order of increasing index. If the operand is a minor structure of a multiple-copy structure, then the copy of the minor structure in each structure copy is unraveled in turn in order of increasing index.
2. The method of unraveling a copy is as follows. Each structure terminal on a "branch" connecting back to the given major or minor structure operand is unraveled in turn. The order taken is the order of appearance of the terminals in the structure template.
3. Each structure terminal is unraveled according to the Rules given below.

RAVELING & UNRAVELING (CON'T.)

example:

```
STRUCTURE A:  
  1 B,  
  2 C SCALAR,  
  2 D VECTOR(3),  
  1 E INTEGER;  
DECLARE A A-STRUCTURE(3);
```

- order of unraveling of B is B_1 , $l=1,2,3$
- order of unraveling of each B_l is C_l, D_l

RULES FOR OTHER OPERANDS:

1. An operand of any type (integer, scalar, vector, matrix, bit, character, or event) may possess arrayness as described in Section 5.4. Each dimension of arrayness, starting from the leftmost is unraveled in turn, in order of increasing index.
2. Integer, scalar, bit, character, and event types are considered for unraveling purposes as having only one data element.
3. Vector types are unraveled component by component, in order of increasing index.
4. Matrix types are unraveled row by row, in order of increasing index. The components of each row are unraveled in turn in order of increasing index.

RAVELING & UNRAVELING (CON'T.)

example:

```
DECLARE V ARRAY(2,2) VECTOR(3);
```

- order of unravelling of V is $V_{i,*}$, $i = 1,2$
- order of unravelling of each $V_{i,*}$ is $V_{i,j,*}$, $j = 1,2$
- order of unravelling of each $V_{i,j,*}$ is $V_{i,j:k}$, $k = 1,2,3$

(standard HAL/S subscript notation used)

RAVELING & UNRAVELING (CON'T.)

IF

STRUCTURE Q:

1 A,
2 I INTEGER,
2 S,
1 B,
2 VI VECTOR,
2 T,

DECLARE Q Q-STRUCTURE(2)

INITIAL(2,4.0,1,2,3,6.7,-2,8.4,4,5,6,9.5);

DECLARE M MATRIX INITIAL(1,0,2,2,0,-1,1,0,3);

DECLARE A ARRAY(7) BOOLEAN

INITIAL(TRUE, FALSE, TRUE, 4#FALSE);

THEN

(Q, M, A) UNRAVELS INTO THE LINEAR STRING:

← copy 1 → ← copy 2 →

2, 4.0, 1, 2, 3, 6.7, -2, 8.4, 4, 5, 6, 9.5,

←row 1→ ←row 2→ ←row 3→

1, 0, 2, 2, 0, -1, 1, 0, 3, TRUE,

FALSE, TRUE, FALSE, FALSE, FALSE,

FALSE

EXPLICIT CONVERSIONS (CON'T.)

A. REVIEW AND EXTENSION OF VECTOR/MATRIX CONVERSIONS

WITH THE VECTOR AND MATRIX CONVERSION FUNCTIONS,
VECTORS AND MATRICES CAN BE DYNAMICALLY MANUFACTURED.

THESE CONVERSION FUNCTIONS ACCEPT AS INPUT A LINEARIZED
LIST OF SCALARS (OR INTEGERS) THAT MAY HAVE BEEN THE
RESULTS OF UNRAVELING OTHER DATA ITEMS. THE CONVERSION
FUNCTIONS THEN SHAPE THIS LINEAR STREAM INTO A VECTOR OR
MATRIX -- THIS IS WHY THESE FUNCTIONS ARE ALSO KNOWN AS

SHAPING FUNCTIONS

EXPLICIT CONVERSIONS (CON'T.)

THE EXPLICIT CONVERSIONS ALLOW FAIRLY GENERALIZED INPUT STREAMS, BUT ARE MORE RESTRICTIVE THAN, SAY, A WRITE STATEMENT. FOR EXAMPLE, THE INPUT STREAM TO AN EXPLICIT CONVERSION FUNCTION MAY NOT CONTAIN A STRUCTURE.

The argument list of a VECTOR or MATRIX conversion may take the following general form:

(exp¹, exp²)

1. Each exp is an expression of any of the following types:

MATRIX	INTEGER
VECTOR	SCALAR

2. Any expression may possess arrayness in the sense described in Section 20.2.
3. The total number of values summed over all expressions must match the length of the vector result, or the product of the row and column dimensions of the result, as appropriate.

EXPLICIT CONVERSIONS (CON'T.)

EXAMPLE (ILLEGAL USAGE)

```
1 N↑ SHAPE:
1 N↑ PROGRAM;
2 N↑   STRUCTURE Q:
2 N↑     1 A,
2 N↑     2 V1 VECTOR,
2 N↑     2 M1 MATRIX,
2 N↑     1 B,
2 N↑     2 V2 VECTOR,
2 N↑     2 M2 MATRIX;
3 N↑   DECLARE Q Q-STRUCTURE(4) INITIAL(3#1, 9#2, 3#3, 9#4);
4 N↑   DECLARE S ARRAY(4) SCALAR DOUBLE INITIAL(10, 11, 12, 13);
5 N↑   DECLARE MN MATRIX(10, 10) DOUBLE;
  C↑
  E↑   *           +
6 N↑   MN = MATRIX ([Q], [S]);
  S↑   @DOUBLE, 10, 10
*** QX1   ERROR #1 OF SEVERITY 1. *****
*** CONVERSION FUNCTIONS MAY NOT HAVE ARGUMENTS OF STRUCTURE TYPE
*** QD1   ERROR #2 OF SEVERITY 1. *****
*** DIMENSIONS OF VECTOR/MATRIX CONVERSION FUNCTION DO NOT AGREE WITH THE NUMBER
*** OF DATA ELEMENTS SUPPLIED IN THE ARGUMENT LIST
7 N↑ CLOSE;
```

EXPLICIT CONVERSIONS (CON'T.)

EXAMPLE (PROPER USAGE)

```
1 N↑ SHAPE:
1 N↑ PROGRAM;
2 N↑   DECLARE VECTOR(3) INITIAL(1, 2, 3),
2 N↑       V1, V2, V3, V4, V5, V6, V7, V8, V9, V10;
3 N↑   DECLARE MATRIX(3, 3) INITIAL(1, 0, 0, 0, 1, 0, 0, 0, 1),
3 N↑       M1, M2, M3, M4, M5;
4 N↑   DECLARE S ARRAY(5, 5) SCALAR DOUBLE INITIAL(3);
5 N↑   DECLARE MM MATRIX(10, 10) DOUBLE;
  C↑
  E↑   *
6 N↑   MM = MATRIX      (V1, M1, [S], V2, M2, V3, M3, V4, M4,
  S↑      @DOUBLE, 10, 10
7 N↑ CLOSE;           V5, M5, V6, V7, V8, V9, V10);
```

WRITTEN AS

MM = MATRIX\$(@DOUBLE, 10, 10)(V1, M1
...)

EXPLICIT CONVERSIONS (CON'T.)

IT IS ALSO POSSIBLE TO MIX IN '#' FORM OF REPETITION:

```
DECLARE V VECTOR INITIAL(1,2,3);
```

```
DECLARE M MATRIX INITIAL(1);
```

```
DECLARE MM MATRIX(4,4);
```

```
...
```

```
MM = MATRIX$(4,4)(V,4#6,M);
```

OR

```
MM = MATRIX$(4,4)(16#0);
```

BOTH WILL RESULT IN SINGLE PRECISION MATRICES.

ALSO,

```
V = VECTOR(6,8,9);
```


MATRIX/VECTOR SHAPING (CONVERSION) FUNCTION DEFAULTS ARE:

VECTOR(3)
MATRIX(3,3) & SINGLE PRECISION

B. INTEGER AND SCALAR CONVERSIONS

SIMPLE FORM

The simple form of the INTEGER and SCALAR conversion functions is:

<p>INTEGER(<i>exp</i>) SCALAR(<i>exp</i>)</p>				
<p>1. <i>exp</i> is an expression of any of the following types:</p> <table><tr><td>BIT STRING (and BOOLEAN)</td><td>INTEGER</td></tr><tr><td>CHARACTER</td><td>SCALAR</td></tr></table>	BIT STRING (and BOOLEAN)	INTEGER	CHARACTER	SCALAR
BIT STRING (and BOOLEAN)	INTEGER			
CHARACTER	SCALAR			
<p>2. <i>exp</i> may possess arrayness, in which case the arrayness must match that of the expression of which the conversion forms a part. The result is to cause an elemental conversion for every elemental evaluation of the outer expression (See Section 20.2).</p>				
<p>3. Conversions to integer or scalar type proceed according to the rules given in Appendix A.</p>				

EXPLICIT CONVERSIONS (CON'T.)

Q. WHAT DOES ALL THAT MEAN?

REMEMBER THAT WE HAVE IMPLICIT CONVERSIONS FOR

SCALAR SINGLE }
INTEGER SINGLE } ⇒ SCALAR DOUBLE
INTEGER DOUBLE }

SCALAR DOUBLE }
INTEGER SINGLE } ⇒ SCALAR SINGLE
INTEGER DOUBLE }

SCALAR SINGLE }
SCALAR DOUBLE } ⇒ INTEGER DOUBLE
INTEGER SINGLE }

SCALAR SINGLE }
SCALAR DOUBLE } ⇒ INTEGER SINGLE
INTEGER DOUBLE }

IN ASSIGNMENT STATEMENTS AND IN INPUT ARGUMENT ⇒ INPUT PARM.

EXPLICIT CONVERSIONS (CON'T.)

FOR ALL OTHER CASES, THE CONVERSION FUNCTION IS AVAILABLE.
ONE CAN, OF COURSE, USE A CONVERSION FUNCTION EVEN WHEN AN
IMPLICIT CONVERSION CAPABILITY EXISTS:

DECLARE S SCALAR DOUBLE,

I INTEGER;

ONE CAN USE:

S = I;

OR

S = SCALAR\$(@DOUBLE)(I);

EQUIVALENT CODE WILL RESULT.

EXPLICIT CONVERSIONS (CON'T.)

EXAMPLES

① DECLARE ARRAY(20), S SCALAR,
T SCALAR, I INTEGER,

...

S = I;
I = S;

CONVERSIONS (ARRAYED DATA)

OR

S = SCALAR(I);
I = INTEGER(S);

EXPLICIT CONVERSIONS (ARRAYED DATA)

AND ALSO,

S = T + I;

OR

S = T + SCALAR(I);

THE OUTPUT WRITER, OF COURSE, WILL SHOW

[S] = [I]

[I] = [S]

[S] = SCALAR([I]);

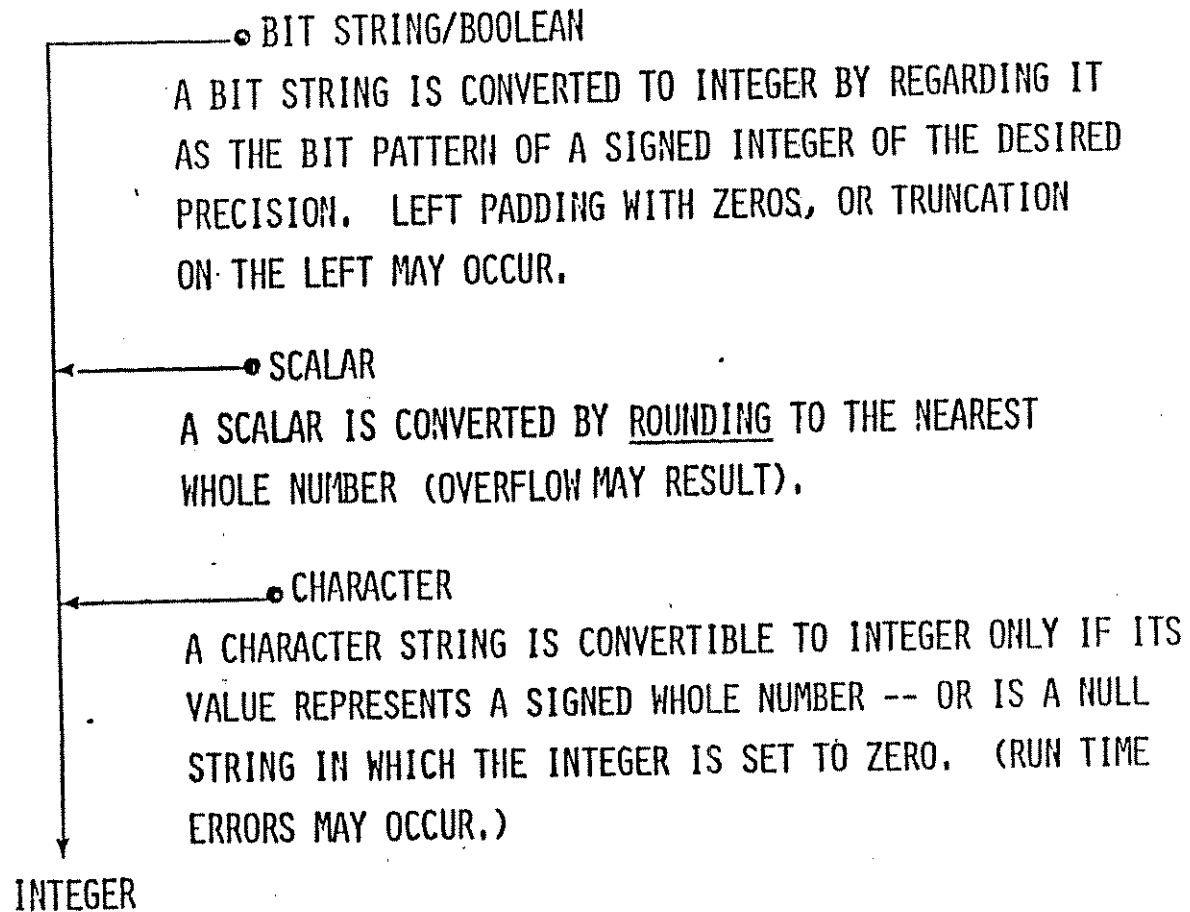
[I] = INTEGER([S]);

[S] = [T] + [I]

[S] = [T] + SCALAR([I]);

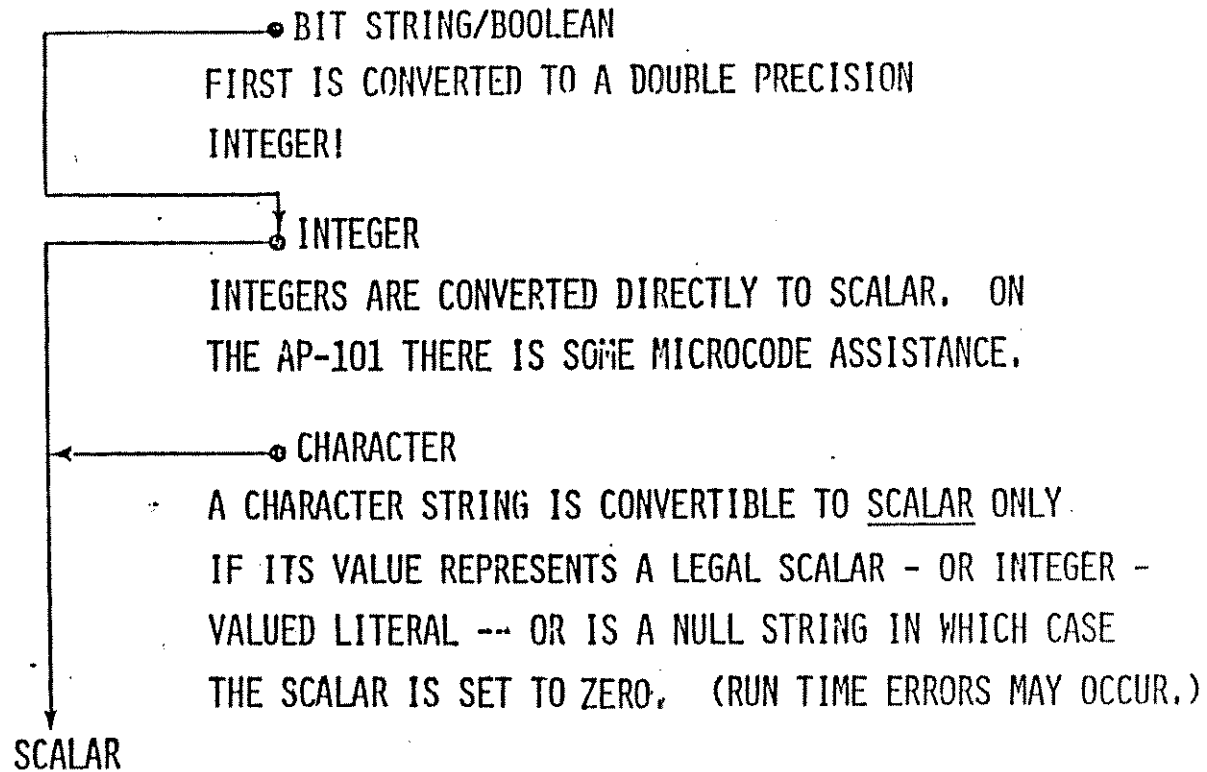
EXPLICIT CONVERSIONS (CON'T.)

CONVERSIONS TO INTEGER:



EXPLICIT CONVERSIONS (CON'T.)

CONVERSION TO SCALAR:



EXPLICIT CONVERSIONS (CON'T.)

FURTHER EXAMPLES (SIMPLE INTEGER & SCALAR CONVERSIONS)

DECLARE I INTEGER,
S SCALAR;

① I = INTEGER('306');

② I = INTEGER('-402');

③ I = INTEGER('ABC');

ILLEGAL

④ S = SCALAR('10');

⑤ S = SCALAR('-6.94E-31');

⑥ S = SCALAR('AB');

ILLEGAL

⑦ I = INTEGER('');
S = SCALAR('');

NULL STRING

I & S WILL BE 0.

EXPLICIT CONVERSIONS (CON'T.)

ALSO, IF

```
    DECLARE B BIT(8) INITIAL(BIN'10110101'),  
           I INTEGER;
```

THEN

```
    I = B;           IS ILLEGAL
```

BUT

```
    I = INTEGER(B);
```

RESULTS IN

```
    I = 181
```


LIST FORM

The list form of the INTEGER and SCALAR conversion functions creates an array result, in addition to type converting the list of expressions constituting its arguments. Its form is as follows:

INTEGER n^1, n^2, \dots (exp^1, exp^2, \dots)
SCALAR n^1, n^2, \dots (exp^1, exp^2, \dots)

1. The subscripts n^i for $i = 1, 2, \dots$ are positive integers specifying the number and size of dimensions of the resulting array. The total number of values summed over all the expressions in the list must be consistent with the number of array elements implied. The upper limit on i is 3*.
2. The subscripts may be omitted entirely, in which case a linear 1-dimensional array is created, whose length is equal to the total number of values summed over all the expressions.

EXPLICIT CONVERSIONS (CON'T.)

3. Each *exp* is an expression of any of the following types:

INTEGER MATRIX
SCALAR BIT STRING (and BOOLEAN)
VECTOR CHARACTER

and may optionally possess arrayness.

4. Conversions to integer or scalar type proceed according to the rules given in Appendix A.

THE LIST FORM THUS ALLOWS CREATION OF INTEGER/SCALAR ARRAYS OF FROM 1 TO 3 DIMENSIONS.

AS IN THE CASE OF THE VECTOR/MATRIX CONVERSION FUNCTIONS, A PRECISION MAY BE SPECIFIED (DEFAULT IS SINGLE) FOR THE INTEGER AND SCALAR CONVERSION FUNCTIONS:

EXAMPLES

```
(1) DECLARE A ARRAY(5) INTEGER
      INITIAL(1,2,3,4,5),
      B ARRAY(5) INTEGER
      INITIAL(6,7,8,9,10);
```

```
INTEGER(A,B)          CREATES AN ARRAY(10)
```

```
≡ (1,2,3,4,5,6,7,8,9,10)
```

```
INTEGER$(@DOUBLE)(A,B)  CREATES AN ARRAY(10) OF DP INTEGERS
```

```
≡ (1,2,3,4,5,6,7,8,9,10)
```

```
INTEGER$(@DOUBLE,10)(A,B) CREATES AN ARRAY(10) OF DP INTEGERS
```

```
≡ (1,2,3,4,5,6,7,8,9,10)
```

```
SCALAR$(@DOUBLE,2,5)(A,B) CREATES A 2-D ARRAY OF DOUBLE PRECISION SCALARS.
```

EXPLICIT CONVERSIONS (CON'T.)

(2) DECLARE B ARRAY(3,4);
B = SCALAR\$(3,4)(4#1, 4#2, 4#3);

CREATES $\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{bmatrix}$

(3) DECLARE C ARRAY(4,4,4) DOUBLE;
C = SCALAR\$(@DOUBLE,4,4,4)
(4#6,15#9,27#3,18#0);

(4) DECLARE V VECTOR INITIAL(1,2,3);
DECLARE B BIT(8) INITIAL(BIN(8)'1');
DECLARE M MATRIX INITIAL(0);

SCALAR\$(@DOUBLE)(V,B,M) CREATES AN ARRAY(13) OF DOUBLE
PRECISION SCALARS

= (1,2,3,255,0,0,0,0,0,0,0,0,0)

EXPLICIT CONVERSIONS (CON'T.)

READALL(5) CARD1;

T = SCALAR @DOUBLE (CARD1 1 TO 20);

R = VECTOR @DOUBLE (SCALAR @DOUBLE (CARD1 21 TO 40, CARD1 41 TO 60, CARD1 61 TO 80));

READALL(5) CARD1;

V = VECTOR @DOUBLE (SCALAR @DOUBLE (CARD1 1 TO 20, CARD1 21 TO 40, CARD1 41 TO 60));

EPHGMT = SCALAR @DOUBLE (CARD1 61 TO 80);

IF EPHGMT = 0 THEN

DO;

READALL(5) CARD1;

READALL(5) CARD2;

READALL(5) C;

* RNP = MATRIX @DOUBLE (SCALAR @DOUBLE (CARD1 1 TO 20, CARD1 21 TO 40, CARD1 41 TO 60, CARD1 61 TO 80, CARD2 1 TO 20, CARD2 21 TO 40, CARD2 41 TO 60, CARD2 61 TO 80, C));

* RNP = RNP * T;

END;

EXPLICIT CONVERSIONS (CON'T.)

SIMPLE FORM

THE SIMPLE FORM OF BIT CONVERSION IS AS FOLLOWS:

**COMPONENT
SUBSCRIPT**

BIT *subscript* (*exp*)

- exp* is an expression of any of the following types:

INTEGER	BIT STRING (and BOOLEAN)
SCALAR	CHARACTER
- exp* may possess arrayness in which case the arrayness must match that of the expression of which the conversion forms a part. The result is to cause an elemental conversion for every elemental evaluation of the outer expression (see Section 20.2).
- Conversion to bit string type proceeds according to the rules given in Appendix A. The result is always a 32-bit string*.
- subscript* represents component subscripting on the result of the conversion. It possesses the same forms as component subscripting on bit string data items as described in Section 17.3.
- If *subscript* is absent, the result of the function is the entire bit string generated by the conversion.

83-24

EXPLICIT CONVERSIONS (CON'T.)

SINCE THERE ARE NO IMPLICIT CONVERSIONS FROM INTEGER OR SCALAR TO BIT STRINGS, THE BIT CONVERSION FUNCTION IS IMPORTANT.

EXAMPLES: (SIMPLE BIT CONVERSION)

If I is a halfword integer with $I \equiv 5$

then $\text{BIT}(I) \equiv 00000005_{16}$

If C is a character data item with $C = '10110011101'$

then $\text{BIT}(C) \equiv 0000000000000000000010110011101_2$

$\text{BIT}_{17 \text{ TO } 32}(C) \equiv 0000010110011101_2$

and $\text{BIT}_{28 \text{ TO } 32}(C) = 11101_2$

EXPLICIT CONVERSIONS (CON'T.)

RADIX FORM (MOSTLY USEFUL IN I/O)

The radix form of BIT conversion is used when a character value is to be converted by an explicit rule to a bit string. A radix specifying the conversion rule is supplied in place of a subscript. The possible forms are as follows:

BIT_{@BIN}(*exp*)

BIT_{@OCT}(*exp*)

BIT_{@DEC}(*exp*)

BIT_{@HEX}(*exp*)

1. *exp* is an expression of character type whose value must consist entirely of a string of digits legal for the specified radix.
2. The radices have the following meanings:

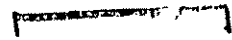
radix	digit string
@BIN	binary
@OCT	octal
@DEC	decimal
@HEX	hexadecimal

3. *exp* may possess arrayness with the same implications as in the simple form of BIT conversion.
4. The conversion generates the binary representation of the input digit string. The binary representation is truncated or padded with binary zeroes on the left to create a 32-bit string*.

EXPLICIT CONVERSIONS (CON'T.)

EXAMPLES

ALL 32-BIT



BIT@HEX('FA0') ≡ 0000FA0₁₆

BIT@DEC('1024') ≡ 0000400₁₆

BIT@OCT('17777') ≡ 0000FFF₁₆

BIT@HEX('F0F1F2F3F4') ≡ F1F2F3F4₁₆

↑
TRUCATION ON THE LEFT

EXPLICIT CONVERSIONS (CON'T.)

SIMPLE FORM

The simple form of CHARACTER conversion is as follows:

CHARACTER_{subscript}^(exp)

1. *exp* is an expression of any of the following types:
INTEGER BIT STRING (and BOOLEAN)
SCALAR CHARACTER
2. *exp* may possess arrayness, with the same implications as in the BIT conversion function. (See Section 21.3).
3. Conversion to character type proceeds according to the rules given in Appendix A. The length of the result of conversion depends on the type of the input data.
4. *subscript* represents component subscripting on the result of the conversion. It possesses the same forms as component subscripting on character data items as described in Section 6.1.
5. If *subscript* is absent, then the result of the function is the entire string of characters generated by the conversion.

EXPLICIT CONVERSIONS (CON'T.)

EXAMPLES

If I is a halfword integer with $I = 173$

then $\text{CHARACTER}(I) \equiv '173'$

$\text{CHARACTER}_{1 \text{ TO } 2}(I) \equiv '17'$

$\text{CHARACTER}_{1 \text{ TO } 5}(I) \equiv '173'$

If B is a bit string of length 4 with

$B \equiv 0101_2$

then

$\text{CHARACTER}(B) \equiv '101'$

(note removal of leading zeroes)

NOTE THAT THE LENGTH OF THE RESULTANT CHARACTER STRING
IS DATA DEPENDENT, WHEREAS FOR THE BIT CONVERSION
FUNCTION IT IS ALWAYS 32 BITS.

EXPLICIT CONVERSIONS (CON'T.)

RADIX FORM

The radix form of CHARACTER conversion is used when a bit string value is to be converted by an explicit rule to a character string. Analogous to the radix form of BIT function, a radix specifying the conversion rule is supplied in place of a subscript. The possible forms are as follows:

CHARACTER_{@BIN}(exp)

CHARACTER_{@OCT}(exp)

CHARACTER_{@DEC}(exp)

CHARACTER_{@HEX}(exp)

1. exp is an expression of bit string type, and possibly possessing arrayness, with the same implications as in the BIT conversion function.
2. The value of the bit string is converted to a string of digits as specified by the radix, removing leading zeroes.
3. The radices have the following meanings:

radix	digit string
@BIN	binary
@OCT	octal
@DEC	decimal
@HEX	hexadecimal

4. The length of the resulting string varies depending on the value of exp.

EXPLICIT CONVERSIONS (CON'T.)

EXAMPLES:

CHARACTER_{@BIN}(BIN'001010') = '001010'

CHARACTER_{@OCT}(BIN'001010') = '12'

CHARACTER_{@DEC}(BIN'001010') = '10'

CHARACTER_{@HEX}(BIN'001010') = '0A'

PRIME NUMBER GENERATOR

```
//CWS1840T JOB 7481, SCHULENBERG, C, TIME=1, PRTY=1, REGION=350K,  
//      NOTIFY=CWS1840  
//HAL      EXEC HALSCLD, ACCT=NOTIFY, OPTION='LIST'  
//HAL. SYSPRINT DD SYSOUT=2  
//HAL. SYSIN DD *  
ERATOSTHENES: PROGRAM;  
  REPLACE N BY "5000";  
  DECLARE INTEGER, I, J, COUNT INITIAL(1);  
  DECLARE SIEVE ARRAY(N) INTEGER;  
  DO FOR I = 2 TO N;  
    SIEVE(I) = 1;  
  END;  
  DO FOR I = 2 TO N;  
    IF SIEVE(I) \= 0 THEN DO;  
      WRITE (6) COUNT, SIEVE(I);  
      COUNT = COUNT + 1;  
      DO FOR J = 2 I TO N BY I;  
        SIEVE(J) = 0;  
      END;  
    END;  
  END;  
CLOSE ERATOSTHENES;  
/*  
//GO. CHANNEL6 DD SYSOUT=2  
//GO. REQUESTS DD *  
  EXECUTE TEMPNAME;  
  AT END: PROFILE;  
/*
```


HAL/S COMPILER PHASE 1 -- VERSION 13.12 OF SEPTEMBER 14, 1975. CLO
TODAY IS OCTOBER 11, 1975. CLOCK TIME = 13:31:2.16.

PARN FIELD: LIST

COMPLETE LIST OF COMPILE-TIME OPTIONS IN EFFECT

*** TYPE 1 OPTIONS ***

NODUMP INSTEAD OF DUMP
NOLISTING2 INSTEAD OF LISTING2
LIST INSTEAD OF NOLIST
TRACE INSTEAD OF NOTRACE
NODECK INSTEAD OF DECK
TABLES INSTEAD OF NOTABLES
NOTABLST INSTEAD OF TABLST
NOADDRS INSTEAD OF ADDRS
NOSRN INSTEAD OF SRN
NOSDL INSTEAD OF SDL
NOTABDMP INSTEAD OF TABDMP
ZCON INSTEAD OF NOZCON
NOFCDATA INSTEAD OF FCDATA

*** TYPE 2 OPTIONS ***

TITLE =
LINECT = 59
PAGES = 250
SYMBOLS = 200
MACROSIZE = 500
LITSTRINGS = 2000
COMPUNIT = 0
XREFSIZE = 2000
CARDTYPE =
LABELSIZE = 1200

HAL/S COMPILATION

INTERNETRICS, INC.

STMT

SOURCE

```

1 M↑ ERATOSTHENES:
1 M↑ PROGRAM;
2 M↑   REPLACE N BY "5000";
3 M↑   DECLARE INTEGER,
3 M↑     I, J,
3 M↑     COUNT INITIAL(1);
4 M↑   DECLARE SIEVE ARRAY(N) INTEGER;
5 M↑   DO FOR I = 2 TO N;
6 M↑     SIEVE = I;
7 M↑   END;
8 M↑   DO FOR I = 2 TO N;
9 M↑     IF SIEVE \= 0 THEN
10 M↑       DO;
11 M↑         WRITE(6) COUNT, SIEVE ;
12 M↑         COUNT = COUNT + 1;
13 M↑         DO FOR J = 2 I TO N BY I;
14 M↑           SIEVE = 0;
15 M↑         END;
16 M↑       END;
17 M↑     END;
18 M↑ CLOSE ERATOSTHENES;

```

*** COMPILATION LAYOUT ***

ERATOSTHENES: PROGRAM;
HAL/S COMPILATION

INTERMETRICS, INC.

OCTOBER 11, 1975

SYMBOL & CROSS REFERENCE TABLE LISTING:

(CROSS REFERENCE FLAG KEY: 4 = ASSIGNMENT, 2 = REFERENCE, 1 = SUBSCRIPT USE, 0 = DEFINITION)

DCL	NAME	TYPE	ATTRIBUTES & CROSS REFERENCE						
3	COUNT	INTEGER	SINGLE, ALIGNED, STATIC, INITIAL	XREF: 0 0003	2 0011	6 0012			
1	ERATOSTHENES	PROGRAM	XREF: 0 0001						
3	I	INTEGER	SINGLE, ALIGNED, STATIC	XREF: 0 0003	4 0005	3 0006	4 0008	1 0009	
			1 0011 2 0013						
3	J	INTEGER	SINGLE, ALIGNED, STATIC	XREF: 0 0003	4 0013	1 0014			
2	N	REPLACE MACRO	MACRO-TEXT INDEX=1	XREF: 0 0002	2 0004	2 0005	2 0008	2 0013	
4	SIEVE	INTEGER ARRAY	ARRAY(5000), SINGLE, ALIGNED, STATIC	XREF: 0 0004	4 0006	2 0009			
			2 0011 4 0014						

MACRO TEXT LISTING:

LOC TEXT

1 5000

HAL/S COMPILATION

INTERMETRICS, INC.

OCTOBER 11, 1975

NAME	OPTIONAL TABLE SIZES	
	REQUESTED	USED
++++	+++++	++++
SYMBOLS	200	6
MACROSIZE	500	5
LITSTRINGS	2000	0
XREFSIZE	2000	24

8-56

CALLS TO SCAN = 107
CALLS TO IDENTIFY = 28
NUMBER OF REDUCTIONS = 300
MAX STACK SIZE = 13
MAX IND. STACK SIZE = 6
END IND. STACK SIZE = 1
END ARRAY STACK SIZE = 0
MAX EXT+ARRAY INDEX = 3
STATEMENT COUNT = 18
MINOR COMPACTIFIES = 0
MAJOR COMPACTIFIES = 0
MAX NESTING DEPTH = 1
FREE STRING AREA = 59643

END OF HAL/S PHASE 1, OCTOBER 11, 1975. CLOCK TIME = 13:31:4.63.

17 CARDS WERE PROCESSED.
NO ERRORS WERE DETECTED DURING PHASE 1 .

TOTAL CPU TIME FOR PHASE 1 0:0:0.69.
CPU TIME FOR PHASE 1 SET UP 0:0:0.09.
CPU TIME FOR PHASE 1 PROCESSING 0:0:0.54.
CPU TIME FOR PHASE 1 CLEAN UP 0:0:0.06.
PROCESSING RATE: 1800 CARDS PER MINUTE.

***** TEMPLATE LIBRARY MEMBER @ERATOS NOT FOUND - ADDED , VERSION=1
HAL/S COMPILATION I N T E R M E T R I C S , I N C .

HAL/S COMPILER PHASE 2 -- VERSION 360-13.11 OF SEPTEMBER 15, 1975.

HAL/S PHASE 2 ENTERED OCTOBER 11, 1975. CLOCK TIME = 13:31:8.01

HAL/S COMPILATION I N T E R M E T R I C S , I N C .

ESDID	NAME	TYPE	LENGTH	BLOCK NAME
0001	@ERATOS	0000	000126	ERATOSTHENES
0002	#ERATOS	0000	000048	
0003	#TERATOS	0000	000048	
0004	#DERATOS	0000	002716	
0005	@ERATOS	0002		
0006	IOINIT	0002		
0007	IOUT	0002		

HAL/S COMPILATION

INTERMETRICS, INC.

LOCCTR	CODE	LABEL	INSN	OPERANDS	SYMBOLIC OPERAND
000000		ST#1	EQU	* TIME = 143	
000000		#0ERATOS	CSECT	ESDID= 0001	
000000		ERATOSTH	EQU	*	ERATOSTHENES
000000	47F0F018		BC	15,24(0,15)	
000004	00000120		DC	A'00000120'	#FERATOS
000008	0058		DC	X'0058'	
00000A	0CC5D9C1		DC	X'0CC5D9C1'	
00000E	E3D6E2E3		DC	X'E3D6E2E3'	
000012	C8C5D5C5		DC	X'C8C5D5C5'	
000016	E2		DC	X'E2'	
000017	00		DC	X'00'	
000018	5000F004		L	11,4(0,15)	
00001C	906A0020		LM	6,10,40(11)	
000020	9201D000		MVI	0(13),1	
000024	05EB		BALR	14,11	
000026	0001		DC	X'0001'	
000028		ST#2	EQU	* TIME = 0	
000028		ST#3	EQU	* TIME = 0	
000004		#0ERATOS	CSECT	ESDID= 0004	
000004	0001		DC	X'0001'	
000006		ST#4	EQU	* TIME = 0	
000006		ST#5	EQU	* TIME = 98	
000028		#0ERATOS	CSECT	ESDID= 0001	
000028	41900002		LA	9,2(0,0)	
00002C		LBL#2	EQU	*	
00002C	4090A000		STH	9,0(0,10)	I
000030	05EB		BALR	14,11	
000032	0005		DC	X'0005'	
000034	4990B046		CH	9,70(0,11)	H'5000'
000038	472F0056		BC	2,86(15,0)	000056 LBL#3
00003C		ST#6	EQU	* TIME = 54	
00003C	1A99		RR	9,9	
00003E	4020A000		LH	2,0(0,10)	I
000042	4029A004		STH	2,4(9,10)	SIEVE
000046	05EB		BALR	14,11	
000048	0006		DC	X'0006'	
00004A		ST#7	EQU	* TIME = 50	
00004A		LBL#4	EQU	*	
00004A	41900001		LA	9,1(0,0)	
00004E	4A90A000		AH	9,0(0,10)	I
000052	47FF002C		BC	15,44(15,0)	00002C LBL#2
000056		LBL#3	EQU	*	
000056	05EB		BALR	14,11	
000058	0007		DC	X'0007'	

```

00005A          ST#8      EQU  *      TIME = 98
00005A 41900002    LA   9,2(0,0)
00005E          LBL#5     EQU  *
00005E 4090A000    STH  9,0(0,10)      I
000062 05EB       BALR 14,11
000064 0008       DC   X'0008'
000066 4990B046    CH   9,70(0,11)      H'5000'
00006A 472F011A    BC   2,292(15,0)     00011A LBL#6
00006E          ST#9     EQU  *      TIME = 72
00006E 05EB       BALR 14,11
000070 0009       DC   X'0009'

```

HAL/S COMPILATION

I N T E R M E T R I C S , I N C .

LOCCTR	CODE	LABEL	INSN	OPERANDS	SYMBOLIC OPERAND
000072	1A99		AR	9,9	
000074	4029A004		LH	2,4(9,10)	SIEVE
000078	1222		LTR	2,2	
00007A	478F010E		BC	0,270(15,0)	00010E LBL#7
00007E		ST#10	EQU	* TIME = 0	
00007E	05EB		BALR	14,11	
000080	000A		DC	X'000A'	
000082		ST#11	EQU	* TIME = 105	
000082	41100006		LA	1,6(0,0)	
000086	41000003		LA	0,3(0,0)	
00008A	05EC		BALR	14,12	
00008C	00000000		DC	A'00000000'	IOINIT
000090	4800A004		LH	0,4(0,10)	COUNT
000094	05EC		BALR	14,12	
000096	00000000		DC	A'00000000'	IOUT
00009A	1802		LR	0,2	
00009C	05EC		BALR	14,12	
00009E	00000000		DC	A'00000000'	IOUT
0000A2	05EB		BALR	14,11	
0000A4	000B		DC	X'000B'	
0000A6		ST#12	EQU	* TIME = 61	
0000A6	4030A004		LH	3,4(0,10)	COUNT
0000AA	4A30B044		AH	3,60(0,11)	H'1'
0000AE	4030A004		STH	3,4(0,10)	COUNT
0000B2	05EB		BALR	14,11	
0000B4	000C		DC	X'000C'	

0000B6	ST#13	EQU	*	TIME = 335	
0000B6 41400002		LA	4,2(0,0)		
0000BA 4C40A000		MH	4,0(0,10)		I
0000BE 4850A000		LH	5,0(0,10)		I
0000C2 4050D050		STH	5,80(0,13)		
0000C6	LBL#8	EQU	*		
0000C6 4040A002		STH	4,2(0,10)		J
0000CA 05EB		BALR	14,11		
0000CC 0000		DC	X'0000'		
0000CE 412F0106		LA	2,262(15,0)		000106 LBL#9
0000D2 9100D050		TM	80(13),128		
0000D6 05E0		BALR	14,0		
0000D8 4780E010		BC	8,16(0,14)		0000E8 **16
0000DC 4940B046		CH	4,70(0,11)		H'5000'
0000E0 0742		BCR	4,2		
0000E2 05E0		BALR	14,0		
0000E4 47F0E00A		BC	15,10(0,14)		0000EE **10
0000E8 4940B046		CH	4,70(0,11)		H'5000'
0000EC 0722		BCR	2,2		
0000EE	ST#14	EQU	*	TIME = 47	
0000EE 1A44		AR	4,4		
0000F0 1B22		SR	2,2		
0000F2 4024A004		STH	2,4(4,10)		SIEVE
0000F6 05EB		BALR	14,11		
0000F8 000E		DC	X'000E'		
0000FA	ST#15	EQU	*	TIME = 52	
0000FA	LBL#10	EQU	*		
000FA 4840D050		LH	4,80(0,13)		
0000FE 4A40A002		AR	4,2(0,10)		J
HAL/S COMPILATION					I N T E R M E T R I C S , I N C .

LOCCTR	CODE	LABEL	INSH	OPERANDS	SYMBOLIC OPERAND
000102	47FF00C6		BC	15, 198(15, 0)	0000C6 LBL#8
000106		LBL#9	EQU	*	
000106	05EB		BALR	14, 11	
000108	000F		DC	X'000F'	
00010A		ST#16	EQU	* TIME = 0	
00010A	05EB		BALR	14, 11	
00010C	0010		DC	X'0010'	
00010E		ST#17	EQU	* TIME = 50	
00010E		LBL#7	EQU	*	
00010E		LBL#11	EQU	*	
00010E	41900001		LA	9, 1(0, 0)	
000112	4A90A000		AH	9, 0(0, 10)	
000116	47FF005E		BC	15, 94(15, 0)	I 00005E LBL#5
00011A		LBL#6	EQU	*	
00011A	05EB		BALR	14, 11	
00011C	0011		DC	X'0011'	
00011E		ST#18	EQU	* TIME = 18	
00011E	05EB		BALR	14, 11	
000120	0012		DC	X'0012'	
000122	47F0C004		BC	15, 4(0, 12)	

000128	47F0C174	BC	15,372(0,12)	STRACE
00012C	00000000	DC	'00000000'	#DERATOS
000130	00000170	DC	'00000170'	#TERATOS
000134	FF00016C	DC	'FF00016C'	#TERATOS
000138	00000000	DC	'00000000'	
00013C	0001	DC	'X'0001'	
00013E	0012	DC	'X'0012'	
000140	00000000	DC	'X'00000000'	
000144	00000000	DC	'X'00000000'	
000148	00000000	DC	'X'00000000'	
00014C	00000000	DC	'X'00000000'	
000150	00000000	DC	'A'00000000'	#DERATOS
000154	00000000	DC	'A'00000000'	#DERATOS
000158	00000000	DC	'A'00000000'	#DERATOS
00015C	00000000	DC	'A'00000000'	#DERATOS
000160	00000000	DC	'A'00000000'	#DERATOS
000164	00000001	DC	'X'00000001'	
000168	00000000	DC	'A'00000000'	#DERATOS
00016C	0001	DC	'X'0001'	#DERATOS
00016E	1388	DC	'X'1388'	#DERATOS

ESD10= 0003

END

HAL/S COMPILATION

I N T E R M E T R I C S , I N C .

RLD POS REF FLAG ADDRESS

0001	0002	08	000005
0001	0006	08	000080
0001	0007	08	000097
0001	0007	08	00009F
0002	0004	08	000120
0002	0003	08	000131
0002	0003	08	000135
0002	0001	08	000139
0002	0004	08	000151
0002	0004	08	000155
0002	0004	08	000159
0002	0004	08	000150
0002	0004	08	000161
0002	0005	08	000169

```

LOC      B DISP  NAME
          UNDER ERATOSTHENES   STACK=80
000000  10 000  I
000002  10 002  J
000004  10 004  COUNT
000006  10 004  SIEVE

```

INSTRUCTION FREQUENCIES

```

INSN  COUNT
BALR  5
BCR   2
LTR   1
LR    1
AR    3
SR    1
STH   7
LA    8
BC    11
LH    6
CH    4
AH    4
MH    1
L     1
TM    1
MVI   1
LM    1

```

48 HALMAT OPERATORS CONVERTED

58 INSTRUCTIONS GENERATED

368 BYTES OF PROGRAM, 10006 BYTES OF DATA

```

MAX. OPERAND STACK SIZE      =9
END OPERAND STACK SIZE      =0
HAL/S COMPILATION           INTERMETRICS, INC.

```

```

NUMBER OF STATEMENT LABELS USED =11
MAX. STORAGE DESCRIPTOR STACK SIZE =1
END STORAGE DESCRIPTOR STACK SIZE =0
NUMBER OF MINOR COMPACTIFIES     =0
NUMBER OF MAJOR COMPACTIFIES     =0
FREE STRING AREA                  =54356

```

END OF HAL/S PHASE 2 OCTOBER 11, 1975. CLOCK TIME = 13:31:09.00
 TOTAL CPU TIME FOR PHASE 2 0:0:0.49
 CPU TIME FOR PHASE 2 SET UP 0:0:0.04
 CPU TIME FOR PHASE 2 GENERATION 0:0:0.10
 CPU TIME FOR PHASE 2 CLEAN UP 0:0:0.35
 HAL/S COMPILATION I N T E R N E T R I C S , I N C .

HAL/S COMPILER PHASE 3 -- VERSION 15.0 OF SEPTEMBER 21, 1975.

HAL/S PHASE 3 ENTERED OCTOBER 11, 1975. CLOCK TIME = 13:31:10.66

SIMULATION DATA FILE #HERATOS HAS BEEN CREATED

PAGING AREA SIZE (PAGES) = 97
 NUMBER OF LOCATES = 201

PREDICTED SDF SIZE (PAGES) = 1
 ACTUAL SDF SIZE (PAGES) = 1

DIRECTORY FREE SPACE (BYTES) = 72
 DATA FREE SPACE (BYTES) = 920
 SDF SIZE (BYTES) = 688
 SDF DENSITY (%) = 40
 NUMBER OF BLOCK NODES = 1
 NUMBER OF SYMBOL NODES = 5
 NUMBER OF STATEMENT NODES = 18

NUMBER OF BLOCKS DELETED = 0
 NUMBER OF SYMBOLS DELETED = 1
 NUMBER OF TEMPLATES DELETED = 0
 NUMBER OF MINOR COMPACTIFIES = 0
 NUMBER OF MAJOR COMPACTIFIES = 0

END OF HAL/S PHASE 3 OCTOBER 11, 1975. CLOCK TIME = 13:31:11.61
 TOTAL CPU TIME IN PHASE 3 0:0:0.18
 CPU TIME FOR PHASE 3 INITIALIZATION 0:0:0.02
 CPU TIME FOR PHASE 3 FILE GENERATION 0:0:0.05
 CPU TIME FOR PHASE 3 FILE EMISSION 0:0:0.11

8-44

HAL/S-360 V13.0 START TIME: 13:31:46.60 DAY: 75/284

1	2
2	3
3	5
4	7
5	11
6	13
7	17
8	19
9	23
10	29
11	31
12	37
13	41
14	43
15	47
16	53
17	59
18	61
19	67
20	71
21	73
22	79
23	83
24	89
25	97
26	101
27	103
28	107
29	109
30	113
31	127
32	131
33	137
34	139
35	149
36	151
37	157
38	163
39	167
40	173
41	179
42	181
43	191
44	193
45	197
0	0
0	0

7-45

647	4801
648	4813
649	4817
650	4831
651	4861
652	4871
653	4877
654	4889
655	4903
656	4909
657	4919
658	4931
659	4933
660	4937
661	4943
662	4951
663	4957
664	4967
665	4969
666	4973
667	4987
668	4993
669	4999

*** AT END

STATEMENT PROFILE FOR PROGRAM ERATOSTHENES

STATEMENT#	USE AND % OF TOTAL	TIME	COST AND % OF TOTAL
1	1 0.00	14	14 0.00
2 TO	4 NOT EXECUTED		
5	16 0.03	10	160 0.02
6	4,999 10.83	5	24,995 3.97
7	1 0.00	5	5 0.00
8	5,000 10.83	10	50,000 7.94
9	4,999 10.83	7	34,993 5.56
10	669 1.44		0 0.00
11	669 1.44	11	7,359 1.16
12	669 1.44	6	4,014 0.63
13	11,738 25.43	34	399,092 63.43
14	11,869 23.98	5	55,345 8.79
15	669 1.44	5	3,345 0.53
16	669 1.44		0 0.00
17	1 0.00	5	5 0.00
18	1 0.00	2	2 0.00

COMPILATION UNIT USE SUMMARY

STATEMENT#	USE AND % OF TOTAL	TIME	COST AND % OF TOTAL
	46,154 100.00		629,169 100.00
	46,154 100.00		629,169 100.00

5-46

DATA STORAGE AND ACCESS

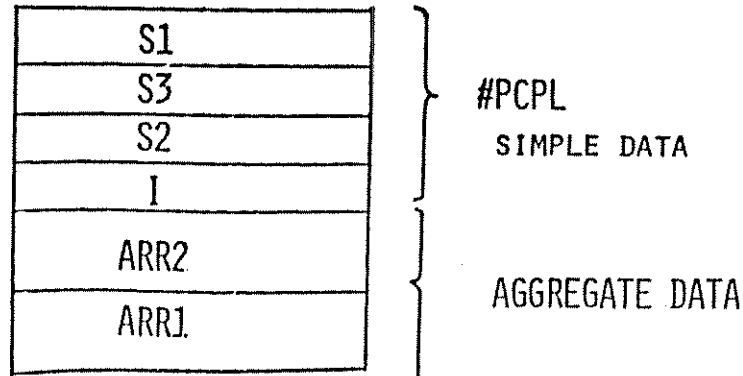
S1
461

- DATA IS NOT NECESSARILY PHYSICALLY ALLOCATED IN THE ORDER IN WHICH THE USER MAKES THE DECLARES.

EXAMPLE

```
CPL: COMPOOL;  
  DECLARE ARR1 ARRAY(3,6) SCALAR;  
  DECLARE ARR2 ARRAY(20) INTEGER;  
  DECLARE I INTEGER;  
  DECLARE S1 SCALAR DOUBLE;  
  DECLARE S2 SCALAR;  
  DECLARE S3 SCALAR;  
CLOSE CPL;
```

WOULD PROBABLY BE ALLOCATED AS:



DATA STORAGE AND ACCESS (CONTINUED)

S2
462

GENERAL ALLOCATION RULES:

- 1) HAL REORDERS DATA IN AN ATTEMPT TO PROVIDE ADDRESSING COVERAGE FOR THE DATA USING A MINIMUM OF DISTINCT BASE REGISTER VALUES.
- 2) SIMPLE DATA (OFFSET = 0) COMES BEFORE AGGREGATE DATA (OFFSET \neq 0).
SIMPLE DATA: INTEGER, SCALAR, STRINGS (BIT & CHARACTER).
AGGREGATE DATA: ARRAYS, VECTORS, MATRICES AND STRUCTURES.
- 3) WITHIN EACH OF THE TWO GROUPS, DATA IS ORDERED SUCH THAT ITEMS REQUIRING THE SAME BOUNDARY ALIGNMENTS ARE ADJACENT -- THUS MINIMIZING WASTED SPACE.
- 4) WITHIN THE AGGREGATE GROUP SINGLE DIMENSIONAL ARRAYS COME BEFORE MULTI-DIMENSIONAL ARRAYS.

DATA STORAGE AND ACCESS (CONTINUED)

53
463

- 5) STRUCTURE TEMPLATES ARE INTERNALLY ORDERED SUCH THAT THE MINIMUM BOUNDARY ALIGNMENT WITHIN ANY NODE LEVEL IS REQUIRED. (MORE ON THIS LATER.)

GENERALLY, THEN, DATA IS NOT ALLOCATED IN THE DECLARED ORDER EXCEPT IN SOME OBVIOUS CASES, E.G.

DECLARE VECTOR DOUBLE, V1, V2, V3, V4, V5, V6, V7, V8, V9;
IN SUCH A CASE THE COMPILER WILL NOT ALTER THE ORDER SINCE NOTHING WOULD BE GAINED.

NONETHELESS, WHEN THE HAL/S COMPILER IS GIVEN FREE REIN TO REORDER DATA, USERS SHOULD NEVER TRY TO PREDICT THE RESULTING ORDER, NOR ASSUME THAT THE ORDER GENERATED WILL NOT CHANGE WHEN COMPILER IMPROVEMENTS ARE MADE.

DATA STORAGE AND ACCESS (CON'T.)

32
464

WHEN USERS MUST CONTROL THE PHYSICAL LAYOUT OF DATA (E.G. FOR INTERFACING TO EXTERNALLY DEFINED DATA LAYOUTS), THE RIGID KEYWORD MAY BE USED.

RIGID TELLS THE COMPILER TO ALLOCATE DATA IN THE DECLARED ORDER. IT MAY BE SPECIFIED ONLY ON COMPOOL BLOCK HEADERS, AND ON STRUCTURE TEMPLATES.

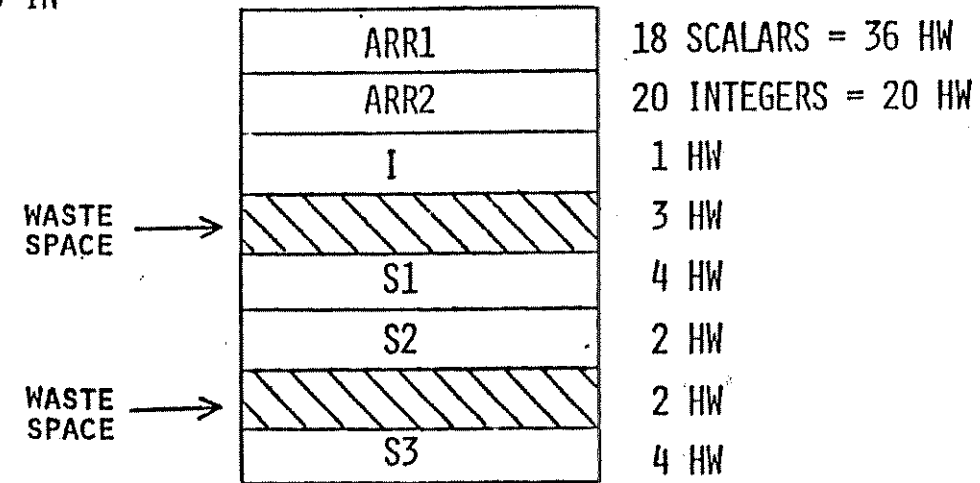
EXAMPLE 1:

```
CPL: COMPOOL RIGID;
      DECLARE ARR1 ARRAY(3,6) SCALAR;
      DECLARE ARR2 ARRAY(20) INTEGER;
      DECLARE I INTEGER;
      DECLARE S1 SCALAR DOUBLE;
      DECLARE S2 SCALAR;
      DECLARE S3 SCALAR DOUBLE;
CLOSE CPL;
```

53
465

DATA STORAGE AND ACCESS (CONTINUED)

RESULTS IN



NOTES

- 1) UNLESS USERS ARE CAREFUL, RIGID CAN CAUSE WASTED SPACE.
- 2) ON THE FC, DOUBLE PRECISION SCALARS ARE ALLOCATED ON DOUBLE-WORD BOUNDARIES EVEN THOUGH THE HARDWARE (AP-101) DOES NOT REQUIRE IT.
WHY? FOR STORAGE LAYOUT COMPATIBILITY WITH THE 360.

DATA STORAGE AND ACCESS (CONTINUED)

52
466

EXAMPLE 2:

```
STRUCTURE Q RIGID;  
  1 S1 SCALAR,  
  1 I1 INTEGER,  
  1 B1 BIT(16),  
  1 S2 SCALAR DOUBLE,  
  1 V VECTOR;  
DECLARE Q Q-STRUCTURE
```

RESULTS IN:

DOUBLE WORD
BOUNDARY



S1
I1
B1
S2
V\$1
V\$2
V\$3

HW
2
1
1
4
2
2
2

DATA STORAGE AND ACCESS (CONTINUED)

31
467

NOTES

- 1) USE OF RIGID ON STRUCTURE TEMPLATES IS ESPECIALLY LIKELY TO CAUSE WASTE OF MEMORY UNLESS CARE IS TAKEN.
- 2) RIGID MUST BE SPECIFIED ON THE TEMPLATE DECLARATION -- OR A MINOR STRUCTURE THEREOF. RIGID MAY NOT BE APPENDED TO A STRUCTURE DECLARATION.
- 3) WHETHER A STRUCTURE TEMPLATE IS RIGID OR NOT, THE BOUNDARY ALIGNMENT SELECTED FOR THE TEMPLATE IS DETERMINED BY THE TERMINAL WITH THE MOST RESTRICTIVE BOUNDARY ALIGNMENT REQUIREMENTS.
- 4) MINOR STRUCTURES ARE ALWAYS ALLOCATED AS SEPARATE, DISCRETE UNITS WITHIN THE OVERALL STRUCTURE.
- 5) THE KEYWORD "RIGID" ON A COMPOOL BLOCK HEADER DOES NOT GUARANTEE STRUCTURE RIGIDITY -- ONLY WHEN RIGID IS APPLIED TO A TEMPLATE DOES IT PENETRATE.

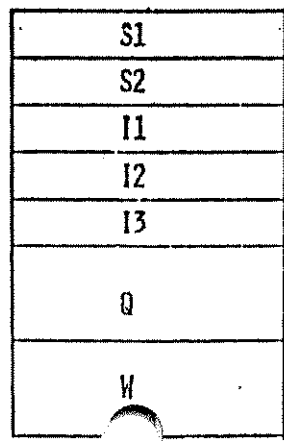
52
468

DATA STORAGE AND ACCESS (CONTINUED)

EXAMPLE 3:

```
CPL: COMPOOL RIGID;  
  DECLARE SCALAR, S1, S2;  
  DECLARE ARRAY(10) INTEGER DOUBLE, I1, I2, I3;  
  STRUCTURE Q:  
    1 V VECTOR,  
    1 SS SCALAR,  
    1 J INTEGER;  
  DECLARE Q Q-STRUCTURE;  
  DECLARE W MATRIX(9,9) DOUBLE;  
  . . .  
CLOSE CPL;
```

RESULTS IN:



ALL DATA HAS BEEN ALLOCATED IN
THE DECLARED ORDER.

INTERNAL TO Q, TERMINALS U, SS,
AND J MAY BE REORDERED!

DATA STORAGE AND ACCESS (CONTINUED)

53
469

EXAMPLE 4:

STRUCTURE Q:

1 A,
2 B1 BIT(4),
2 B2 BIT(8),
2 I INTEGER,
1 B RIGID,
2 S1 SCALAR,
2 S2 SCALAR DOUBLE,
2 J INTEGER,
1 C,
2 V1 VECTOR,
2 V2 VECTOR DOUBLE;

} THIS MUCH OF Q WILL BE RIGID

WE CALL A, B, AND C MINOR STRUCTURES OR FORKS. RIGID MAY BE APPLIED TO MINOR STRUCTURES IN WHICH CASE IT PROPAGATES DOWNWARDS TO ALL LOWER TERMINALS.

DATA STORAGE AND ACCESS (CONTINUED)

SI
470

- THE PHYSICAL ALLOCATION OF RIGID STRUCTURES IS FAIRLY SIMPLE TO GRASP ONCE THE IDEA OF DATA ALIGNMENT AND INDEPENDENCE OF MINOR STRUCTURES IS UNDERSTOOD.
- AS WILL BE SEEN THE CONCEPT OF DENSE STRUCTURES MAKES THINGS SLIGHTLY MORE COMPLICATED.
- BUT AS TO WHAT HAPPENS WHEN HAL/S IS FREE TO REORDER STRUCTURES -- THAT IS A MUCH MORE COMPLEX TOPIC THAT WILL BE TAKEN UP AT A LATER TIME.

BEFORE DISCUSSING DENSE WE NEED SOME FACTS ABOUT PHYSICAL ORGANIZATIONS OF DATA:

9-

DATA STORAGE AND ACCESS (CONTINUED)

**SKIP
471**

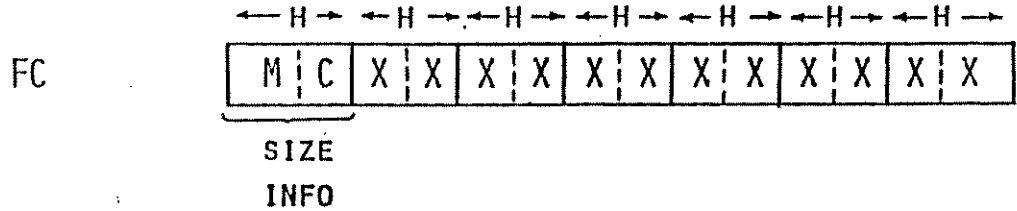
INTEGER	<p>360</p> <div style="border: 1px solid black; width: 80px; height: 20px; margin: 0 auto;"></div> <p>1 HW (2 BYTES)</p>	<p>FC</p> <div style="border: 1px solid black; width: 80px; height: 20px; margin: 0 auto;"></div> <p>1 HW</p>						
INTEGER DOUBLE	<div style="border: 1px solid black; width: 120px; height: 25px; margin: 0 auto;"></div> <p>2 HW (FULLWORD) (4 BYTES)</p>	<div style="border: 1px solid black; width: 120px; height: 25px; margin: 0 auto;"></div> <p>2 HW (FULLWORD)</p>						
SCALAR	<table border="1" style="border-collapse: collapse; margin: 0 auto;"> <tr> <td style="width: 20px; text-align: center;">8</td> <td style="width: 80px; height: 20px;"></td> <td style="width: 20px; text-align: center;">24</td> </tr> </table> <p>2 HW (FULLWORD) (4 BYTES)</p>	8		24	<table border="1" style="border-collapse: collapse; margin: 0 auto;"> <tr> <td style="width: 20px; text-align: center;">8</td> <td style="width: 80px; height: 20px;"></td> <td style="width: 20px; text-align: center;">24</td> </tr> </table> <p>2 HW (FULLWORD)</p>	8		24
8		24						
8		24						
SCALAR DOUBLE	<table border="1" style="border-collapse: collapse; margin: 0 auto;"> <tr> <td style="width: 20px; text-align: center;">8</td> <td style="width: 120px; height: 25px;"></td> <td style="width: 20px; text-align: center;">56</td> </tr> </table> <p>4 HW (DOUBLE WORD) (8 BYTES)</p>	8		56	<table border="1" style="border-collapse: collapse; margin: 0 auto;"> <tr> <td style="width: 20px; text-align: center;">8</td> <td style="width: 120px; height: 25px;"></td> <td style="width: 20px; text-align: center;">56</td> </tr> </table> <p>4 HW (DOUBLE WORD)</p>	8		56
8		56						
8		56						

SUMMARY: EXACT CORRESPONDENCE IN LENGTH AND BOUNDARY ALIGNMENTS
BETWEEN FC AND 360.

DATA STORAGE AND ACCESS (CONTINUED)

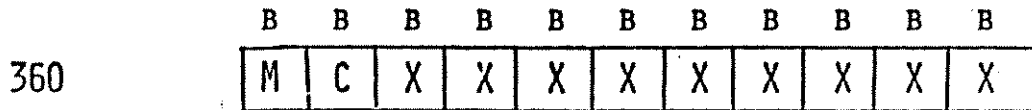
**SKIP
472**

CHARACTER STRINGS:



M ≡ MAX NUMBER OF CHARACTERS
C ≡ CURRENT NUMBER OF CHARACTERS

NOTE: AN FC CHARACTER STRING ALWAYS OCCUPIES AN INTEGRAL NUMBER OF HALFWORDS (2 CHARACTERS PER HW). AN UNUSED 8 BITS MAY EXIST AT THE END.



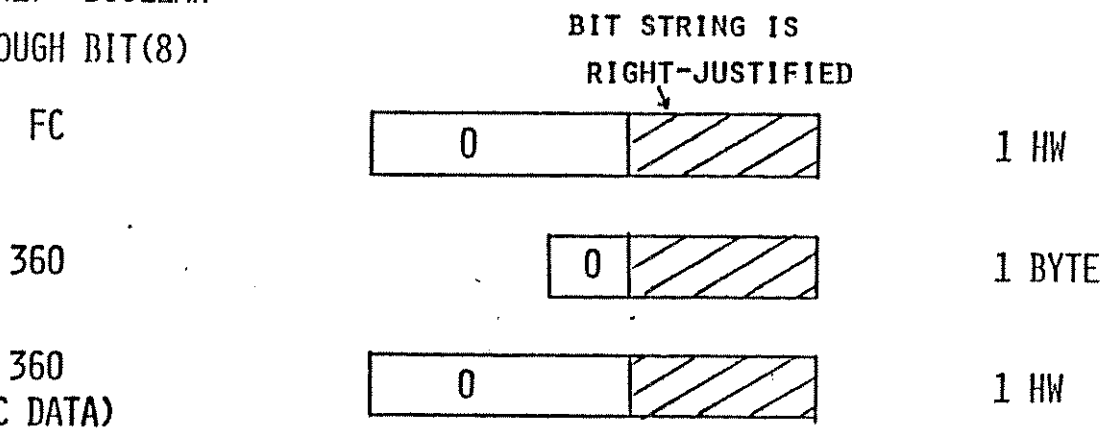
A 360 CHARACTER STRING IS BYTE-ORIENTED. NO WASTED SPACE AT THE END.
BUT, THE FC DATA OPTION (SPECIFIABLE TO THE 360 COMPILER) WILL RESULT IN THE GENERATION OF FC-EQUIVALENT CHARACTER STRINGS.

SKIP
473

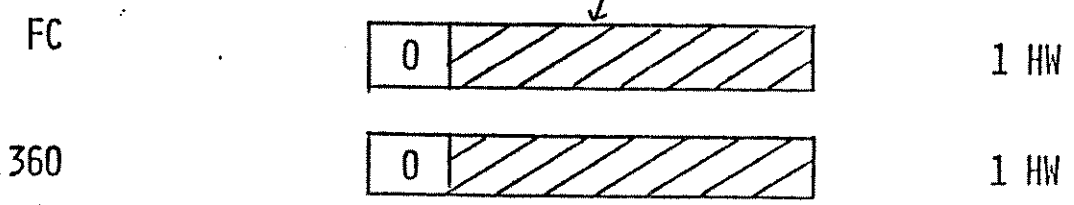
DATA STORAGE AND ACCESS (CONTINUED)

BIT STRINGS:

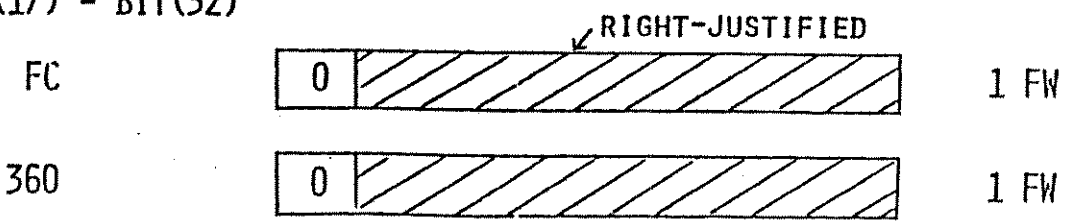
A) BIT(1) BOOLEAN
THROUGH BIT(8)



B) BIT(9) - BIT(16)



C) BIT(17) - BIT(32)



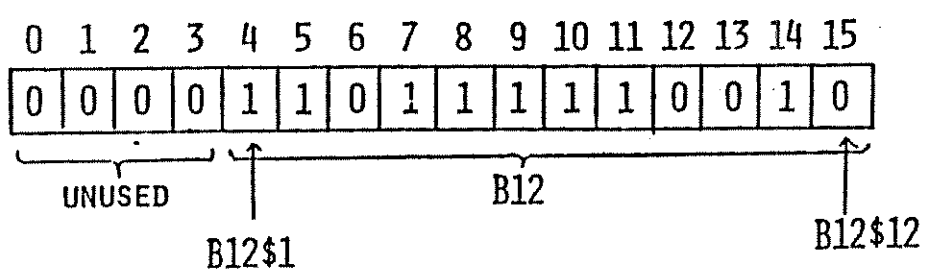
32
474

DATA STORAGE AND ACCESS (CONTINUED)

NOTES:

- (1) HAL/S ALWAYS CAUSES IMPLICIT INITIALIZATION OF BIT STRINGS TO ZERO.
- (2) AT RUN TIME, HAL/S ASSUMES PORTION OF WORD NOT OCCUPIED BY THE BIT STRING IS ZERO.
- (3) BIT STRINGS ARE ALWAYS (EXCEPT FOR DENSE BIT STRINGS) RIGHT-JUSTIFIED IN THEIR MEMORY LOCATION -- COMPONENT SUBSCRIPTING STILL STARTS WITH 1 BEGINNING WITH THE LEFT-MOST BIT OF THE ACTUAL STRING.

```
DECLARE B12 BIT(12) INITIAL(BIN'11011110010');
```

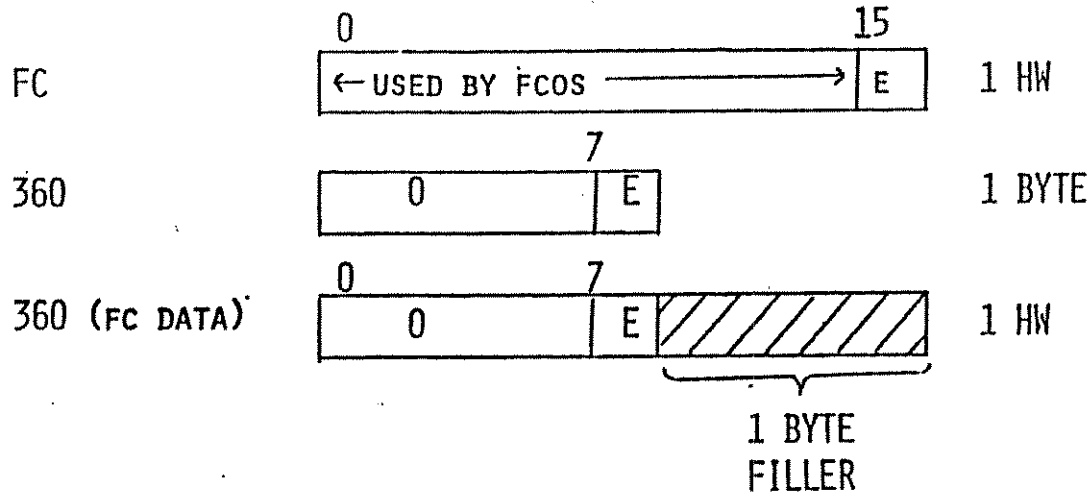


9(-1)

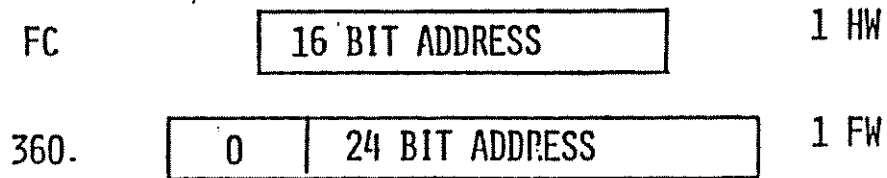
DATA STORAGE AND ACCESS (CONTINUED)

**SKIP
478**

EVENT VARIABLES (ACTS LIKE A BOOLEAN)



NAME VARIABLES* (POINTER DATA)



* NAME VARIABLES REPRESENT AN ALLOCATION INCOMPATIBILITY BETWEEN THE FC AND 360 COMPILERS THAT FC DATA CANNOT OVERCOME!

DATA STORAGE AND ACCESS (CONTINUED)

53
476

DENSE/ALIGNED

THE KEYWORD DENSE IS ONLY EFFECTIVE WHEN APPLIED TO
STRUCTURE DATA (FORKS OR TERMINALS) AND EVEN THEN ONLY
CONTROLS PACKING OF BIT AND BOOLEAN DATA.

DENSE ALLOWS BIT STRINGS WITHIN STRUCTURES TO BE COMPRESSED --
SAVING CORE MEMORY BUT INCREASING ACCESS TIME GENERALLY.

EXAMPLE 1:

```
DECLARE S SCALAR DENSE; ← INEFFECTUAL
DECLARE B6 BIT(6) DENSE; ← INEFFECTUAL
STRUCTURE Q DENSE:
  1 B4 BIT(4),
  1 B10 BIT(10),
  1 B2 BIT(2);
DECLARE Q Q-STRUCTURE;
DECLARE R Q-STRUCTURE DENSE; ← ILLEGAL
```

DATA STORAGE AND ACCESS (CONTINUED)

SI
477

ALL DATA IS INHERENTLY ALIGNED (UNLESS DENSE IS SPECIFIED)
THUS THERE IS NO NEED TO SPECIFY ALIGNED ON DECLARED
DATA -- ALTHOUGH THE KEYWORD ALIGNED WILL BE AUTOMATICALLY
SUPPLIED IN THE CROSS-REFERENCE TABLE.

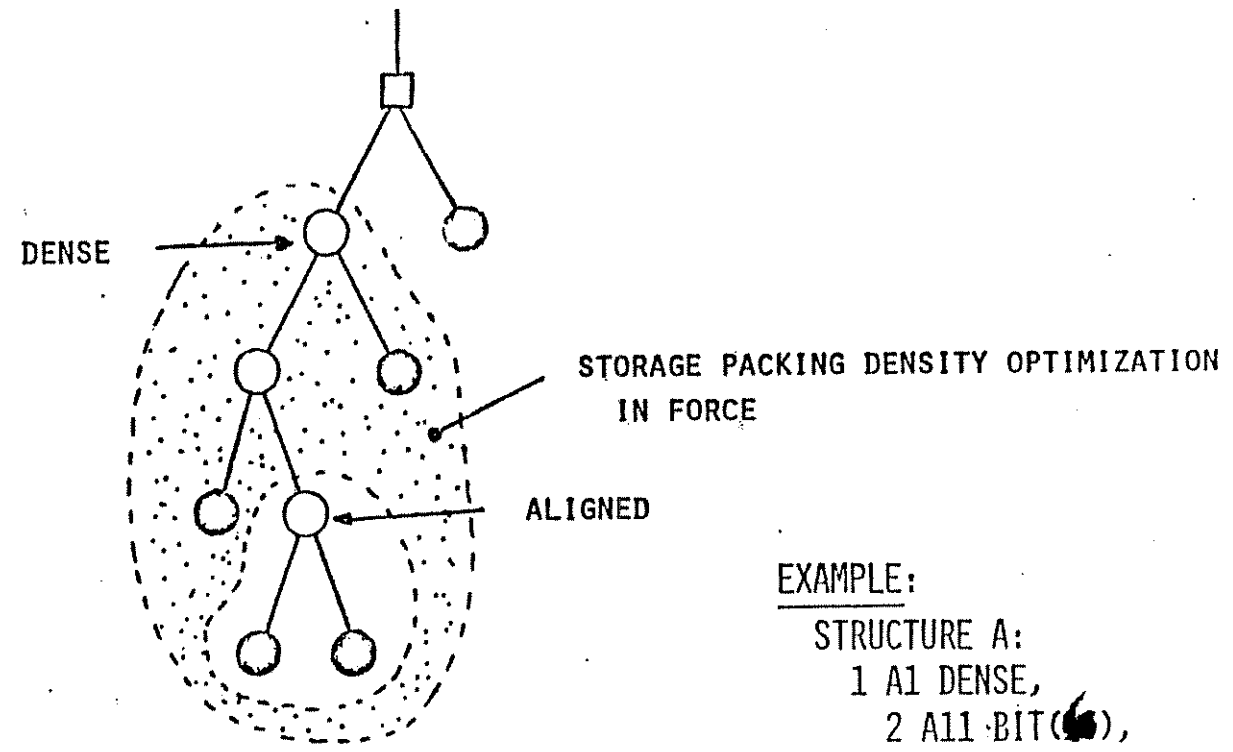
DECLARE I INTEGER ALIGNED;

↑
LEGAL BUT WASTED KEYPUNCHING

ALIGNED, HOWEVER, CAN BE USED WITHIN A STRUCTURE TEMPLATE
TO "TURN OFF" A DENSE SPECIFICATION THAT MAY BE IN FORCE.

53
478

DATA STORAGE AND ACCESS (CONTINUED)



EXAMPLE:

```

STRUCTURE A:
  1 A1 DENSE,
  2 A11 BIT(6),
  2 A12 INTEGER,
  2 A13 ARRAY(10) BOOLEAN
  2 ALIGNED,
  1 A2 CHARACTER(80);
DECLARE ZA A-STRUCTURE;

```

~~NOTE: THIS EXAMPLE IS A BIT FAULTY IN~~
 THAT DENSE NEVER HAS AN EFFECT ON
 ARRAYS OF BIT STRINGS!

52
479

DATA STORAGE AND ACCESS (CONTINUED)

NOTES:

- 1) DENSE IS ONLY EFFECTIVE ON SINGLE (UNARRAYED BIT STRINGS) WITHIN STRUCTURES.
- 2) BOTH DENSE AND RIGID CAN BE SPECIFIED FOR A STRUCTURE TEMPLATE.

	RIGID	-----
DENSE	ADJACENT BIT STRINGS ARE PACKED WHEN POSSIBLE	BIT STRINGS ARE SORTED (WITHIN A MINOR STRUCTURE) AND PACKED TOGETHER.
ALIGNED	NO PACKING OF BITS TAKES PLACE. ALL DATA IS ALLOCATED AS DECLARED.	WITHIN A MINOR STRUCTURE, DATA IS SORTED TO SAVE SPACE. NO BIT PACKING OCCURS.

A FURTHER COMPLICATION EXISTS IN THAT STRUCTURES MAY BE PARTIALLY DENSE AND/OR PARTIALLY RIGID.

THE DETAILED RULES FOLLOWED BY THE COMPILER ARE TOO COMPLEX TO BE TREATED HERE.

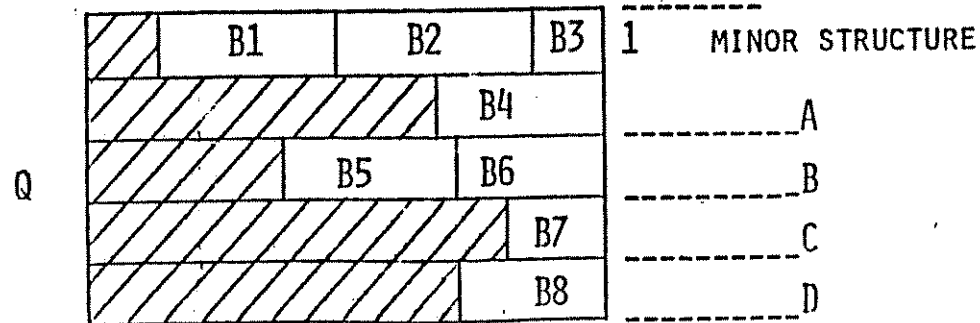
S3
480

DATA STORAGE AND ACCESS (CONTINUED)

EXAMPLE:

STRUCTURE Q DENSE RIGID:

- 1 A,
- 2 B1 BIT(4),
- 2 B2 BIT(8),
- 2 B3 BIT(2),
- 2 B4 BIT(6),
- 1 B,
- 2 B5 BIT(5),
- 2 B6 BIT(5),
- 1 C,
- 2 B7 BIT(3),
- 1 D,
- 2 B8 BIT(5),



NOTE THAT MINOR STRUCTURES ARE INDEPENDENTLY ALLOCATED!!

DATA STORAGE AND ACCESS (CONTINUED)

51
481

A PROGRAMMER USES A DENSE BIT STRING THE SAME WAY A NON-DENSE ONE IS USED ...

STRUCTURE Q DENSE:

1 BOOL1 BOOLEAN,

1 BOOL2 BOOLEAN,

1 B10 BIT(10),

1 B4 BIT(4);

DECLARE Q Q-STRUCTURE INITIAL(TRUE, FALSE, BIN'111', BIN'1')

...

IF BOOL1 AND NOT BOOL2 THEN DO;

...

IF B4 = BIN'1011' THEN DO;

...

DATA STORAGE AND ACCESS (CONTINUED)

51
482

INSTEAD OF THIS,

```
STRUCTURE Q DENSE RIGID:
  1 ENGINE_ON BOOLEAN,
  1 LOW_FUEL BOOLEAN,
  1 WHEELS_DOWN BOOLEAN,
  . . .
  1 SOMETHING_IS_BURNING BOOLEAN;
DECLARE Q Q-STRUCTURE;
```

DO THIS:

```
STRUCTURE Q:
  1 DISCRETES BIT(16);
DECLARE Q Q-STRUCTURE;
REPLACE ENGINE_ON BY "DISCRETES$1";
REPLACE LOW_FUEL BY "DISCRETES$2";
. . .
REPLACE SOMETHING_IS_BURNING BY "DISCRETES$16";
```

DATA STORAGE AND ACCESS (CONTINUED)

SI
483

TEMPORARY DATA

- NORMAL DECLARE'D DATA IS PERMANENTLY ALLOCATED TO A STATIC DATA AREA.
 - THE TEMPORARY FACILITY ALLOWS USERS TO OBTAIN SCRATCH SPACE FOR LOCALIZED CALCULATIONS THAT WILL BE OBTAINED FROM THE STACK.
 - TEMPORARY VARIABLES EXIST ONLY WITHIN THE DO ... END GROUP WITHIN WHICH THEY ARE DEFINED.
 - TWO FORMS EXIST:
 - TEMPORARY S SCALAR,
 - TEMPORARY MATRIX DOUBLE, M1, M2; } NORMAL FORM
- AND
- DO FOR TEMPORARY I = 1 TO 9,
A SPECIAL LOOP VARIABLE FORM.

DATA STORAGE AND ACCESS (CONTINUED)

52
484

RESTRICTIONS

- 1) TEMPORARY DATA ITEMS MAY NOT BE INITIALIZED.
- 2) TEMPORARY EVENT VARIABLES AND NAME VARIABLES ARE NOT ALLOWED.
- 3) THE NAME OF A TEMPORARY MAY NOT DUPLICATE THE NAME OF ANOTHER TEMPORARY DATA ITEM IN THE SAME DO...END GROUP.
- 4) THE NAME OF A TEMPORARY DATA ITEM MAY NOT DUPLICATE THE NAME OF AN ORDINARY DATA ITEM KNOWN BY THE SCOPING RULES TO THE BODY OF THE DO...END GROUP.
- 5) TEMPORARY VARIABLES CAN ONLY BE DEFINED IMMEDIATELY FOLLOWING A DO ... AND PRIOR TO THE NEXT EXECUTABLE FORM (EXCEPT FOR THE LOOP TEMPORARY).

DATA STORAGE AND ACCESS (CONTINUED)

S3
485

6) TEMPORARY VARIABLES MAY NOT BE DEFINED FOR A DO CASE BLOCK.

EXAMPLES

```
-DO FOR TEMPORARY I = 1 TO 10;  
  . . .  
  M$I = M$I + 1;  
-END
```



I IS ONLY KNOWN WITHIN THE DO...END BLOCK. I IS ALWAYS A SINGLE PRECISION INTEGER IN THIS LOOP VARIABLE SPECIAL FORM.

```
-DO;  
  TEMPORARY I INTEGER DOUBLE;  
  TEMPORARY MATRIX, M1, M2, M3;  
  . . .  
  I = I + 1;  
  M1 = M2 + M3;  
  . . .  
-END;
```



I, M1, M2, M3 ARE ONLY KNOWN WITHIN THIS DO...END GROUP (AND ANYTHING NESTED WITHIN IT).

DATA STORAGE AND ACCESS (CONTINUED)

53
486

WRONG:

```
DO;  
  I = I + 1;  
  TEMPORARY INTEGER J;  
  J = 2I;  
END;
```

↑ DOES NOT IMMEDIATELY FOLLOW DO

RIGHT:

```
DO;  
  TEMPORARY INTEGER J;  
  I = I + 1;  
  J = 2I;  
END;
```

WRONG:

```
DO;  
  TEMPORARY INTEGER J INITIAL(3);  
  . . .  
END;
```

↑ TEMPORARIES CANNOT BE INITIALIZED

RIGHT:

```
DO;  
  TEMPORARY INTEGER J;  
  J = 3;  
END;
```

DATA STORAGE AND ACCESS (CONTINUED)

53
487

WRONG:

```
DO;  
  STRUCTURE Q:  
    1 A SCALAR,  
    1 B SCALAR;  
  TEMPORARY R Q-STRUCTURE;  
  . . .  
END;
```

RIGHT:

```
STRUCTURE Q: }  
  1 A SCALAR,  
  1 B SCALAR;  
  . . .  
DO;  
  TEMPORARY R Q-STRUCTURE;  
  . . .  
END;
```

IN DECLARE GROUP OF ANY OUTER BLOCK

WRONG:

```
DO;  
  TEMPORARY INTEGER I;  
  . . .  
  DO;  
    TEMPORARY INTEGER I;  
    . . .  
  END;  
END;
```

TWO I'S IN SAME DO...END GROUP

DATA STORAGE AND ACCESS (CONTINUED)

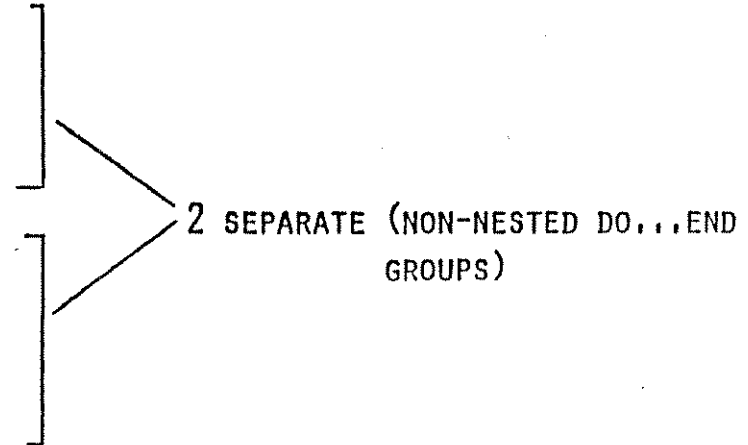
53
488

RIGHT:

```
DO;  
  TEMPORARY INTEGER I;  
  . . .  
    DO;  
      TEMPORARY INTEGER I1;  
      . . .  
    END;  
END;
```

THIS IS OK!

```
DO;  
  TEMPORARY MATRIX M1;  
  . . .  
END;  
. . .  
DO;  
  TEMPORARY MATRIX M1;  
  . . .  
END;
```



NOTE: IN A CASE LIKE THIS, NOT ONLY WILL M1 AND M2 BE ALLOCATED IN THE STACK, BUT M1 AND M2 WILL QUITE LIKELY OCCUPY THE SAME LOCATIONS.

DATA STORAGE AND ACCESS (CONTINUED)

53
489

THIS IS WRONG:

```
DO CASE J;  
  TEMPORARY MATRIX M1;  
  DO; ]  
  . . . ] _____ CASE 1  
  END; ]  
  DO; ]  
  . . . ] _____ CASE 2  
  END; ]  
  . . .  
END;
```

BUT THIS IS OK!

```
DO CASE J;  
  DO;  
    TEMPORARY MATRIX M1;  
    . . .  
  END;  
  DO;  
    . . .  
  END;  
  . . .  
END;
```

DATA STORAGE AND ACCESS (CONTINUED)

53
490

WRONG:

```
PROC: PROCEDURE
  DECLARE I INTEGER;
  . . .
  DO FOR TEMPORARY I = 1 TO 5;
  . . .
END;
```

← DUPLICATE IDENTIFIER

TEMPORARIES SCOPE INTO NESTED DO...END GROUPS ALSO ---

```
{ DO;
  TEMPORARY Z ARRAY(10) INTEGER;
  . . .
  Z$1 = 3;
  . . .
  { DO FOR TEMPORARY I = 1 TO 10;
    IF Z$I = 0 THEN ...
  }
  END;
  . . .
END;
```

DATA STORAGE AND ACCESS (CONTINUED)

53
491

TEMPORARIES ALSO SCOPE INTO NESTED BLOCKS BUT SINCE TEMPORARIES ARE IN THE STACK WE HAVE THE SAME STACK "WALK-BACK" PROBLEM THAT EXISTS FOR PARAMETERS.

EXAMPLE:

```
DO;  
  TEMPORARY MATRIX M;  
  . . .  
  CALL PROC_A;  
  . . .  
  PROC_A: PROCEDURE;  
    . . .  
    IF M$(2,3) = 0 THEN ...  
    . . .  
  CLOSE PROC_A;  
  . . .  
END;
```

M IS VISIBLE BECAUSE WE ARE STILL IN THE DO...END GROUP WHERE M WAS DEFINED. A STACK WALK-BACK LOOP WILL BE GENERATED, HOWEVER.

DATA STORAGE AND ACCESS (CONTINUED)

31
492

LOCK GROUPS

THE CONCEPT OF A LOCK GROUP IS LINKED TO THAT OF AN UPDATE BLOCK. THIS IS DUE TO THE FACT THAT A DATA ITEM WHICH HAS BEEN DECLARED TO BE LOCKED CAN ONLY BE ACCESSED (REFERENCED OR MODIFIED) WITHIN AN UPDATE BLOCK.

ALTHOUGH ANY DECLARED DATA ITEM MAY HAVE A LOCK GROUP SPECIFICATION, IN PRACTICE ONLY CERTAIN COMPOOL VARIABLES WILL NEED SUCH PROTECTION.

THE CURRENT IMPLEMENTATION ALLOWS FOR 15 SEPARATE LOCK GROUPS. CONTROLLING CRUCIAL VARIABLES ON A GROUP BASIS TURNS OUT TO BE FAR MORE PRACTICAL THAN PROTECTING THEM INDIVIDUALLY.

IF A VARIABLE IS TO BE PROTECTED THEN IT IS GIVEN A LOCK SPECIFICATION AT THE TIME IT IS DECLARED.

DATA STORAGE AND ACCESS (CONTINUED)

52
493

EXAMPLE:

```
DECLARE WVECT VECTOR(3),  
        QVECT VECTOR(3) LOCK(7),  
        RVECT VECTOR(3) LOCK(4),
```

HERE WVECT IS UNPROTECTED, QVECT IS LOCKED AND BELONGS TO GROUP 7 AND RVECT IS LOCKED AND BELONGS TO GROUP 4. WVECT CAN BE REFERENCED OR ASSIGNED ANYWHERE. QVECT AND RVECT CAN ONLY BE ACCESSED WITHIN AN UPDATE BLOCK.

FOR THE SAKE OF FLEXIBILITY, A VARIABLE CAN BE DECLARED AS LOCK(*). THIS TECHNICALLY SIGNIFIES THAT THE VARIABLES BELONGS TO ALL LOCK GROUPS SIMULTANEOUSLY.

EXAMPLE:

```
DECLARE VEHICLE_POSITION VECTOR DOUBLE LOCK(*);
```

33
494

DATA STORAGE AND ACCESS (CONTINUED)

THE FOLLOWING EXAMPLES ILLUSTRATE THE POSITIONING OF THE CONSTRUCT
WITHIN DECLARATIONS:

EXAMPLES:

```
| DECLARE I INTEGER DOUBLE LOCK(3);  
| DECLARE S SCALAR INITIAL(5,5) LOCK(*);  
| DECLARE V VECTOR(3) LOCK(1) INITIAL(0);  
| DECLARE B ARRAY(1000) BOOLEAN LOCK(*);  
| STRUCTURE Q DENSE:  
|   1 QI INTEGER,  
|   1 QS SCALAR,  
|   1 QB BIT(16);  
| DECLARE ZQ Q-STRUCTURE(20) LOCK(3);
```

UPDATE BLOCKS

AN UPDATE BLOCK IS AN EXPLICITLY DELIMITED BODY OF CODE WHEREIN LOCKED
DATA MAY BE REFERENCED OR MODIFIED. AN UPDATE BLOCK SUPERFICIALLY LOOKS
LIKE ANY OTHER HAL BLOCK.

label: UPDATE



CLOSE *label*;

DATA STORAGE AND ACCESS (CONTINUED)

51
495

THE UPDATE BLOCK, HOWEVER, IS UNIQUE IN SEVERAL RESPECTS:

- 1) THE BLOCK LABEL IS OPTIONAL. IF IT IS OMITTED THE COMPILER AUTOMATICALLY GENERATES LABELS, I.E.

\$UPDATE1, \$UPDATE2, ETC.

- 2) UPDATE BLOCKS CANNOT BE CALLED, INVOKED, OR SCHEDULE. INSTEAD, THEY ARE EXECUTED WHEN THEY ARE ENCOUNTERED IN THE PATH OF EXECUTION.

NOTE: ALTHOUGH ONE "FALLS INTO" UPDATE BLOCKS (AND THAT IS WHY A LABEL IS OPTIONAL) THEY STILL ARE SEPARATE CSECTS (CONTROL SECTIONS) LIKE ANY OTHER BLOCK AND THUS THERE IS A HIDDEN BRANCH TO THE BLOCK WHICH LOOKS LIKE A NORMAL PROCEDURE CALL. THIS CAN BE SEEN IN THE GENERATED CODE.

32
496

DATA STORAGE AND ACCESS (CONTINUED)

- 3) A BLOCK LABEL, IF THE USER SUPPLIES ONE, MAY BE CONSIDERED AS A STATEMENT LABEL, I.E. IT IS POSSIBLE TO USE A GO TO STATEMENT TO GET TO AN UPDATE BLOCK.

```
| I = I + 1;  
| IF I < 0 THEN GO TO ENTER;  
| J = J + 1;  
| ENTER: UPDATE;  
| E M = M + U.U NTN;  
| CLOSE ENTER;  
| :  
| :
```

- 4) AT THE BEGINNING OF THE EMITTED CODE FOR AN UPDATE BLOCK THERE IS A SUPERVISOR CALL (SVC) TO THE FCOS DESCRIBING WHICH LOCK GROUPS WILL BE USED BY THE UPDATE BLOCK. EXECUTION MAY BE HELD UP AT THIS POINT IF ONE OR MORE OF THE REQUESTED GROUPS ARE ALREADY IN USE.

DATA STORAGE AND ACCESS (CONTINUED)

32
497

- 5) AT THE END OF THE EMITTED CODE FOR AN UPDATE BLOCK THERE IS AN SVC WHICH INFORMS THE FCOS THAT THE LOCK GROUPS ARE NOW FREE.
- 6) AN UPDATE BLOCK CAN HAVE DECLARED LOCAL DATA JUST LIKE ANY OTHER BLOCK.
- 7) THE FOLLOWING (UNUSUAL) RESTRICTIONS APPLY, HOWEVER:
 - NO I/O STATEMENTS OF ANY KIND ARE LEGAL, E.G. READ, WRITE, ETC. (HAL/S WILL NOT STOP %SVC, HOWEVER).
 - MOST REAL-TIME STATEMENTS ARE ILLEGAL, NAMELY:
SCHEDULE, WAIT, CANCEL, TERMINATE,
AND UPDATE PRIORITY.

DATA STORAGE AND ACCESS (CONTINUED)

33
498

- AN UPDATE BLOCK CAN HAVE NESTED PROCEDURES AND FUNCTIONS BUT NO UPDATE BLOCKS OR TASK BLOCKS.
- THE ONLY PROCEDURE OR FUNCTION INVOCATIONS WHICH ARE LEGAL ARE THOSE REFERENCING PROCEDURE OR FUNCTION BLOCKS DEFINED WITHIN IT.

UPDATE BLOCKS SHOULD GENERALLY BE AS FAST AS POSSIBLE SINCE THEY TIE UP LOCK GROUPS THAT MAY KEEP MORE CRITICAL UPDATE BLOCKS FROM BEING EXECUTED.

NOR SHOULD UPDATE BLOCKS BE TOO FREQUENT SINCE THERE IS NOT ONLY THE OVERHEAD OF 2 FCOS SERVICES BUT ALL GPC'S ARE USUALLY SYNCHED UPON ENTRY TO AN UPDATE BLOCK.

DATA STORAGE AND ACCESS (CONTINUED)

53
500

NOTE: THE COMPILER AUTOMATICALLY INFORMS FCOS ON WHICH LOCK GROUPS TO LOCK BASED ON WHICH LOCKED VARIABLES ARE REFERENCED.

FURTHERMORE, IF LOCKED DATA IS ONLY REFERENCED (NOT ASSIGNED) FCOS IS INFORMED OF THIS ALSO -- ALTHOUGH REFERENCE-ONLY UPDATE BLOCKS ARE NOT CURRENTLY GIVEN ANY FAVORED TREATMENT.

9-3

3

3

DATA STORAGE AND ACCESS (CONTINUED)

52
500-

EXECUTION OF UPDATE BLOCKS

THE BEHAVIOR OF PROCESSES ON ENCOUNTERING UPDATE BLOCKS HAS ALREADY BEEN DESCRIBED IN THIS SECTION, BUT ONLY SUPERFICIALLY BY EXAMPLE. THIS BEHAVIOR IS NOW RE-EXAMINED IN MORE DETAIL.

THE SIMPLEST CASE IS THAT OF TWO PROCESSES WISHING TO USE DATA ITEMS FROM THE SAME LOCK GROUP. EACH PROCESS HAS TO EXECUTE AN UPDATE BLOCK TO USE THE PROTECTED DATA ITEMS. THE FOLLOWING ACTIVITY TAKES PLACE:

- IF BOTH OF THE PROCESSES REQUIRE DATA ITEMS FROM THE SAME LOCK GROUP TO BE MODIFIED THEN THE FIRST PROCESS TO ENTER ITS UPDATE BLOCK MUST COMPLETE EXECUTION OF IT BEFORE THE SECOND PROCESS CAN ENTER ITS OWN UPDATE BLOCK. THE RTE PLACES THE SECOND PROCESS IN A WAITING STATE FOR THIS PERIOD OF TIME.

WISHFUL
THINKING

- IF ONE OR BOTH OF THE PROCESSES ONLY REQUIRE TO REFERENCE THE DATA THEN IN SOME IMPLEMENTATIONS OF HAL/S, THE BEHAVIOR OF THE RTE WILL BE THE SAME AS BEFORE. ALTERNATIVELY, IN OTHER IMPLEMENTATIONS, TO REDUCE THE SECOND PROCESS' WAITING TIME, THE RTE MAY ALLOW PARTIAL OVERLAP IN EXECUTION OF THE UPDATE BLOCKS, CONSISTENT WITH EXCLUSIVE USE OF DATA BY THE PROCESS MODIFYING IT*.

DATA STORAGE AND ACCESS (CONTINUED)

52
500-2

IF THE TWO PROCESSES WISH TO USE DATA FROM MORE THAN ONE LOCK GROUP, THE RTE TRACKS THE USE OF EACH LOCK GROUP IN THE ABOVE WAY. IF ONE OR BOTH PROCESSES USE DATA PROTECTED BY LOCK(*), THEN THE SITUATION IS EQUIVALENT TO ONE IN WHICH THE PROCESS OR PROCESSES WISH TO USE DATA IN EVERY LOCK GROUP.

IF DATA IS SHARED BY MORE THAN TWO PROCESSES, THEN ALL PROCESSES EXCEPT ONE ARE PUT IN A WAITING STATE BY THE RTE. THE EVENTUAL ORDER IN WHICH THE PROCESSES COMPLETE EXECUTION OF THEIR UPDATE BLOCKS WILL DEPEND ON THE CONTENTS OF THE PROCESS QUEUE AND THE RELATIVE PRIORITY OF THE PROCESSES.

DATA STORAGE AND ACCESS (CONTINUED)

5/
500-3

EXAMPLE

IN SOME REAL TIME APPLICATION, IT IS REQUIRED THAT A PROCESS ALPHA PRINT THE VALUES OF A COVARIANCE MATRIX M ONCE EVERY 19 SECONDS. THE VALUES ARE UPDATED ONCE EVERY 1.5 SECONDS BY A SECOND PROCESS BETA. THE IMPLEMENTATION MUST GUARANTEE THAT A PARTIALLY UPDATED COVARIANCE MATRIX NOT BE PRINTED.

THE COVARIANCE MATRIX M IS DECLARED THUS:

```
DECLARE M MATRIX(3,3) LOCK(1);
```

TWO TASK BLOCKS CORRESPONDING TO ALPHA AND BETA ARE SHOWN BELOW:

DATA STORAGE AND ACCESS (CONTINUED)

52
500-4

EXAMPLE CONTINUED:

```
| ALPHA: TASK;  
|   DECLARE M_LOCAL MATRIX(3,3);  
| U1: UPDATE;  
|   M_LOCAL = M;  
|   CLOSE U1;  
|   WRITE(6) 'COVARIANCE=', M_LOCAL;  
| CLOSE ALPHA;  
|  
| C  
| BETA TASK;  
|   DECLARE VT VECTOR(3);  
| U2: UPDATE;  
|  
| E  
|   V = (PHI M PHIT + QA)Z;  
|   M = V V/(QB + Z.V);  
|   CLOSE U2;  
| CLOSE BETA;  
|
```

9-4

DATA STORAGE AND ACCESS (CONTINUED)

52
500-4

EXAMPLE CONTINUED:

```
| ALPHA: TASK;  
|   DECLARE M_LOCAL MATRIX(3,3);  
| U1: UPDATE;  
|   M_LOCAL = M;  
|   CLOSE U1;  
|   WRITE(6) 'COVARIANCE=', M_LOCAL;  
| CLOSE ALPHA;  
| C  
| BETA TASK;  
|   DECLARE VT VECTOR(3);  
| U2: UPDATE;  
| E  
|   V = (PHI M PHIT + QA)Z;  
|   M = V V/(QB + Z.V);  
|   CLOSE U2;  
| CLOSE BETA;  
|
```

DATA STORAGE AND ACCESS (CONTINUED)

S3
500-5

EXAMPLE CONTINUED:

PROCESSES ALPHA AND BETA COULD BE CREATED BY INVOKING THESE TASK BLOCKS WITH CYCLIC SCHEDULE STATEMENTS OF THE FOLLOWING FORM:

```
| SCHEDULE ALPHA PRIORITY(10), REPEAT EVERY(19);  
| SCHEDULE BETA PRIORITY(20), REPEAT EVERY(1,5);  
|
```

THE FOLLOWING DIAGRAM SHOWS THE STATE TRANSITIONS OF THE PROCESSES:

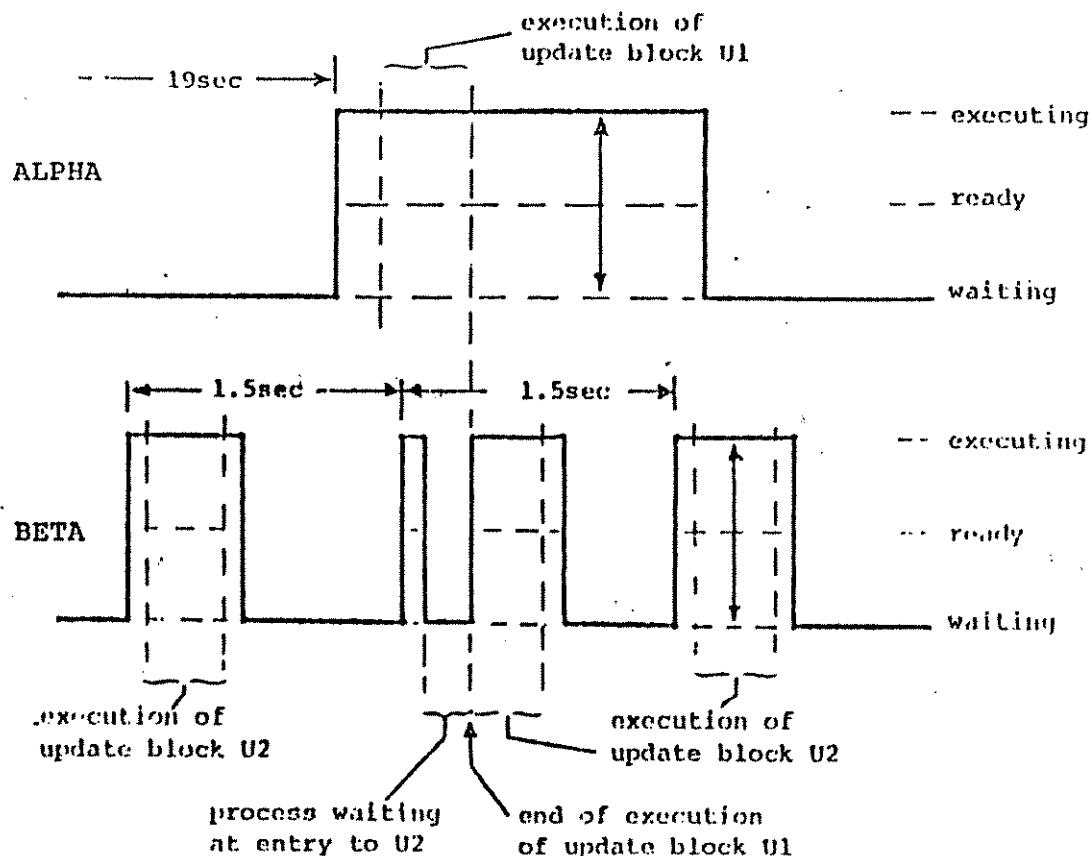
9.57



53
500-6

DATA STORAGE AND ACCESS (CONTINUED)

EXAMPLE CONTINUED:



NOTE THAT IF IN THIS EXAMPLE PROCESS SWAPS OCCURRED ONLY ON STATEMENT BOUNDARIES, UPDATE BLOCKS WOULD NOT BE NEEDED SINCE ALPHA COULD NOT EVER BE BROUGHT INTO EXECUTION WITH COVARIANCE MATRIX M PARTLY UPDATED.

DATA STORAGE AND ACCESS (CONTINUED)

51
500-7

LOCKED ASSIGN ARGUMENTS

THE RULE THAT LOCKED DATA ITEMS CAN ONLY APPEAR IN UPDATE BLOCKS HAS ONE SOLE EXCEPTION: IT IS POSSIBLE FOR LOCKED DATA ITEMS TO APPEAR AS ASSIGN ARGUMENTS IN PROCEDURE INVOCATIONS. THIS PROVIDES THE ABILITY TO "PARAMETERIZE" UPDATE BLOCKS, AS WILL BE SHOWN IN AN ENSUING EXAMPLE.

THE FOLLOWING RULES GOVERN THE PASSAGE OF LOCKED ASSIGN ARGUMENTS:

DATA STORAGE AND ACCESS (CONTINUED)

S2
500-8

LOCKED ASSIGN ARGUMENTS (CONTINUED):

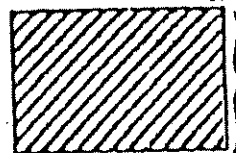
1. IF THE ARGUMENT IS A DATA ITEM BELONGING TO LOCK GROUP N, THEN THE CORRESPONDING PARAMETER MUST BE DECLARED LOCK(N) OR LOCK(*).
2. IF THE ARGUMENT IS A DATA ITEM BELONGING TO ALL LOCK GROUPS, THE CORRESPONDING PARAMETER MUST BE DECLARED LOCK(*).
3. ARGUMENT AND PARAMETER MUST ALSO MATCH IN THE SENSES DESCRIBED, AS APPLICABLE.

DATA STORAGE AND ACCESS (CONTINUED)

52
500-9

EXAMPLE:

```
DECLARE A SCALAR LOCK(1),  
        B SCALAR LOCK(2),  
        C SCALAR LOCK(*);  
PICK: PROCEDURE(P) ASSIGN(Q,R);  
    DECLARE P SCALAR,  
            Q SCALAR LOCK(1),  
            R SCALAR LOCK(*);
```



} body of procedure

```
CLOSE PICK;
```

For the above procedure definitions and declarations,
the following invocations are legal:

```
CALL PICK(1.0) ASSIGN(A,B);  
CALL PICK(2.0) ASSIGN(A,C);
```

The following are illegal:

```
CALL PICK(N) ASSIGN(B,C);  
CALL PICK(3.0) ASSIGN(C,B);
```

locked data item as input argument.
unmatched lock group

DATA STORAGE AND ACCESS (CONTINUED)

S3
500-11

EXAMPLE CONTINUED:

The procedure PICK may contain an update block changing the values of Q and R:

```
PICK: PROCEDURE(P) ASSIGN(Q,R);  
  DECLARE P SCALAR,  
         Q SCALAR LOCK(1),  
         R SCALAR LOCK(+);  
  .  
  .  
  .  
  U: UPDATE;  
    Q = Q + P;  
    R = R - P;  
  CLOSE U;  
  .  
  .  
  .  
  CLOSE PICK;
```

PICK may be invoked with different locked assign arguments, thus effectively parameterizing the update block.

```
| CALL PICK(1) ASSIGN(A,B);      updates A and B  
| CALL PICK(2) ASSIGN(A,C);      updates A and C
```


DATA STORAGE AND ACCESS (CONTINUED)

S1
500-11

FURTHER NOTES:

- 1) STRUCTURES MAY BE LOCKED:

STRUCTURE Q:

1 A,
2 A1 SCALAR,
2 A2 SCALAR,
1 B,
2 B1 INTEGER,
2 B2 INTEGER;

DECLARE Q Q-STRUCTURE LOCK(5);

IN WHICH CASE AN UPDATE BLOCK IS NEEDED TO ACCESS ANY PART OF THE STRUCTURE.

- 2) THE KEYWORD LOCK MAY NOT APPEAR WITHIN THE TEMPLATE ITSELF, I.E., EITHER THE ENTIRE STRUCTURE IS LOCKED OR NO PIECE OF IT IS.

THIS TURNS OUT TO BE AN UNFORTUNATE RESTRICTION IN THE CASE OF NAME VARIABLES (AS WE WILL SEE LATER), I.E. THE FOLLOWING IS ILLEGAL (ALTHOUGH LOGICALLY REASONABLE).

DATA STORAGE AND ACCESS (CONTINUED)

52
500-12

STRUCTURE Q: ILLEGAL!
1 A SCALAR,
1 NM NAME MATRIX LOCK(3),
1 V VECTOR;

- 3) A DISADVANTAGE OF LOCK GROUPS IS THAT AN UPDATE BLOCK IS LEGALLY REQUIRED EVEN WHEN THE SOFTWARE DESIGN GUARANTEES THAT CERTAIN "SAFE" TIMES EXIST WHEN ONLY ONE PROCESS WILL BE ACCESSING THE LOCKED DATA.
- 4) THE NEW %COPY FACILITY DOES ALLOW LOCKED DATA TO BE COPIED OR UPDATED WITHOUT THE NEED FOR AN UPDATE BLOCK.

51
500-13

EXCLUSIVE PROCEDURES/FUNCTIONS

AN EXCLUSIVE PROCEDURE OR FUNCTION (USUALLY A COMSUB) IS ONE WHICH POSSESSES A SPECIAL FCOS INTERFACE THAT GUARANTEES THAT ONLY ONE PROCESS AT A TIME MAY ENTER THE BLOCK, I.E. THE FIRST PROCESS TO ENTER THE BLOCK LOCKS OUT ALL OTHER POTENTIAL CALLERS UNTIL IT HAS FINISHED AND HAS EXITED THE BLOCK.

DEFINING AN EXCLUSIVE PROCEDURE

THE FORM OF THE OPENING STATEMENT OF AN EXCLUSIVE PROCEDURE IS AS SHOWN BELOW:

```
| label: PROCEDURE(i1,i2,...) ASSIGN(a1,a2,...) EXCLUSIVE;
```

1. *label* is a legal HAL/S identifier constituting the procedure name.
2. *i*¹,*i*²,... and *a*¹,*a*²,... are lists of input and assign parameters as described.
3. The keyword EXCLUSIVE designates an exclusive procedure.

EXCLUSIVE PROCEDURES/FUNCTIONS (CONTINUED)

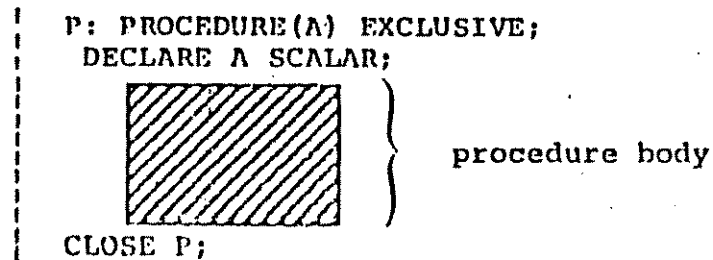
S2
500-14

NOTE: ALTHOUGH IT IS FAR EASIER TO MAKE A BLOCK EXCLUSIVE THAN REENTRANT, EACH EXECUTION OF AN EXCLUSIVE BLOCK CARRIES THE OVERHEAD OF 2 FCOS SUPERVISOR CALLS.

EXCLUSIVE BLOCKS ALSO MAY RESULT IN TEMPORARY "LOCK-OUT" OF MORE CRITICAL PROCESSES.

THIS IS ALL IT TAKES TO MAKE A BLOCK EXCLUSIVE

Example:



The template corresponding to an exclusive external procedure must also bear the keyword EXCLUSIVE.

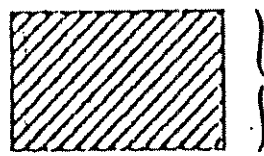
83
500-15

EXCLUSIVE PROCEDURES/FUNCTIONS (CONTINUED)

Example:

The template corresponding to

P: PROCEDURE(A) EXCLUSIVE;
 DECLARE A SCALAR;



} procedure body

CLOSE P;

would be:

P: EXTERNAL PROCEDURE(A) EXCLUSIVE;
 DECLARE A SCALAR;
CLOSE P;

EXCLUSIVE PROCEDURES/FUNCTIONS (CONTINUED)

S2
500-16

DEFINING AN EXCLUSIVE FUNCTION

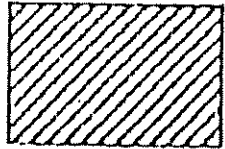
THE FORM OF THE OPENING STATEMENT OF AN EXCLUSIVE FUNCTION IS AS SHOWN BELOW:

- | *label*: FUNCTION(i^1, i^2, \dots) *attributes* EXCLUSIVE;
|
1. *label* is a legal HAL/S identifier constituting the function name.
 2. i^1, i^2, \dots is a list of input parameters as described earlier.
 3. *attributes* defines the type and, where applicable, precision of the function.
 4. The keyword EXCLUSIVE designates an exclusive function.

EXCLUSIVE PROCEDURES/FUNCTIONS (CONTINUED)

S3
500-17

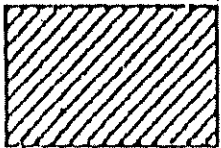
Example:

```
| F: FUNCTION BOOLEAN EXCLUSIVE;  
|  } function body  
| CLOSE F;
```

The template corresponding to an exclusive external function must also bear the keyword EXCLUSIVE.

Example:

* The template corresponding to:

```
| F: FUNCTION BOOLEAN EXCLUSIVE;  
|  } function body  
| CLOSE F;
```

would be:

```
| F: EXTERNAL FUNCTION BOOLEAN EXCLUSIVE;  
| CLOSE F;
```

9-57

EXCLUSIVE PROCEDURES/FUNCTIONS (CONTINUED)

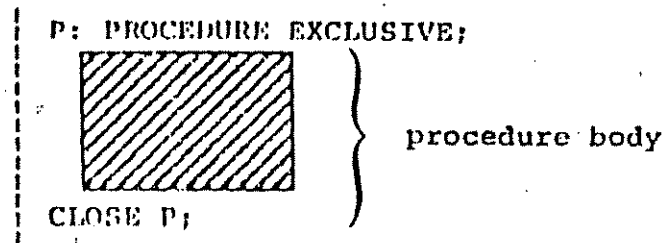
S/
500-18

BEHAVIOR OF EXCLUSIVE PROCEDURES AND FUNCTIONS

If an exclusive procedure or function is in use by a process A, and a process B tries to invoke it, then the RTE places process B in the waiting state until process A returns from its use.

Example:

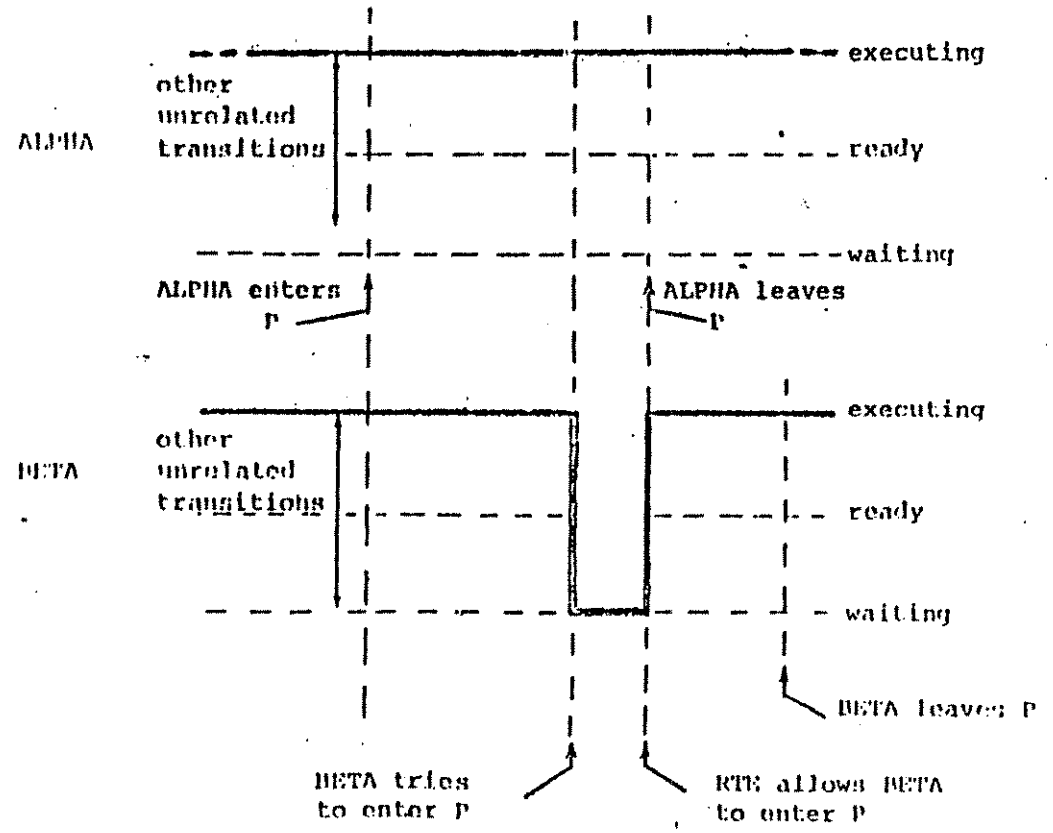
Two processes ALPHA and BETA can invoke the following procedure:



S2
500-19

EXCLUSIVE PROCEDURES/FUNCTIONS (CONTINUED)

Suppose that ALPHA invokes P first and during its execution, BETA tries to invoke it. The state transitions for this situation is shown below:



9-7

9-7

9-7

REENTRANT PROCEDURES/FUNCTIONS

S/
500-20

- DUE TO THE HAL/S STACK MECHANISM AND THE METHOD OF PARAMETER PASSAGE AND PROCEDURE CALL, EVERY PROCEDURE/FUNCTION IS ALMOST REENTRANT.
- TO MAKE A BLOCK LEGALLY REENTRANT IT IS NECESSARY TO APPEND THE KEYWORD REENTRANT TO THE BLOCK HEADER.
- TO MAKE A BLOCK ACTUALLY REENTRANT REQUIRES MORE WORK AND A LOT OF CARE!

REENTRANCY MEANS, OF COURSE, THAT TWO OR MORE PROCESSES (PROGRAMS OR TASKS) MAY BE "SIMULTANEOUSLY" EXECUTING THE BLOCK.

REENTRANT PROCEDURES/FUNCTIONS (CONTINUED)

52
500-21


DEFINING A REENTRANT PROCEDURE

THE FORM OF THE OPENING STATEMENT OF A REENTRANT PROCEDURE IS SHOWN BELOW:

```
| label: PROCEDURE(i1,i2,...) ASSIGN(a1,a2,...) REENTRANT;
```

1. *label* is a legal HAL/S identifier constituting the procedure name.
2. *i*¹,*i*², and *a*¹,*a*²,... are lists of input and assign parameters.
3. The keyword REENTRANT indicates that the procedure is to be considered reentrant.

Example:

```
| P: PROCEDURE REENTRANT  
|  } procedure body  
| CLOSE P;
```

If P were an external procedure, the corresponding template would be:

```
| P: EXTERNAL PROCEDURE REENTRANT;  
| CLOSE P;
```

961
OO

REENTRANT PROCEDURES/FUNCTIONS (CONTINUED)

S3
500-21

DEFINING A REENTRANT FUNCTION

The form of an opening statement of a reentrant function is shown below:

| *label* : FUNCTION(i^1, i^2, \dots) *attributes* REENTRANT;

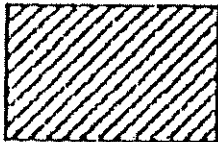
1. *label* is a legal HAL/S identifier constituting the function name.
2. i^1, i^2, \dots is a list of input parameters as described in Section 11.2.
3. *attributes* defines the type and, where applicable, precision of the function as described in Section 11.2.
4. The keyword REENTRANT indicates that the function is to be considered reentrant.

The template corresponding to an external reentrant function must also possess the keyword REENTRANT.

REENTRANT PROCEDURES/FUNCTIONS (CONTINUED)

S3
800-23

Example:

```
| F: FUNCTION MATRIX(4,4) REENTRANT;  
|  } function body  
| CLOSE F;
```

If F were an external function, the corresponding template would be:

```
| F: EXTERNAL FUNCTION MATRIX(4,4) REENTRANT;  
| CLOSE F;
```

9-630

REENTRANT PROCEDURES/FUNCTIONS (CONTINUED)

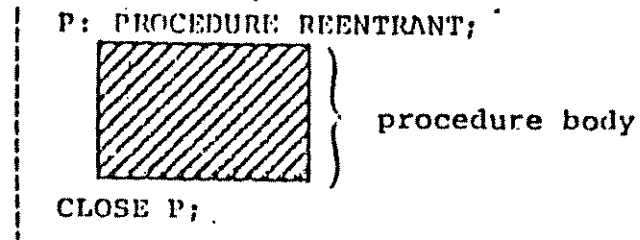
52
500-24

BEHAVIOR OF REENTRANT PROCEDURES AND FUNCTIONS

If a reentrant procedure or function is in use by a process A, and a process B tries to invoke it, the RTE allows the invocation to proceed without restriction.

Example:

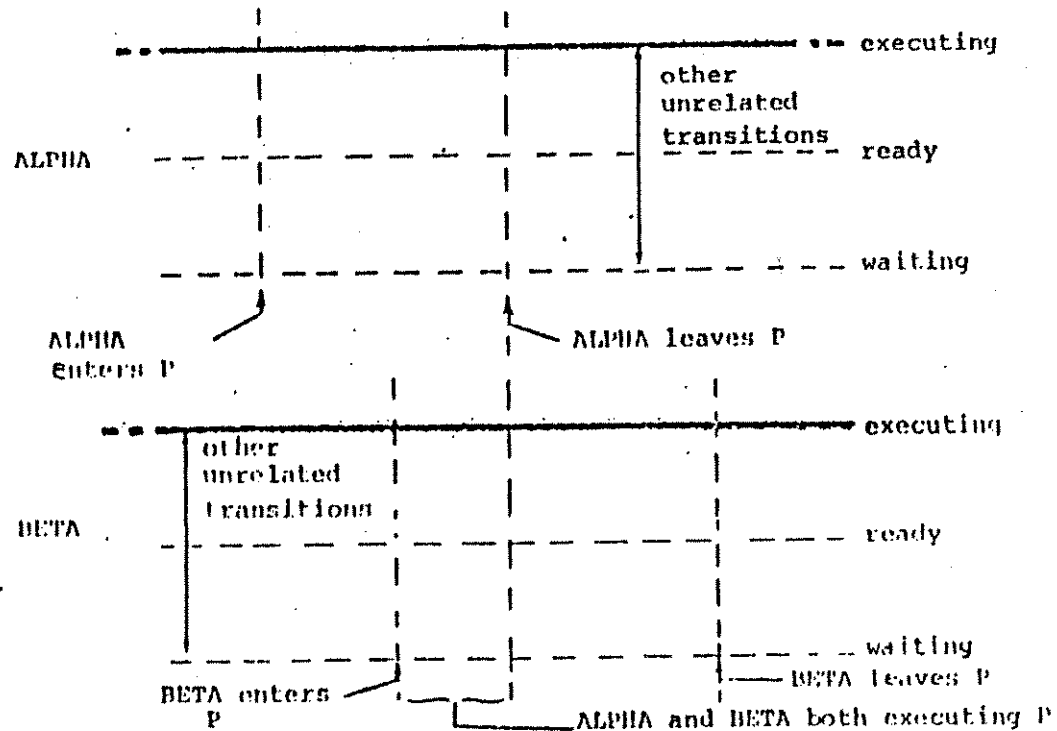
Two processes, ALPHA and BETA, can invoke the following procedure:



Suppose that ALPHA invokes P first and during its execution, BETA invokes it. The state transitions for this situation is as shown below (compare corresponding example for exclusive procedure):

REENTRANT PROCEDRES/FUNCTIONS (CONTINUED)

S3
500-25



REENTRANT PROCEDURES/FUNCTIONS (CONTINUED)

52
500-21

TRUE REENTRANCY GENERALLY REQUIRES THAT EVERY PROCESS ENTERING A REENTRANT BLOCK BE GIVEN ITS OWN COPY OF ANY LOCAL DATA THE BLOCK MAY USE. THIS IS ACCOMPLISHED BY SOMEHOW CAUSING THE LOCAL DATA TO BE ESTABLISHED IN THE STACK OF THE CALLING PROCESS (AKIN TO USING A GETMAIN'ED AREA) RATHER THAN HAVING A SINGLE COPY OF THE DATA PERMANENTLY ALLOCATED IN A STATIC AREA.

NOTE: PARAMETERS PASSED TO THE BLOCK ARE ALREADY IN THE STACK OF THE CALLING PROCESS SO NO PROBLEM EXISTS HERE.

LOCAL DATA WHICH IS TO BE DYNAMICALLY ALLOCATED (FROM THE CALLER'S STACK) MUST BE DECLARED WITH THE AUTOMATIC ATTRIBUTE.

REENTRANT PROCEDURES/FUNCTIONS (CONTINUED)

53
500-27

IN A NON-REENTRANT BLOCK THE KEYWORD AUTOMATIC, IN CONJUNCTION WITH AN INITIAL LIST, SIMPLY MEANT THAT CODE WAS TO BE AUTOMATICALLY GENERATED BY THE COMPILER TO CAUSE THE INDICATED INITIALIZATION AT EACH ENTRY TO THE BLOCK. IT THUS MADE NO SENSE TO NOT HAVE AN INITIAL LIST.

WITH A REENTRANT BLOCK, HOWEVER, AUTOMATIC ADDITIONALLY MEANS THAT THE ALLOCATION OF THE DATA IS TO BE MADE FROM THE WORK AREA (STACK) PROVIDED BY THE CALLER. IN THIS CONTEXT AUTOMATIC MAKES SENSE EVEN WITHOUT AN INITIAL LIST.

NOTE: IN MOST NON-CONTRIVED CASES, ALL DATA DECLARED LOCAL TO A REENTRANT PROCEDURE SHOULD BE DECLARED AS AUTOMATIC. THE DEFAULT (WHICH IS STATIC) WOULD MEAN THAT ALL USERS ARE ASYNCHRONOUSLY MODIFYING THE SAME MEMORY LOCATION.

REENTRANT PROCEDURES/FUNCTIONS (CONTINUED)

52
500-28

EXAMPLES:

IN THE REENTRANT PROCEDURE:

```
P: PROCEDURE(A) ASSIGN(B) REENTRANT;  
  DECLARE A VECTOR;  
  DECLARE B SCALAR; ]  
  DECLARE V VECTOR(3) AUTOMATIC;  
  :  
  :  
  V = VECTOR(B, 0, 0);  
  B = V.A;  
CLOSE P;
```

NOTE: A AND B, BEING PARAMETERS,
ARE ALREADY IN A DYNAMICALLY
ALLOCATED AREA.

REENTRANT PROCEDURES/FUNCTIONS (CONTINUED)

32
500-29

IN CONTRAST, SUPPOSE THE NUMBER OF TIMES A REENTRANT PROCEDURE IS INVOKED IS REQUIRED TO BE KNOWN AND PRINTED EVERY 10 INVOCATIONS. IN THIS UNUSUAL, AND RATHER ARTIFICIAL CASE, IT WOULD BE APPROPRIATE TO USE A LOCAL DATA ITEM NOT DECLARED AUTOMATIC:

```
| P2: PROCEDURE(A,B) ASSIGN(C) REENTRANT;  
|   DECLARE VECTOR, A, B, C;  
|   DECLARE COUNT INTEGER INITIAL(0);  
|   COUNT = COUNT + 1;  
|   IF REMAINDER(COUNT, 10) = 0 THEN  
|     WRITE(6) 'NUMBER OF ENTRIES='||COUNT;  
|     .  
|     .  
|   CLOSE P2;
```

IN AN IMPLEMENTATION WHERE PROCESS SWAPS CAN ONLY OCCUR AT THE END OF EVERY EXECUTABLE STATEMENT, THE CODE SHOWN WOULD MAINTAIN A CORRECT COUNT OF THE NUMBER OF INVOCATIONS.

REENTRANT PROCEDURES/FUNCTIONS (CONTINUED)

S3
500-3d

HISTORICAL NOTE:

AT VARIOUS TIMES IN THE PAST IT HAS BEEN SUGGESTED THAT THE
DEFAULT BE AUTOMATIC FOR DATA DECLARED IN A REENTRANT PROCEDURE
INSTEAD OF THE NORMAL STATIC. THIS HAS NOT BEEN DONE.

FURTHER CONSIDERATIONS FOR ACHIEVING REENTRANCY:

- 1) PROCEDURES AND FUNCTIONS DEFINED WITHIN A REENTRANT BLOCK MUST
ALSO POSSESS THE REENTRANT ATTRIBUTE IF THEY TO POSSESS
LOCAL DATA WHICH IS REQUIRED TO PARTICIPATE IN THE REENTRANCY.
- 2) UPDATE BLOCKS WITHIN A REENTRANT BLOCK MUST NOT DECLARE ANY
LOCAL DATA (STATIC OR AUTOMATIC).
- 3) A PROCEDURE OR FUNCTION CALLED BY A REENTRANT BLOCK MUST ITSELF
ALSO BE REENTRANT.

SI
500-31

HAL/S CSECTS

IN GENERAL, A HAL/S COMPILATION RESULTS IN THE GENERATION OF A NUMBER OF CSECTS (CONTROL SECTIONS). WE MIGHT CALL THIS A CSECT FAMILY SINCE ALL CSECTS EMITTED BY A SINGLE COMPILATION WILL SHARE 6 (OR LESS) LETTERS -- THE GENERIC NAME.

THE GENERIC NAME IS ESTABLISHED BY TAKING THE FULL NAME OF THE COMPILATION BLOCK ITSELF, REMOVING ALL UNDERSCORES, AND THEN TAKING THE FIRST 6 CHARACTERS OF WHAT REMAINS (RIGHT-PADDING WITH BLANKS IF NECESSARY). FOR THIS REASON, COMPILATION UNIT NAMES SHOULD BE CHOSEN SUCH THAT THE GENERIC NAMES WILL BE UNIQUE.

EXAMPLE:

```
CG1_GNC: COMPOOL;  
    .  
    .  
    .  
CLOSE;
```

THE GENERIC NAME WILL BE

CG1GNC

9-71

00



0

HAL/S CSECTS (CONTINUED)

SI
500-32

ASSUMING THE GENERIC NAME IS NNNNNN HAL/S PRODUCES THE FOLLOWING
CSECTS:

(MAJOR DECLARED DATA CSECTS)

#PNNNNNN A COMPOOL DATA CSECT

#DNNNNNN A PROGRAM OR COMSUB DATA CSECT

(OUTER CODE BLOCKS)

\$ONNNNNN A PROGRAM CODE BLOCK

#CNNNNNN A COMSUB CODE BLOCK

(INTERNAL CODE BLOCKS)

anNNNNNN INTERNAL PROCEDURES, FUNCTIONS, UPDATE BLOCKS.
 a = (A-Z)
 n = (1-9)

\$CNNNNNN TASK CODE BLOCKS.
 C = 1-9, THEN A-Z

82
500-33

HAL/S CSECTS (CONTINUED)
(360 ONLY - FSIM TYPE CSECTS)

#FNNNNNN	FSIM CSECT (ALSO CONTAINS LITERALS AND ADCONS)
#TNNNNNN	COST/USE CSECT
	(FC ONLY)
#ZNNNNNN	ZCON CSECT FOR COMSUB OR REMOTE DATA
#ENNNNNN	PROCESS DIRECTORY ENTRY (PROGRAMS AND TASKS ONLY)
#XNNNNNN	CONTROL AREA FOR EXCLUSIVE BLOCK(S)
#RNNNNNN	DATA AREA FOR <u>REMOTE DATA</u>

HAL/S CSECTS (CONTINUED)

53
500-34

IN ADDITION, HAL/S SPECIFIES INFORMATION TO HALLINK (360)
AND THE AP-101 LINKAGE EDITOR THAT ENABLES THE PRODUCTION
OF STACK CSECTS (ONE PER PROCESS).

@ONNNNNN	PROGRAM STACK
@CNNNNNN	TASK STACK
	C = 1-9, THEN A-Z

ALSO, THE COMPILER GENERATES A BLOCK TEMPLATE

@@NNNNNN
AND A SIMULATION DATA FILE (SDF)

##NNNNNN

S3
500-35

HAL/S CSECTS (CONTINUED)

FURTHERMORE, IN AN FC LOAD MODULE THE FOLLOWING TYPES OF CSECTS WILL BE SEEN:

aaNNNNNN	HAL/S LIBRARY ROUTINE OR NON-HAL (E.G. FCOS) ROUTINE. a = (A-Z)
#QNNNNNN	ZCON FOR LIBRARY ROUTINE.
#LNNNNNN	DATA FOR A LIBRARY ROUTINE.
#ØNNNNNN	SECTOR Ø LIBRARY ROUTINE.

9-75

DATA CSECTS

SI
500-36

#PNNNNNN

COMPOOL DATA CSECT

A COMPOOL COMPILATION RESULTS IN THE GENERATION OF A SINGLE CSECT AND ITS CONTENTS ARE THE SAME FOR BOTH FC AND 360.

A #P CONTAINS COMPOOL DECLARED DATA AND USERS CAN CONTROL THE ORDER OF DATA ALLOCATED WITHIN IT BY MEANS OF THE RIGID KEYWORD.

KEY POINTS:

- 1) REPLACE MACROS TAKE UP NO SPACE.
- 2) ENTRY POINTS CAN BE CREATED BY THE USER (VIA THE EQUATE EXTERNAL FACILITY) WHICH ALLOW NON-HAL MODULES TO ACCESS COMPOOL DATA.

EXAMPLE

```
DECLARE ARR ARRAY(10) SCALAR DOUBLE;  
EQUATE EXTERNAL EXTNAME TO ARR$4;
```

AN EQUATE EXTERNAL TAKES UP NO SPACE IN THE #P. WHAT IT DOES IS

DATA CSECTS (CONTINUED)

S2
500-37

TO GENERATE AN ESD RECORD IN THE OBJECT DECK WHICH MARKS THE SPECIFIED VARIABLE AS AN ENTRY POINT (NAMED BY EXTNAME IN THE PREVIOUS EXAMPLE) OF THE CSECT IN WHICH THE VARIABLE IS LOCATED.

THE EQUATE EXTERNAL IS DESIGNED FOR EXTERNAL (NON-HAL) USE ONLY. IT IS ALSO NOT RESTRICTED TO COMPOOL DATA, BUT MAY BE USED WITH ANY DECLARED (STATIC) DATA. THE EQUATE MAY POINT TO ANY PREVIOUSLY DEFINED HAL VARIABLE AND COMPLEX SUBSCRIPTING IS ALLOWED PROVIDED ALL SUBSCRIPTING CAN BE EVALUATED AT COMPILE TIME.

- 3) THE #P (LIKE ALL OTHER HAL/S DATA CSECTS) CONTAINS DATA WHICH IS, IN GENERAL, MODIFIED. AS A RESULT THE ENTIRE #P IS UNPROTECTED EVEN WHEN PORTIONS OF IT CONTAIN CONSTANT DATA. AT RUN-TIME, THEN, CONSTANT DATA IS NO MORE SACRED THAN VARIABLE DATA.

DATA CSECTS (CONTINUED)

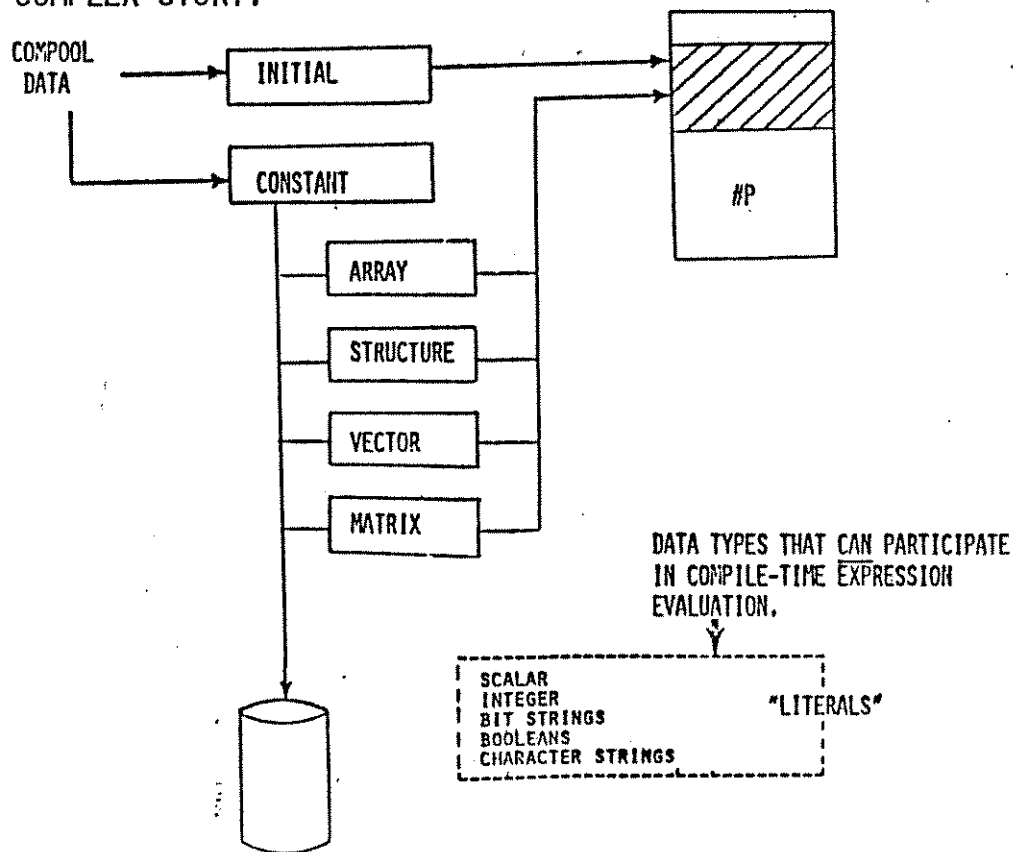
S1
500-38

CONSTANT DATA

(A DIGRESSION)

IN A COMPOOL, ALL DECLARED DATA WITH INITIAL LISTS WILL BE PHYSICALLY ALLOCATED IN THE #P CSECT.

DATA DECLARED WITH A CONSTANT LIST, ON THE OTHER HAND, IS A MORE COMPLEX STORY:



S2
500-39

DATA CSECTS (CONTINUED)

THUS, SIMPLE COMPOOL CONSTANT DATA IS NOT GIVEN SPACE IN THE #P -- SUCH DATA IS OFTEN (INCORRECTLY) CALLED LITERAL DATA.

SINCE ALL OF THE SOURCE LINES OF THE COMPOOL DO GET PUT IN ITS TEMPLATE, HOWEVER, THE ACTUAL VALUES OF THE LITERAL DATA ARE VISIBLE TO ALL COMPILATIONS THAT INCLUDE THE COMPOOL.

NOTE: ALL SUCH LITERAL DATA IS INACCESSIBLE (ALMOST) TO DIAGNOSTICS.

Q. IF LITERALS ARE NOT IN THE #P, WHERE DO THEY GO?

A: WHEN A COMPOOL IS INCLUDED IN A PROGRAM, THE COMPILER SEES BOTH ITS LITERALS PLUS ANY SIMILAR LITERALS THAT MAY HAVE BEEN DECLARED IN THE PROGRAM -- ALL ARE TREATED ALIKE. THEN THE FOLLOWING HAPPENS:

9-79



DATA CSECTS (CONTINUED)

83
500-40

- 1) IF A LITERAL IS NOT USED, IT IS SIMPLY FLUSHED.
- 2) IF THE LITERAL IS USED, IT WILL BE ALLOCATED IN THE #D CSECT (DATA AREA OF THE PROGRAM/COMSUB) UNLESS ITS USE IS SO SPECIALIZED THAT IT ACTUALLY GETS BUILT IN TO AN INSTRUCTION.
- 3) IF THE LITERAL IS ALLOCATED IN THE #D THEN IT MAY:
 - A) BE OF DIFFERENT TYPE (E.G. THE ITEM WAS DECLARED AS AN INTEGER BUT USAGES WERE IN A SCALAR CONTEXT).
 - B) BE OF DIFFERENT PRECISION OR SCALING (E.G. IF PI IS DECLARED CONSTANT(3.14) BUT (2 PI) IS USED, THEN THE VALUE STORED WILL BE 6.28).
 - C) SHARE WITH ANOTHER LITERAL, I.E. TWO LITERALS WITH EXACTLY THE SAME VALUE WILL SHARE THE SAME SPACE.

DATA CSECTS (CONTINUED)

S/
500-41

#DNNNNNN

PROGRAM/COMSUB DATA CSECT

SIMILAR TO #P

- 1) EQUATE EXTERNAL CAN BE USED.
- 2) CONTAINS ALL DATA DECLARE'D IN THE PROGRAM/COMSUB INCLUDING LOCAL DATA OF ALL INTERNAL BLOCKS.

BUT DIFFERENT:

- 1) RIGID CANNOT BE USED.
- 2) MAY CONTAIN A MIXTURE OF BOTH COMPOOL LITERALS (PROVIDED THEY ARE USED) AND ITS OWN LITERALS (AGAIN, ONLY IF USED).
- 3) ALSO CONTAINS SPECIAL DATA NEEDED BY FCOS OR THE COMPILER, E.G.

ADDRESS CONSTANTS

LOCAL BLOCK DATA AREAS

AGAIN, ALTHOUGH INVARIANT DATA MAY RESIDE IN THE #D, IT IS COMPLETELY UNPROTECTED FROM WRITES.

DATA CSECTS (CONTINUED)

SI
500-42

@ONNNNNN	PROGRAM STACKS
@INNNNNN	TASK STACKS
:	
@ZNNNNNN	TASK STACKS

WE HAVE NOT YET DISCUSSED THE TASK BLOCK BECAUSE IT IS VIRTUALLY UNUSED IN THE CURRENT SHUTTLE DESIGN.

FOR THE CURRENT DISCUSSION WE WILL THEREFORE LIMIT ATTENTION TO HAL/S PROGRAMS AND THEIR ASSOCIATED STACK.

Q. WHAT IS A STACK?

A. A STACK IS A SPECIAL DATA CSECT WHICH IS DEDICATED TO A SINGLE PROCESS, I.E. HAL/S PROGRAM. THE STACK HAS DYNAMICALLY CHANGING CONTENTS -- THE SAME LOCATION WILL GENERALLY CONTAIN DIFFERENT DATA AS A FUNCTION OF TIME. THE STACK IS USED FOR THE FOLLOWING PURPOSES:

S2
500-43

DATA CSECTS (CONTINUED)

- (1) REGISTER SAVE AREAS
- (2) PARAMETER PASSING
- (3) SCRATCH SPACE NEEDED BY THE COMPILER
- (4) USER DEFINED TEMPORARIES
- (5) AND OTHER THINGS.

DATA CSECTS (CONTINUED)

33
500-44

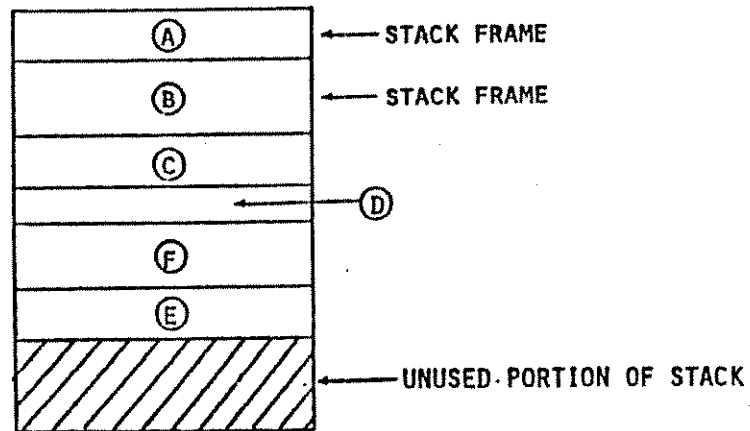
SOME FACTS:

- 1) THE COMPILER NEVER GENERATES A STACK. STACKS ARE EITHER GENERATED AT LINK EDIT TIME (VIA HALLINK ON THE 360 AND THE AP-101 LINKAGE EDITOR ON THE FC) -- OR ARE DOLED OUT BY THE FCOS AT THE TIME A PROGRAM IS INITIATED.
- 2) THE COMPILER DOES INFORM THE LINKAGE EDITOR OF THE SIZE OF THE STACK FRAME REQUIRED BY EACH CODE CSECT. THIS IS DONE VIA SYM CARDS ISSUED IN THE OBJECT DECK.
- 3) A STACK IS DYNAMICALLY DIVIDED INTO STACK FRAMES. EACH CODE BLOCK (GENERALLY) HAS A STACK FRAME PROVIDED FOR IT WHILE IT IS IN EXECUTION. THE STACK FRAME IS ALWAYS THE SAME SIZE FOR A PARTICULAR CODE BLOCK BUT MAY OCCUPY DIFFERENT LOCATIONS IN THE STACK CSECT DEPENDING ON THE CALL CHAIN BY WHICH THE CODE BLOCK WAS REACHED.

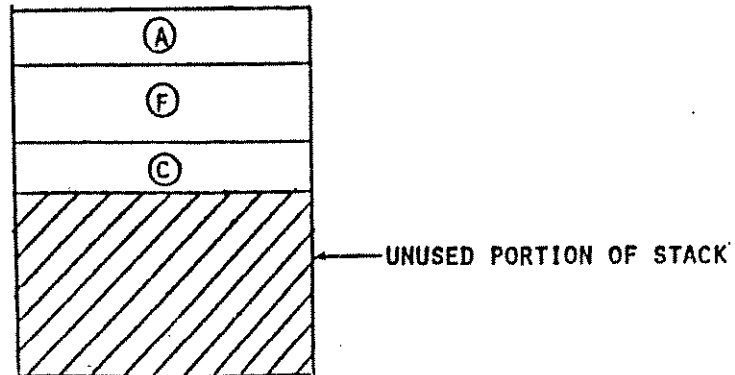
51
500-45

DATA CSECTS (CONTINUED)

A STACK AT ONE POINT IN TIME:



AND LATER:



NOTE: IF THE LINKAGE EDITOR EVER CALCULATES A REQUIRED STACK LENGTH INCORRECTLY, THERE IS NOTHING AT FLIGHT TIME TO KEEP US FROM FALLING OFF THE END.

9-85

DATA CSECTS (CONTINUED)

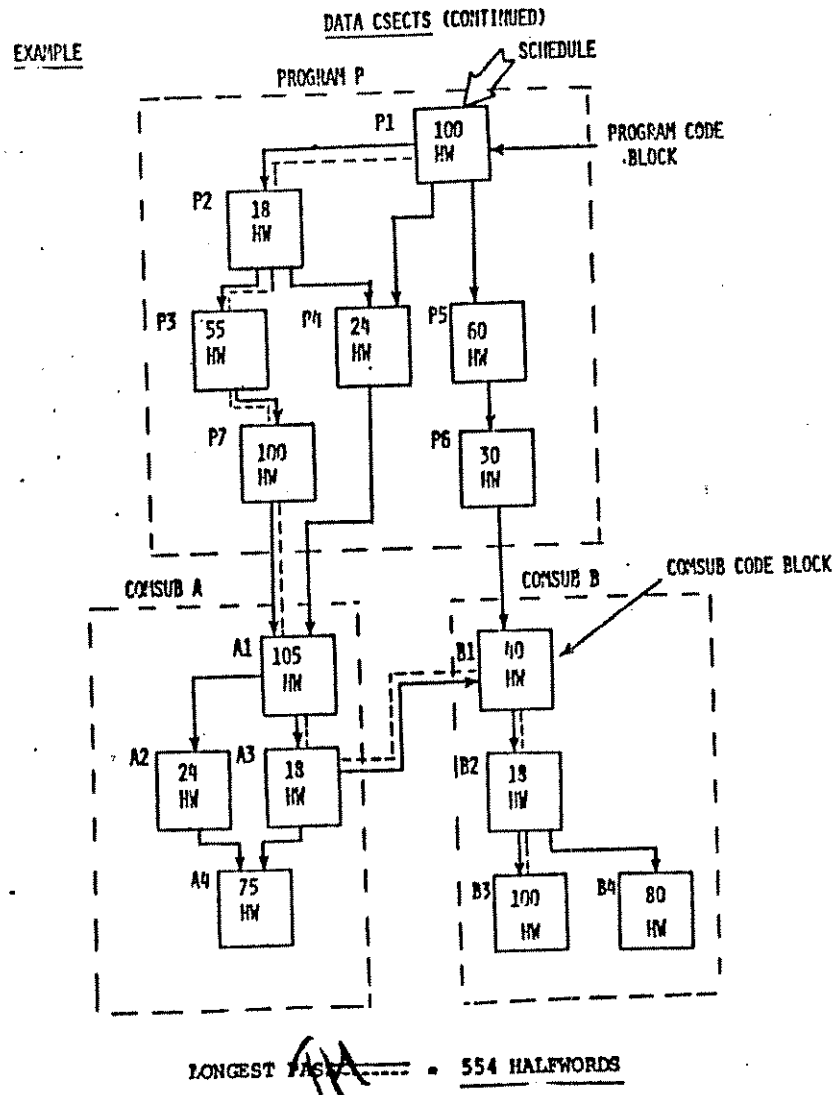
52
500-41

STACK SIZE CALCULATION

- 1) A SPECIAL SYM CARD IS GENERATED (FOR EACH CODE BLOCK) IN THE OBJECT MODULE DEFINING THE SIZE OF THE STACK FRAME REQUIRED BY THE CODE BLOCK (ON THE FC THE MINIMUM SIZE IS 18 HALFWORDS).
- 2) THE LINKAGE EDITOR PERFORMS AN ANALYSIS OF ALL CALLS AND FUNCTION INVOCATIONS (BUILDS A CALL TREE) AND CALCULATES THE SIZE NEEDED FOR EACH PROGRAM (PROCESS) STACK.
- 3) THE LINKAGE EDITOR THEN EITHER EMITS AN APPROPRIATELY NAMED STACK CSECT OF THE RIGHT LENGTH OR INFORMS THE FCOS (VIA THE #E CSECT) OF WHAT SIZE STACK WILL BE NEEDED WHEN THE PROGRAM IS SCHEDULED.

NOTE: ONLY PROGRAMS ARE GIVEN STACKS SINCE THEY ARE PROCESSES.
(TASKS ARE TOO!) ANY INTERNAL OR EXTERNAL BLOCKS (COMSUBS) CALLED BY THE PROGRAM UTILIZE THE STACK OF THE PROGRAM -- AND THE LINKAGE EDITOR HAS TAKEN ALL OF THIS INTO ACCOUNT.

SI
500-47

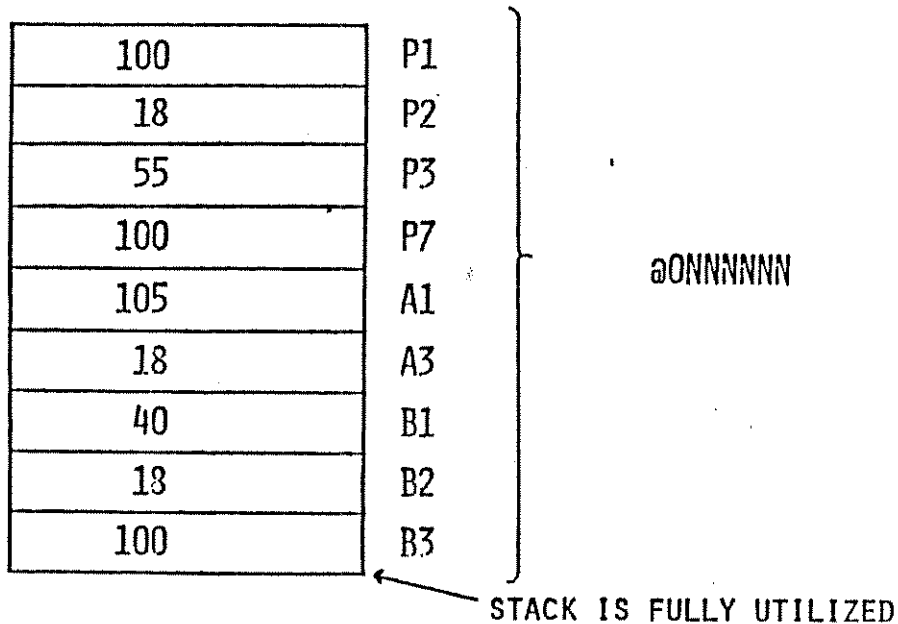


9-87

DATA CSECTS (CONTINUED)

52
500-48

IF WE FOLLOW THE LONGEST PATH THE STACK FRAMES WILL LOOK LIKE:



NOTES:

- 1) IN ACTUAL EXECUTION A STACK WOULD RARELY BE COMPLETELY USED.
- 2) IT IS EASY TO SEE WHY RECURSION IS NOT ALLOWED.

DATA CSECTS (CONTINUED)

53
500-49

STACKS SAVE CORE BECAUSE ...

- 1) LOCAL (TRANSIENT) SPACE NEEDS WITHIN A SINGLE PROCESS ARE SATISFIED FROM A REUSABLE AREA.
- 2) DEPENDING ON THE FCOS IMPLEMENTATION, STACKS THEMSELVES CAN SHARE THE SAME STORAGE IF THE PROCESSES CAN NEVER BE SIMULTANEOUSLY ACTIVE.

USERS CAN PUT SCRATCH DATA IN THE STACK VIA THE TEMPORARY FACILITY. THIS WOULD ONLY INCREASE THE SIZE OF THE STACK NEEDED IF THE CORE BLOCK INVOLVED IS IN THE MAXIMAL CHAIN -- THAT IS, THE GREATEST AMOUNT OF STACK SPACE.

THE HAL/S STACK MECHANISM

THE HAL/S RUNTIME STACK PROVIDES:

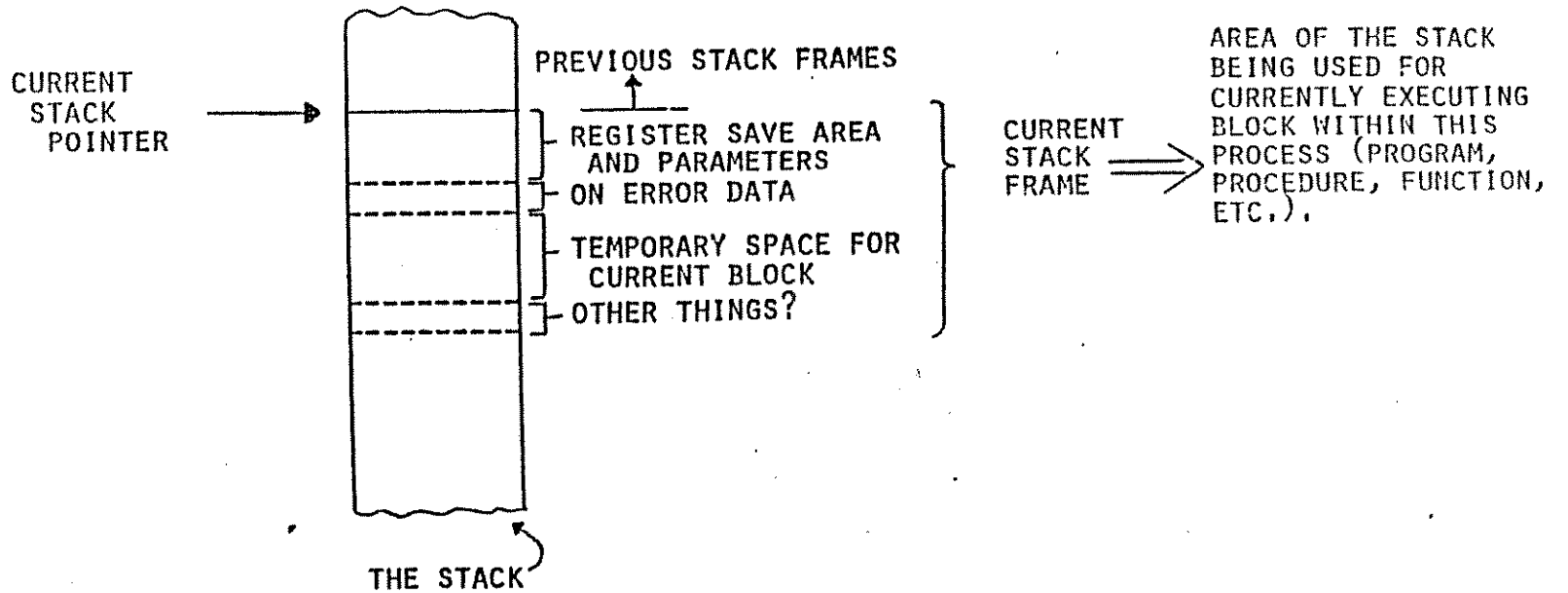
- A SIMPLE MEANS FOR SAVING AND RESTORING ENVIRONMENTS FOR CALLS.
- A DYNAMIC RECORD OF ERROR ENVIRONMENTS.
- EFFICIENT TEMPORARY STORAGE ALLOCATION (COMPILER AND USER)
- THE MEANS BY WHICH SEPARATE PROCESSES MAY SIMULTANEOUSLY EXECUTE THE SAME CODE BLOCK AND YET MAINTAIN INDEPENDENT DATA.

EACH HAL/S PROCESS HAS ITS OWN STACK.

EACH STACK IS A CONTIGUOUS AREA OF MEMORY LARGE ENOUGH TO SERVE ITS PROCESS' NEEDS.

THE HAL/S STACK MECHANISM (CONTINUED)

A SINGLE PROCESS STACK

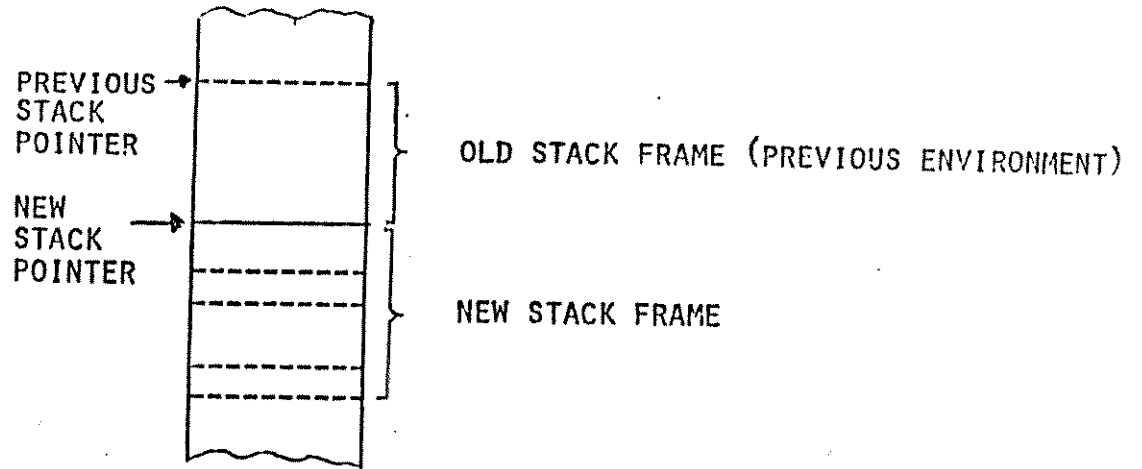


MISC 12

THE CALL/S STACK MECHANISM (CONTINUED)

WHEN A JUMP TO A NEW BLOCK IS MADE:

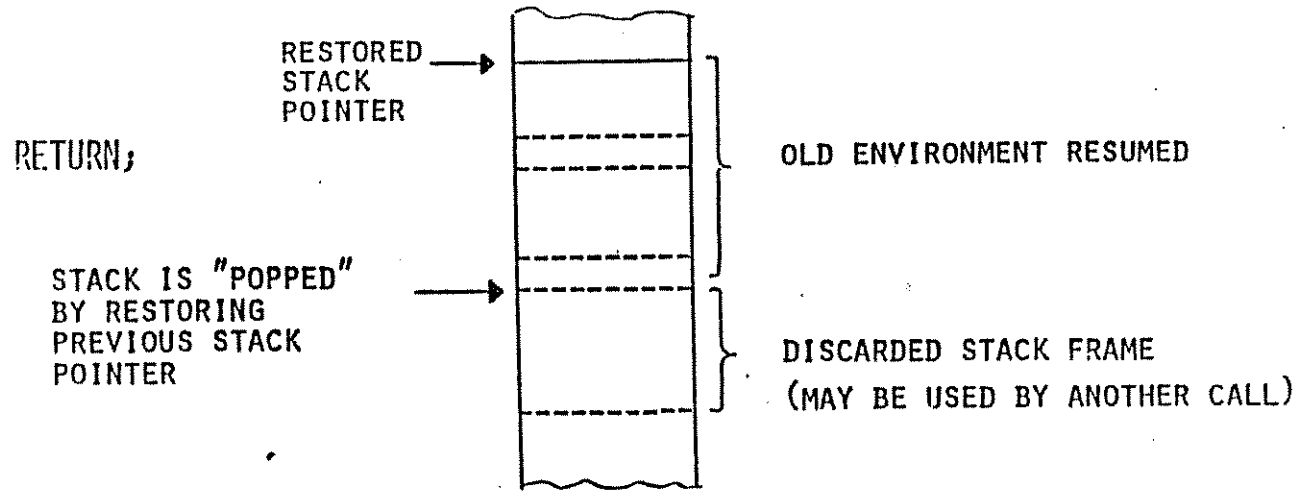
CALL X(A) ASSIGN(B);



- THE STACK AREA ADDRESS IS INCREMENTED BY THE SIZE OF THE CURRENT STACK FRAME.
- A NEW STACK FRAME IS USED FOR THE NEWLY ENTERED ROUTINE.
- THE NEW STACK FRAME CONTAINS ALL THE INFORMATION NECESSARY TO RESTORE THE PREVIOUS ENVIRONMENT.

THE HAL/S STACK MECHANISM (CONTINUED)

WHEN THE CALLED ROUTINE IS FINISHED:



MIS 1-4

HAL/S STACK MECHANISM (CONTINUED)

- DATA DECLARED TO BE STATIC IN THE CURRENT BLOCK IS NOT ALLOCATED IN THE STACK AREA.
- REENTRANT PROCEDURES HAVE ANY AUTOMATIC DATA ALLOCATED IN THE STACK.
- ANY TEMPORARY DATA EITHER DECLARED (TEMPORARY STATEMENT) BY THE PROGRAMMER OR IMPLICITLY NEEDED BY THE COMPILER IS IN THE STACK AREA.
- THE COMPILER KNOWS EXACTLY HOW MUCH STACK AREA EACH CODE BLOCK REQUIRES THAT SIZE INFORMATION IS PLACED IN THE OBJECT DECK.
- THE HALLINK PROGRAM BUILDS A TREE OF ALL BLOCK REFERENCES.
 - RECURSION IS DETECTED HERE,
 - BY ADDING STACK REQUIREMENTS FOR EACH LIMB OF THE TREE, THE CALLING SEQUENCE WHICH REQUIRES THE MOST TOTAL STACK AREA IS FOUND.
 - A STACK OF PROPER SIZE IS CREATED AND MADE PART OF THE PROGRAM,
 - A SEPARATE CALCULATION IS MADE FOR EACH POTENTIAL PROCESS (PROGRAM OR TASK).

THE HAL/S STACK MECHANISM (CONTINUED)

- THE MECHANIZATION OF THE STACK REQUIRES AN INTERFACE BETWEEN THE OPERATING SYSTEM AND THE GENERATED CODE.
 - HOW IS STACK LOCATION ESTABLISHED? (PASSED AT INITIATION OR LOADED BY CODE.)
 - WHAT IS THE STACK LAYOUT? (NEEDED FOR ERROR HANDLING.)
 - HOW ARE STACK FRAMES CHAINED TOGETHER? (STACK WALK.)
 - OTHER INTERFACES.

MISC 1-6

ASSEMBLY LANGUAGE ROUTINES MUST BE INTERFACED TO THE HAL/S
SYSTEM

- OBJECT MODULE (RESULT OF ASSEMBLY) MUST BECOME PART OF HALLINK INPUT.
 - HAND-WRITTEN TEMPLATE MUST BE INCLUDED IN COMPILATION OF HAL/S CALLER.
-

```
// EXEC HALSCLD
//HAL/SYSIN DD *
  ASMSUB: EXTERNAL PROCEDURE(A);
  DECLARE A INTEGER;
  CLOSE AMSUB;
  TEST: PROGRAM;
      :
      CALL ASMSUB(23);
      :
  CLOSE TEST;
//LKED.SYSIN DD <assembly object module>
```

Misc 1-7

ASSEMBLY LANGUAGE ROUTINES MUST OBEY HAL/S LINKAGE CONVENTIONS

ASSEMBLY LANGUAGE MACROS ARE PROVIDED TO HELP.

#CASMSUB HMAIN

⋮

(ACCESS ARGUMENTS IN REGISTERS OR VIA R13)

⋮

HCALL HALSUB

⋮

HEXIT

⋮

END

Misc 1-8

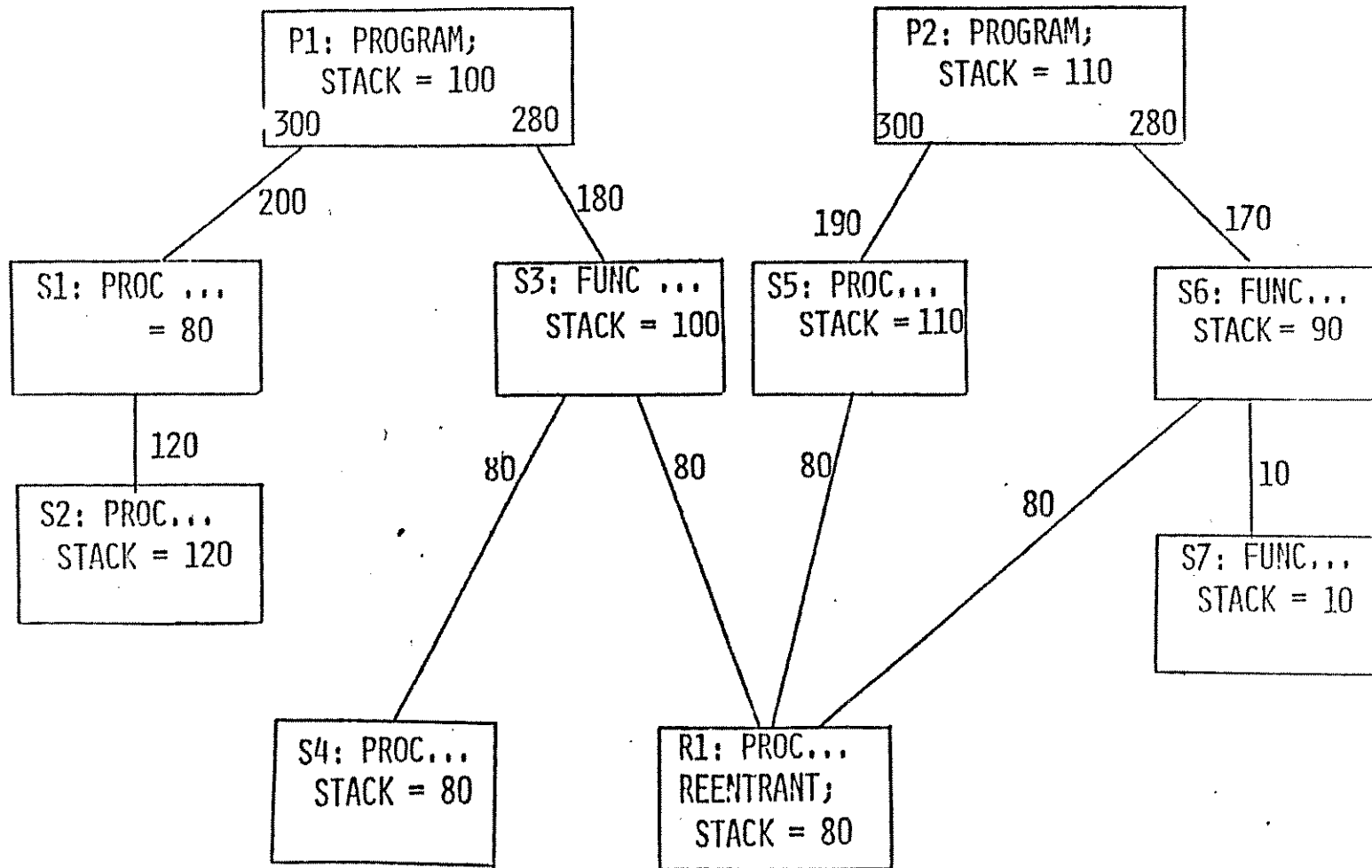
HAL/S-360 LINKAGE CONVENTIONS

R12 —→ SYSTEM INTRINSICS (E.G. PROCEDURE CALLER,
PROCEDURE EXITER)
R13 —→ CURRENT STACK FRAME
R14 —→ RETURN ADDRESS
R15 —→ ENTRY POINT ADDRESS
F0, F2, F4 —→ 1ST THREE SCALAR ARGUMENTS
R0, R1, R2, R3, R4 —→ 1ST FIVE INTEGER OR BIT ARGUMENTS ... OR ... POINTERS
TO ARGUMENTS OF OTHER DATA TYPES.

EXAMPLE:

```
CALL SUB1(A, B, C) ASSIGN(D);  
LH   R0, A           A INTEGER  
LA   R1, B           B VECTOR  
LE   F0, C           C SCALAR  
LA   R2, D           D BIT(16)  
BALR R14, R12        GO TO PROCEDURE CALLER  
DC   A(SUB1)         ENTRY POINT ADDRESS
```


HALLINK STACK CALCULATION

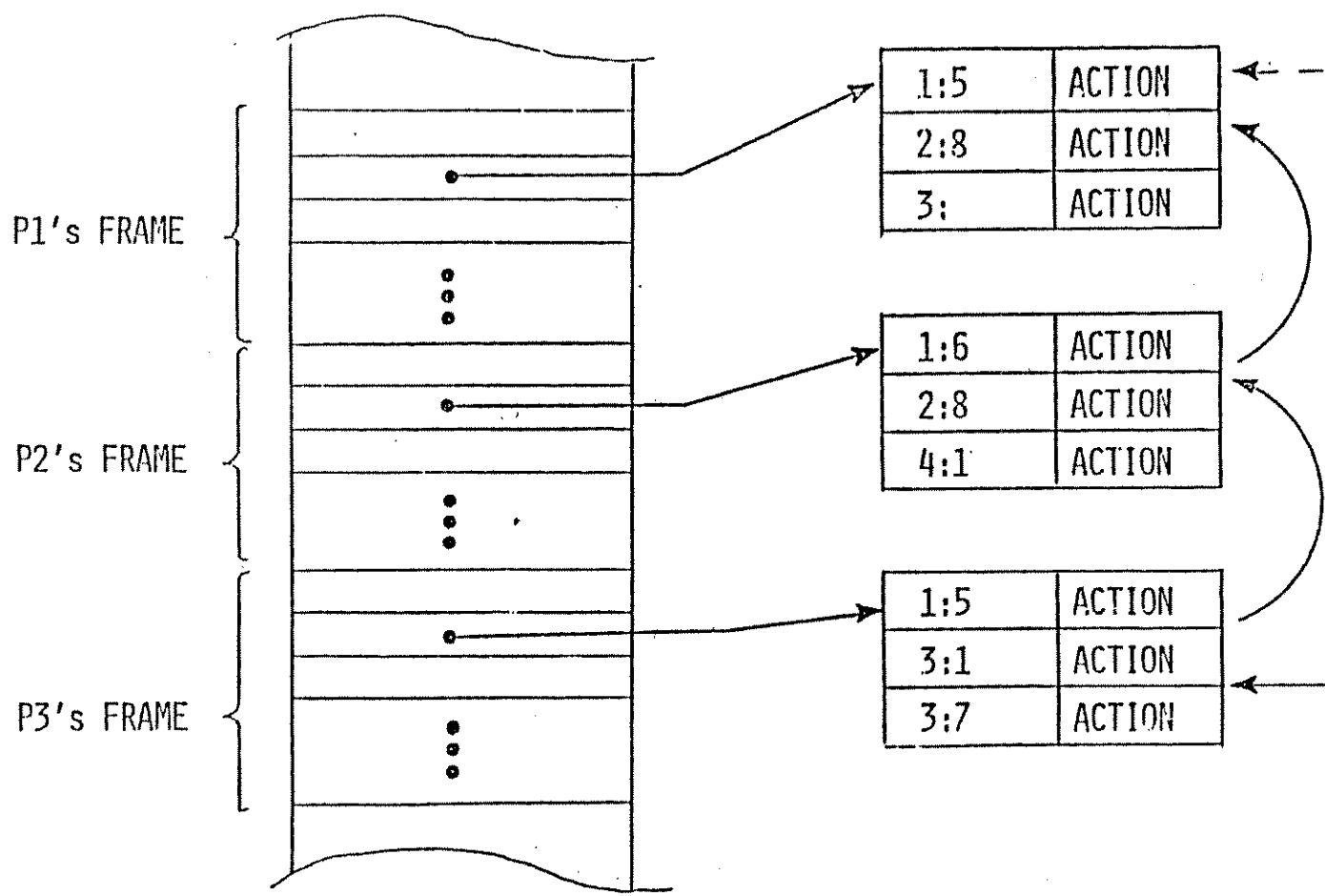


ACTION: CREATE SEPARATE STACKS OF LENGTH 300 FOR BOTH P1 AND P2.

Misc 1-10

THE ERROR ENVIRONMENT IS REPRESENTED ON THE STACK

INCLUDES "ACTIVE"
BIT - TURNED ON BY
- ON ERROR, OFF BY
OFF ERROR



ERROR RECOVERY
EXECUTIVE SEARCHES
UP THE STACK FOR
ERROR HANDLING
SPECIFICATIONS

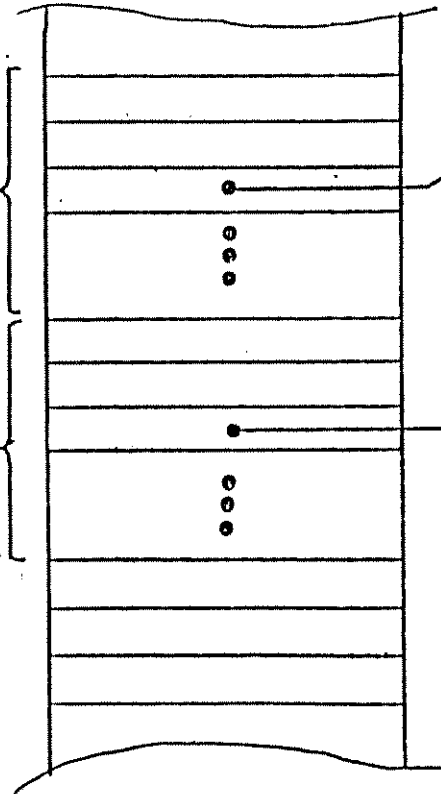
Misc 1-11

EXECUTION OF UPDATE BLOCKS AND EXCLUSIVE PROCEDURES IS
ALSO REPRESENTED ON THE STACK.

*Now using
stack variables*

UPDATE BLOCK
STACK FRAME

EXCLUSIVE
PROCEDURE
STACK FRAME



TO OPERATING SYSTEM REPRESENTATION
OF LOCK GROUPS

TO OPERATING SYSTEM REPRESENTATION
OF EXCLUSIVE PROCEDURE LOCKS

FOR TERMINATE AND ON ERROR
<statement>, MUST "PEEL BACK"
THE STACK AND FREE LOCK GROUPS
AND EXCLUSIVE PROCEDURES.

MISC 1-12

ERROR RECOVERY FEATURES

ON ERROR } *not used*
OFF ERROR }
* SEND ERROR ← *used*

- USED FOR SPECIAL HANDLING OF UNUSUAL CONDITIONS.
- SYSTEM-DEFINED ERRORS
ARITHMETIC OVERFLOW, END OF FILE, ETC.
- USER-DEFINED ERRORS
SIGNALLED WITH SEND ERROR STATEMENT
SYSTEM-DEFINED ERRORS CAN ALSO BE SIMULATED

ON ERROR_{m:n} <statement>;

- WHEN THE ERROR OCCURS, EXIT ANY CALLED BLOCKS, EXECUTE THE <statement>, AND CONTINUE WITH THE STATEMENT FOLLOWING THIS ONE IN THE PROGRAM.

ON ERROR_{m:n} IGNORE;

- CONTINUE FROM THE POINT WHERE THE ERROR OCCURRED, USING A "STANDARD SYSTEM FIXUP" FOR THE ERROR.

ON ERROR_{m:n} SYSTEM;

- A STANDARD ACTION (USUALLY PRINT A MESSAGE AND USE THE "STANDARD SYSTEM FIXUP" IS TAKEN.

MISC 1-14

ON ERROR_{m:n} { IGNORE
SYSTEM }

AND { SET
SIGNAL } <event> ;
{ RESET }

ok SEND ERROR_{m:n} ;

(SIMULATE A SYSTEM- OR USER-DEFINED ERROR)

MISC 1-15

- EACH ERROR IS ASSIGNED A GROUP NUMBER AND A NUMBER WITHIN THE GROUP.
- EACH BLOCK CAN HANDLE A CERTAIN SET OF ERRORS (ON ERROR SOMEWHERE IN THE BLOCK).
- HANDLING OF ERRORS CAN BE ACTIVATED AND DEACTIVATED DYNAMICALLY (THROUGH EXECUTION OF ON/OFF ERROR STATEMENTS).

THE ERROR ENVIRONMENT IS ESTABLISHED THROUGH THE DYNAMIC NESTING OF CALLS

P1: PROCEDURE;
HANDLES ERRORS
1:5, 2:8, 3:

CALL P2;

P2: PROCEDURE;
HANDLES ERRORS
1:6, 2:8, 4:1

CALL P3;

P3: PROCEDURE;
HANDLES ERRORS
1:5, 3:1, 3:7

BEFORE AND AFTER
CALL P2; ERROR 2:8
IS HANDLED HERE.

DURING P2 AND P3
ERROR 2:8 IS HANDLED
HERE.

ERRORS 3:1 AND 3:7
HANDLED HERE; OTHERS
IN GROUP 3 HANDLED IN
P1.

Parts of Halts not used

TASKS

update blocks

Block groups

Mainy Real-Time Starts

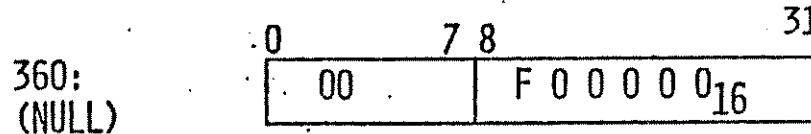
Dependant processes

MISC 1-17

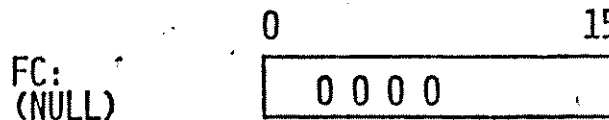
NAME VARIABLES

Q. WHAT IS IN A NAME?

A. AN ADDRESS (POINTER).



ILLEGAL ADDRESS



NO ADDRESSES ARE ILLEGAL ON AP-101

*Name Remote
Needs 32 bits*

NOTE: FC NAME VARIABLES ARE ALWAYS HALFWORDS AND 360 NAME VARIABLES ARE ALWAYS FULLWORDS (AND FULLWORD ALIGNED). THIS IS THE ONE FUNDAMENTAL INCONSISTENCY BETWEEN FC AND 360 STORAGE ALLOCATIONS AND IS NOT SURMOUNTABLE BY USE OF THE FCDATA OPTION.

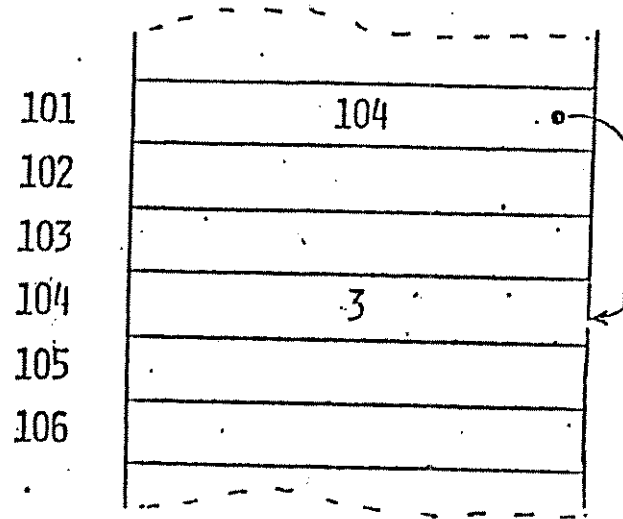
NAME VARIABLES (CONTINUED)

PROPERTIES:

- NAME VARIABLES ARE A CLASS OF DATA ITEMS WHOSE VALUES ARE POINTERS (ADDRESSES) TO OTHER DATA ITEMS.
- A NAME VARIABLE IS ESTABLISHED BY DECLARING IT AS THOUGH IT WERE A DATA ITEM EXCEPT THAT THE KEYWORD NAME IS USED.
- A NAME VARIABLE CAN ONLY POINT AT DATA THAT MATCHES IT IN TYPE, PRECISION, ARRAYNESS, ETC.
- A MECHANISM EXISTS FOR INITIALIZING A NAME VARIABLE TO POINT TO A GIVEN DATA ITEM. NAME-VARIABLES MAY ALSO BE DYNAMICALLY "RE-DIRECTED" TO OTHER DATA ITEMS.
- DATA ITEMS MAY BE MANIPULATED IN THE NORMAL WAYS THROUGH REFERENCES TO NAME VARIABLES POINTING TO THEM.

NAME VARIABLES (CONTINUED)

HARDWARE LEVEL



HAL/S LEVEL

```
DECLARE I INTEGER INITIAL(3),  
        NI NAME INTEGER INITIAL (NAME(I));
```

```
NI = NI + 1;
```

```
/* SAME AS I = I + 1; */
```

Automatic Reference

NAME VARIABLE (CONTINUED)

USES FOR NAME VARIABLES

- MOVE AROUND A POINTER TO A BLOCK OF DATA (OR CODE) RATHER THAN THE WHOLE BLOCK.
- ACCOMMODATE DATA STRUCTURES OF DYNAMICALLY VARYING SIZE IN A FIXED-SIZE MEMORY.

PROBLEMS WITH UNRESTRICTED POINTERS

```
DECLARE I INTEGER;  
        S SCALAR,  
        N NAME;  
NAME(N) = NAME(I);          /* POINT N AT I */  
L: N = N + 1;                /* WHAT CODE IS COMPILED? */  
NAME(N) = NAME(S);          /* POINT N AT S */  
IF N < S THEN GO TO L;      /* CONVERSION REQUIRED? */
```

error illegal

NAME VARIABLES (CONTINUED)

SO, A NAME VARIABLE IS DECLARED TO POINT TO VARIABLES
OF A GIVEN DATA TYPE ...

DECLARE NI NAME INTEGER,

NS NAME SCALAR,

NV NAME VECTOR(4),

NA NAME ARRAY(2,2) BOOLEAN;

DECLARE INTEGER DOUBLE,

I, J, K,

N NAME;

/* FACTORED ATTRIBUTES */

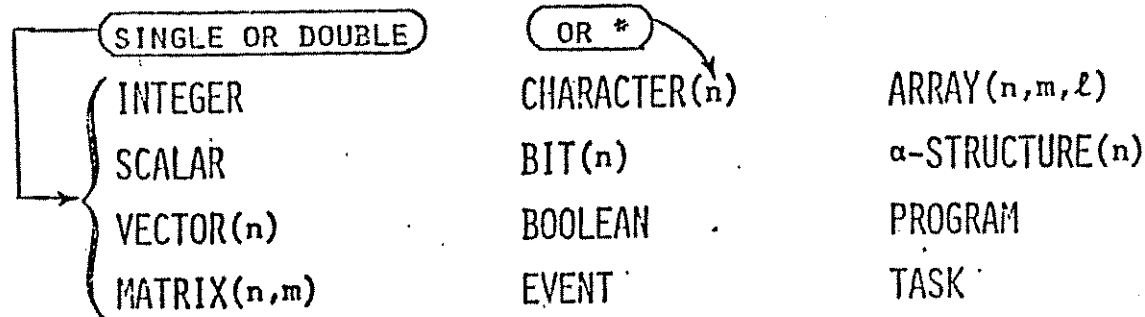
NAME VARIABLES (CONTINUED)

HENCE THE COMPILER CAN CHECK FOR TYPE COMPATIBILITY.

```
DECLARE I INTEGER,  
        S SCALAR,  
        N NAME INTEGER;  
NAME(N) = NAME(I);           /* POINT N AT I */  
L: N = N + 1;                /* CODE FOR INTEGER ADDITION */  
NAME(N) = NAME(S);          /* ERROR-TYPE MISMATCH */  
IF N < S THEN GO TO L;     /* N DEFERENCED AND CONVERTED  
                           TO SCALAR */
```

NAME VARIABLES (CONTINUED)

NAME VARIABLES CAN POINT TO DATA (OR CODE) WITH
THE FOLLOWING ATTRIBUTES:



```
DECLARE MATRIX(2,3), A, B, NB NAME INITIAL(NAME(B));
DECLARE MATRIX(3,3), C, D, ND NAME INITIAL(NAME(D));
A = NB; /* SAME AS A = B; */
A = ND; /* ERROR-DIMENSION MISMATCH */
NAME(NB) = NAME(ND); /* ERROR-DIMENSION MISMATCH */
NB = ND$(2 AT 1,*); /* SAME AS B = D$(2 AT 1*); */
```

NAME VARIABLES (CONTINUED)

NAME DATA ITEMS POINTING TO DATA

Declarations of NAME data items for pointing to data have exactly the same form as declarations of ordinary data items, except that the keyword NAME immediately follows the identifier name declared.

Examples:

```
DECLARE A NAME ARRAY(100) SCALAR;  
DECLARE MATRIX(3,3) DOUBLE, M1 NAME, M2 NAME;  
DECLARE B NAME BIT(16),  
        C NAME CHARACTER(80);  
STRUCTURE Q:  
    1 QI INTEGER,  
    1 QS SCALAR,  
    1 QI,  
    2 QH BIT(16),  
    2 QC CHARACTER(80);  
DECLARE ZQ NAME Q-STRUCTURE;
```

Given the above declarations:

A may only point to 1-dimensional single precision scalar arrays of size 100.

M1, M2 may only point to 3x3 double precision matrices.

B may only point to 16-bit strings.

C may only point to character strings of maximum length 80.

ZQ may only point to Q-STRUCTURES with a single copy.

NAME VARIABLES (CONTINUED)

NAME DATA ITEMS POINTING TO CODE BLOCKS

Declarations of NAME data items for pointing to programs and tasks have the following basic form:

```
DECLARE name NAME PROGRAM;  
DECLARE name NAME TASK;  
  
1. name is any legal HAL/S identifier  
name.
```

Such declarations can be part of a compound or factored declaration statement.

Examples:

```
DECLARE P1 NAME PROGRAM;  
DECLARE T1 NAME TASK;  
DECLARE P2 NAME PROGRAM,  
T2 NAME TASK,  
S1 NAME SCALAR;
```

Given the above declarations:

P1, P2 may only point to program blocks.
T1, T2 may only point to task blocks.

NAME VARIABLES (CONTINUED)

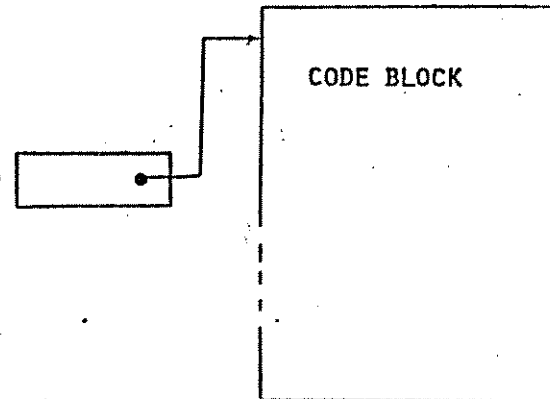
POINTERS TO CODE BLOCKS

```
P1: EXTERNAL PROGRAMJ  
CLOSE P1J  
P2: EXTERNAL PROGRAMJ  
CLOSE P2J  
P3: EXTERNAL PROGRAMJ  
CLOSE P3J
```

TEMPLATES

```
MASTER: PROGRAMJ  
  DECLARE NP NAME PROGRAM,  
           I INTEGERJ  
  :  
  DO WHILE TRUEJ  
    DO FOR I = 1 TO 3J  
      DO CASE IJ  
        NAME(NP) = NAME(P1)J  
        NAME(NP) = NAME(P2)J  
        NAME(NP) = NAME(P3)J  
      ENDJ  
      UPDATE PRIORITY NP TO 200J  
      WAIT 1J  
      UPDATE PRIORITY NP TO 50J  
    ENDJ  
  ENDJ  
  :  
  CLOSE MASTERJ
```

NP



```
/* PROMOTE P1, P2, P3 IN TURN */  
/* TO HIGH PRIORITY FOR 1 SEC */  
/* THEN REVERT TO NORMAL PRI. */
```

NAME VARIABLES (CONTINUED)

NAME DATA ITEMS AS STRUCTURE TERMINALS

Examples:

```
STRUCTURE O:  
  1 QS NAME SCALAR,  
  1 Q1,  
  2 QC NAME CHARACTER(80),  
  2 QR NAME PROGRAM,  
  2 QP NAME BOOLEAN,  
  1 Q2,  
  2 QA ARRAY(4) BIT(16);
```

Note that NAME data items for pointing to events can appear in a structure template, even though events themselves cannot. Note also that NAME data items in a template A may point to structures, even those possessing A as template.

NAME VARIABLES (CONTINUED)

Examples:

The following are legal definitions:

```
STRUCTURE R:  
  1 QR NAME R-STRUCTURE,  
  1 QE NAME EVENT;  
DECLARE ZR R-STRUCTURE;  
DECLARE NZR NAME R-STRUCTURE;
```

In this example NZR may point to ZR. ZR.QR may also point to ZR. The implications of this ability will be investigated later.

NAME VARIABLES (CONTINUED)

NAME DATA ITEMS AND TEMPORARIES

The nature and purpose of temporary data items were described in Section 26.3. The following rules summarize relationships between temporary data items and NAME data items.

1. No NAME data item may point to a temporary data item.
2. NAME data items may not themselves be declared as temporary data items.

EXAMPLE:

```
DO,  
    TEMPORARY INTEGER, I, J,  
ILLEGAL → TEMPORARY NV NAME VECTOR,  
    . . .  
END;
```

NAME VARIABLES (CONTINUED)

INDIRECT ACCESS THROUGH NAME DATA ITEMS

Examples:

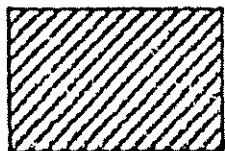
```
DECLARE VECTOR(3), V, NV NAME;  
DECLARE SCALAR, S, NS NAME;  
DECLARE NT NAME TASK;
```

.

.

.

```
T: TASK;
```



} task body

```
CLOSE T;
```

If $NV \rightarrow V$, $NS \rightarrow S$ and $NT \rightarrow T^*$, then

```
NS = NV.NV;  
SCHEDULE NT IN NS PRIORITY(50);
```

effectively performs the operations:

```
S = V.V;  
SCHEDULE T IN S PRIORITY(50);
```

* In this and following examples " \rightarrow " means "points to".

NAME VARIABLES (CONTINUED)

The foregoing statements about appearances of NAME data items, while appearing simple and unequivocal, contain a number of subtle implications arising from:

- interactions in structure data items;
- the effects of subscripting.

NAME VARIABLES (CONTINUED)

INDIRECT ACCESSING AND STRUCTURES

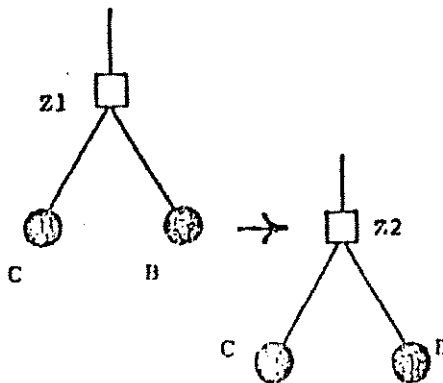
The subtleties of indirect accessing in conjunction with structures arise as a consequence of these two facts:

- Any structure may possess NAME structure terminals some of which may point to structure data items.
- Such a NAME structure terminal can actually point back to the structure containing it.

These subtleties are best illustrated by the extended examination of an apparently very simple example. By the rules given in Section 20.2, the following are legal structure declarations:

```
STRUCTURE A:  
  1 C SCALAR,  
  1 B NAME A-STRUCTURE;  
DECLARE A-STRUCTURE, Z1, Z2, Z3;  
DECLARE Z4 NAME A-STRUCTURE;
```

Z1.B is a NAME structure terminal of A-STRUCTURE type, which may therefore legally point to Z2. Pictorially:



NAME VARIABLES (CONTINUED)

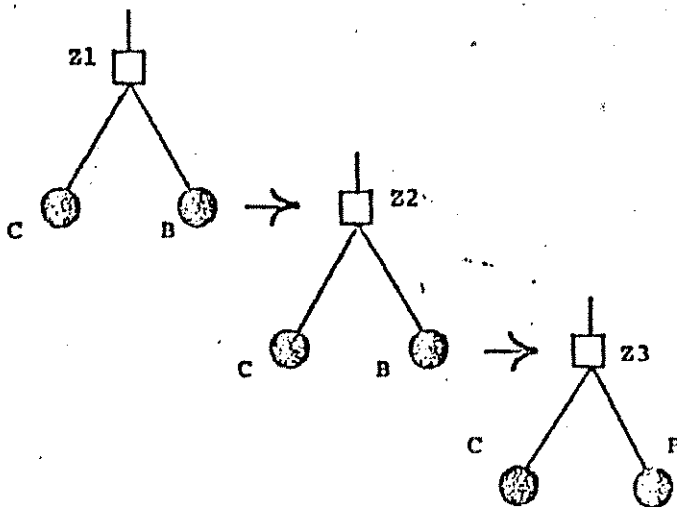
Because Z1.B points to Z2, any appearance of Z2 may be substituted by Z1.B, so achieving indirect access to Z2.

It is crucially important at this point to understand that because Z1.B points to Z2, parts of Z2 as well as Z2 itself may be indirectly accessed. For example, to achieve indirect access to Z2.C, the appearance of Z2 in the qualified name is substituted by Z1.B. That is, indirect access to Z2.C is achieved by the qualified form Z1.B.C.

NAME VARIABLES (CONTINUED)

To illustrate this substitution process further, if Z4 points to Z2, then Z2.C is indirectly accessed by the qualified form Z4.C, and if Z4 points to Z1, then Z2.C is indirectly accessed by the qualified form Z4.B.C.

Multiple levels of indirection are handled in the same way. Suppose for example that in addition Z2.B points to Z3. Then pictorially:



NAME VARIABLES (CONTINUED)

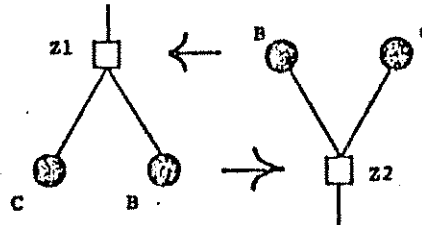
Using the same kind of substitution as before, Z3 may be indirectly accessed by the qualified form Z1.B.B, so that in its turn, structure terminal C in Z3 may be indirectly accessed by the qualified reference Z1.B.B.C.

Restating how the form Z1.B.B.C was arrived at, the following steps were taken:

- substitution of Z2.B.C for Z3.C (since Z2.B points to Z3);
- substitution of Z1.B.B.C for Z2.B.C (since Z1.B points to Z2).

NAME VARIABLES (CONTINUED)

There are other curious consequences arising from the interaction of indirect accessing with structures. Suppose now, for example, that Z2.B points to Z1 rather than Z3. Then, pictorially:



Now Z2.C can be indirectly accessed by the qualified form Z1.B.C, since Z1.B points to Z2. Since Z2.B points to Z1, the following forms are also possible:

Z2.B.B.C
Z1.B.B.B.C
Z2.B.B.B.B.C
Z1.B.B.B.B.B.C

⋮

This example illustrates the logical consequence of a closed indirection loop between two structures.

NAME VARIABLES (CONTINUED)

```
1 N↑ ALPHA;  
1 N↑ PROGRAM;  
2 N↑   STRUCTURE A:  
2 N↑     1 C SCALAR,  
2 N↑     1 B NAME A-STRUCTURE;  
3 N↑   DECLARE A-STRUCTURE,  
3 N↑     Z1, Z2, Z3;  
4 N↑   DECLARE Z4 NAME A-STRUCTURE;  
E↑     +  
5 N↑   NAME(Z1. B) = NAME(Z2);  
6 N↑   Z2. C = 5;  
7 N↑   Z1. B. C = 6;  
8 N↑   WRITE(6) Z2. C, Z1. B. C;  
E↑     +  
9 N↑   NAME(Z2. B) = NAME(Z3);  
10 N↑  Z3. C = 7;  
11 N↑  Z2. B. C = 8;  
12 N↑  Z1. B. B. C = 9;  
13 N↑  WRITE(6) Z3. C, Z2. B. C, Z1. B. B. C;  
14 N↑  CLOSE;
```

```
CALL NAME2  
TEMPNAME ASSUMED AS A MEMBER NAME  
HAL/S-360 V15.0 START TIME: 22:34:02.23 DAY: 76/075  
6.0000000E+00      6.0000000E+00  
9.0000000E+00      9.0000000E+00      9.0000000E+00
```

NAME VARIABLES (CONTINUED)

000009		ST#5	EQU	*	
00009 EC21	0008		LA	4,8(1)	Z2
0000A 8C19	000E		STH	4,6(1)	Z1+2
000000B		ST#6	EQU	*	
0000B 88E5			LFLI	0,5	
0000C 7811	0008		STE	0,8(1)	Z2
000000D		ST#7	EQU	*	
0000D 9A19	0006		LH	2,6(1)	Z1+2
0000E 8AE6			LFLI	2,6	
0000F 7A02	0000		STE	2,0(2)	
0000010		ST#8	EQU	*	
00010 BEE8			LFXI	6,6	
00011 BDE5			LFXI	5,3	
00012 D0FB 0000			SCAL	0,0(3)	IOINIT
00014 7811	0008		LE	0,8(1)	Z2
00015 D0FB 0000			SCAL	0,0(3)	EOUT
00017 9A19	0006		LH	2,6(1)	Z1+2
00018 7802	0008		LE	0,0(2)	
00019 D0FB 0000			SCAL	0,0(3)	EOUT
000001B		ST#9	EQU	*	
0001B EC31	000C		LA	4,12(1)	Z3
0001C BC29	000A		STH	4,10(1)	Z2+2
000001D		ST#10	EQU	*	
0001D 88E7			LFLI	0,7	
0001E 8819	000C		STE	0,12(1)	Z3
000001F		ST#11	FOH	*	
0001F 9A29	000A		LH	2,10(1)	Z2+2
00020 8CE8			LFLI	4,8	
00021 3C02	0008		STE	4,0(2)	
0000022		ST#12	EQU	*	
00022 9A19	0006		LH	2,6(1)	Z1+2
00023 9A0A	0002		LH	2,2(2)	
00024 8EE9			LFLI	6,9	
00025 3E02	0000		STE	6,0(2)	
0000026		ST#13	EQU	*	
00026 BEE8			LFXI	6,6	
00027 BDE5			LFXI	5,3	
00028 D0FB 0000			SCAL	0,0(3)	IOINIT
0002A 7819	000C		LE	0,12(1)	Z3
0002B D0FB 0000			SCAL	0,0(3)	EOUT
0002C 8828	000A		LH	2,10(1)	Z2+2

NAME VARIABLES (CONTINUED)

```
1 N↑ BETA:
1 N↑ PROGRAM:
2 N↑   STRUCTURE A:
2 N↑     1 C SCALAR,
2 N↑     1 B NAME A-STRUCTURE)
3 N↑   DECLARE A-STRUCTURE.
3 N↑     Z1, Z2;
   E↑     +
4 N↑   NAME(Z1, B) = NAME(Z2);
   E↑     +
5 N↑   NAME(Z2, B) = NAME(Z1);
6 N↑   Z1, B, C = 1;
7 N↑   Z2, B, B, C = -2;
8 N↑   Z1, B, B, B, C = 3;
9 N↑   Z2, B, B, B, B, C = 4;
10 N↑   Z2, B, B, B, B, B, C = 5;
11 N↑   Z1, B, B, B, B, B, B, C = 6;
12 N↑   Z2, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B, B, C = 20;
13 N↑ CLOSE;
```


NAME VARIABLES (CONTINUED)

INDIRECT ACCESS AND SUBSCRIPTING

In this discussion, for simplicity, subscripting in connection with structures or structure terminals will at first be excluded. With this restriction, subscripting on NAME data items is straightforward in its meaning.

Subscripting is effective on the data item that is being indirectly accessed.

With this interpretation, it is clear that such subscripts must be legal for the data type pointed to. In particular, NAME data items pointing to programs and tasks may not be subscripted.

NAME VARIABLES (CONTINUED)

Examples:

```
DECLARE VECTOR(3), V, NV NAME;  
DECLARE ARRAY(2) CHARACTER(4), C, NC NAME;  
DECLARE BIT(4), B, NB NAME;
```

Let $V \equiv \begin{bmatrix} 0.5 \\ 1.5 \\ 2.5 \end{bmatrix}$, $C \equiv ('ABCD' 'EFGH')$, $B \equiv 1010_2$

Then if $NV \rightarrow V$, $NC \rightarrow C$, $NB \rightarrow B$:

$NV_3 \equiv 2.5$ since V_3 is indirectly referenced,

$NC_{1:3} \equiv 'C'$ since $C_{1:3}$ is indirectly referenced,

$NB_{2 \text{ TO } 3} \equiv 01_2$ since $B_{2 \text{ TO } 3}$ is indirectly referenced.

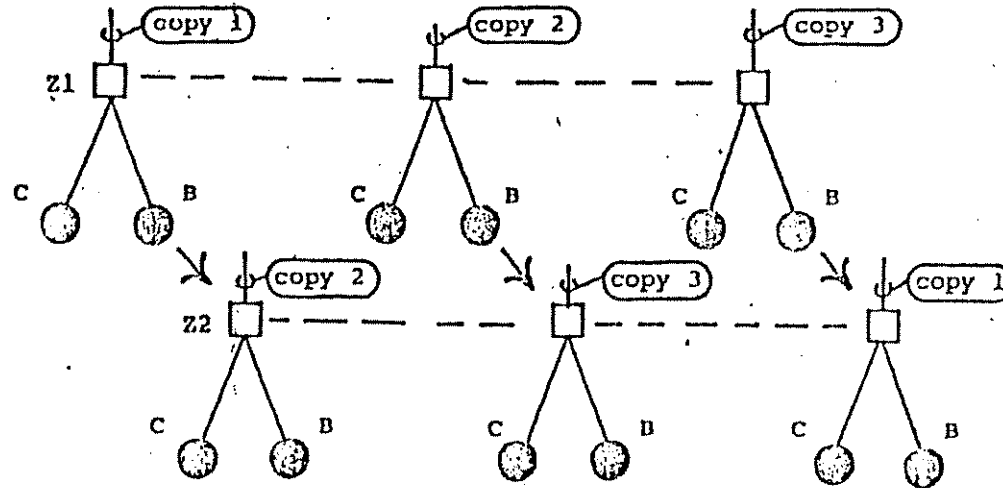
NV_5 , NB_9 are illegal since the subscripting is illegal for V and B respectively. Such subscripting is always illegal since NV can only point to 3-vectors, and B to 4-bit strings.

NAME VARIABLES (CONTINUED)

The complexities arising from structure subscripting are best studied by another apparently simple example. Suppose that the following declarations are made:

```
STRUCTURE A:  
  1 C MATRIX(3,3),  
  1 B NAME A-STRUCTURE;  
DECLARE A-STRUCTURE(3), Z1, Z2, Z3 NAME;
```

Let copies 1, 2 and 3 of Z1.B point respectively to copies 2, 3 and 1 respectively of Z2. Pictorially:



NAME VARIABLES (CONTINUED)

According to the substitution process previously described,
the three copies of structure terminal C and Z2 can be
indirectly accessed by specifying the three copies of Z1.B.C:

Z1.B.C₁; indirectly accesses Z2.C₂;
Z1.B.C₂; indirectly accesses Z2.C₃;
Z1.B.C₃; indirectly accesses Z2.C₁;

NAME VARIABLES (CONTINUED)

Using the terminology of Section 20.1, Z2.C is an operand with arrayness (1:3). Indirectly accessed as Z1.B.C, the operand still has arrayness (1:3) but the order of the individual elements is different. In general of course the three copies of Z1.B may point to three different structures (all with template A), resulting in operand Z1.B.C being synthesized from three different sources.

Note that the structure subscript is effective before indirection not after. As a further illustration, in

$$Z1.B.C_{1;3,3}$$

the structure subscript selects copy 1 of the pointers Z1.B. Note, however, that in contrast the component subscript selects the component in row 3 and column 3 of C in the structure to which Z1.B points.

This is not always true for structure subscripts. For example, let Z3 point to Z2. Then in

$$Z3.B.C_{1;3,3}$$

the structure subscript selects copy 1 of Z2, which is pointed to by Z3.

NAME VARIABLES (CONTINUED)

These examples illustrate the following general rule:

A structure subscript may either be effective on the data being indirectly accessed, or upon the NAME data item accessing it, depending on whether the data pointed to has copies, or whether the NAME data item itself has copies*.

* Note that since a structure terminal which is itself a structure (or a NAME data item pointing to a structure) cannot possess copies, the two forms of structure subscripting are mutually exclusive.

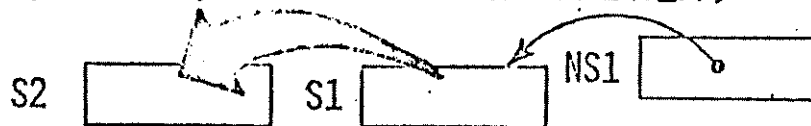
NAME VARIABLES (CONTINUED)

THE NAME PSEUDO-FUNCTION

- ORDINARY REFERENCE TO NAME VARIABLE ACCESSES THE VARIABLE WHICH IT POINTS TO, (A DEREFERENCED USAGE.)
- NAME PSEUDO-FUNCTION IS USED TO ACCESS OR CHANGE THE (POINTER) VALUE OF THE NAME VARIABLE ITSELF.

```
DECLARE SCALAR, S1, S2, NS NAME, NS1 NAME INITIAL(NAME(S1));
```

```
S2 = NS1;
```



```
NAME(NS) = NAME(NS1);
```



```
NAME(NS) = NAME(S2);
```



```
NAME(NS) = NS1;
```

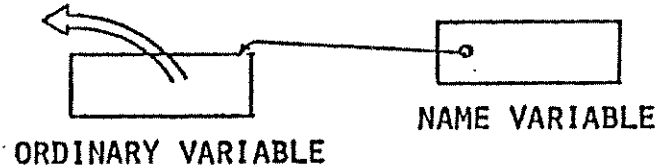
```
NS = NAME(NS1);
```

ERROR-TYPE MISMATCH!

NAME VARIABLES (CONTINUED)

IN GENERAL:

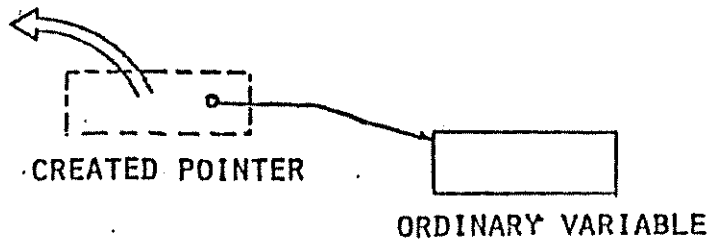
- NAME VARIABLE BY ITSELF DENOTES THE ORDINARY VARIABLE WHICH IT POINTS TO.



- NAME(NAME VARIABLE) OBTAINS THE POINTER CONTENTS OF THE NAME VARIABLE.



- NAME(ORDINARY VARIABLE) CREATES A POINTER TO THAT ORDINARY VARIABLE.



NAME VARIABLES (CONTINUED)

Examples:

Given:

```
DECLARE S SCALAR,  
        NS NAME SCALAR,  
        NT NAME TASK,  
        NA NAME ARRAY(1000) INTEGER;  
STRUCTURE Q:  
  1 QS SCALAR,  
  1 QN NAME Q-STRUCTURE;  
DECLARE ZQ Q-STRUCTURE;
```

the following are legal:

```
NAME(S)      }  
NAME(ZQ.QS)  } reference only  
NAME(NS)  
NAME(NT)  
NAME(NA)  
NAME(ZQ.QN)
```

the following are illegal:

```
NAME(1.5)  
NAME(S/2)
```

NAME VARIABLES (CONTINUED)

SUBSCRIPTING AND NAME VARIABLES

- SUBSCRIPTING OF DEREFERENCED NAME VARIABLES IS ALLOWED
E.G. DECLARE ARRAY(3) VECTOR, V, NV NAME INITIAL(NAME(V));
 $V_{1:3} = NV_{2:3}$ /* SAME AS $V_{1:3} = V_{2:3}$ */

NOT QUITE TRUE

- SUBSCRIPTING INSIDE A REFERENCE TO THE NAME PSEUDO-FUNCTION CAN ONLY APPEAR IN REFERENCE CONTEXT (NOT IN ASSIGNMENT CONTEXT).
SUBSCRIPTING APPLIES TO THE VARIABLE BEING POINTED TO.

E.G. DECLARE INTEGER

A ARRAY(10);

NA NAME ARRAY(10) INITIAL(NAME(A)),

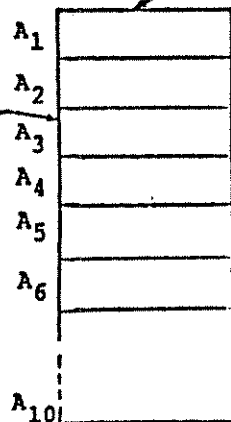
NI NAME; /* NI IS NAME OF SCALAR *INTEGER* */

NAME(NI) = NAME(A₃);

NAME(NI) = NAME(NA₃);

NI

NA

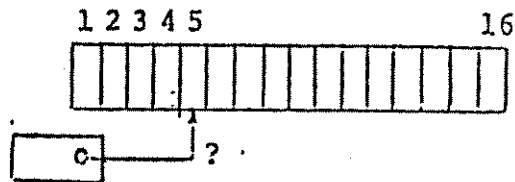


NAME VARIABLES (CONTINUED)

COMPONENT SUBSCRIPTING INSIDE NAME()

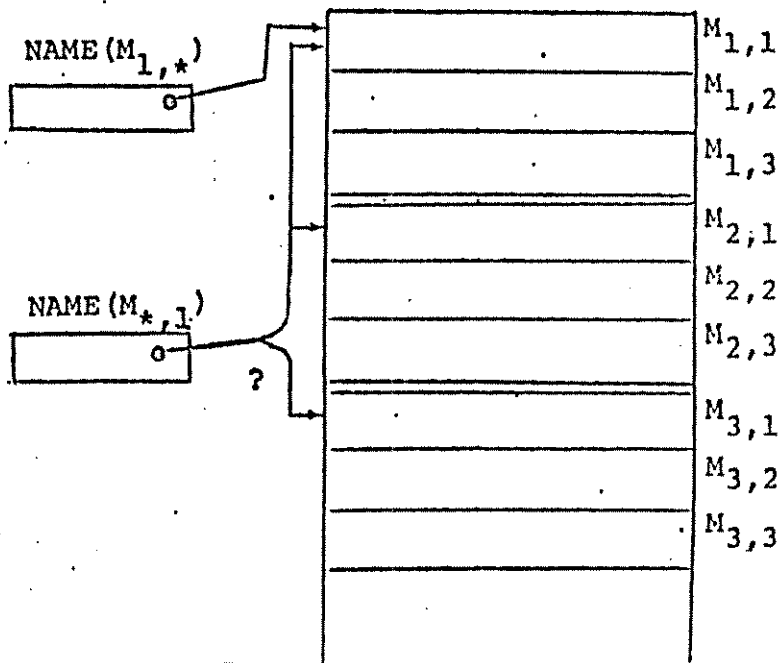
- ILLEGAL FOR BIT AND CHARACTER STRINGS

E.G. DECLARE B BIT(16); NAME(B₅)



- MUST SELECT A SINGLE SCALAR FROM VECTORS AND MATRICES (ALSO SINGLE ELEMENT FROM ARRAYS)

E.G. DECLARE M MATRIX;



NAME VARIABLES (CONTINUED)

STRUCTURE SUBSCRIPTING INSIDE NAME()

- IN ASSIGNMENT CONTEXT, OK ONLY IF NAME() IS APPLIED TO A NAME VARIABLE IN A STRUCTURE WITH MULTIPLE COPIES - THEN IT SELECTS THE APPROPRIATE COPY OF THE NAME VARIABLE ITSELF

```
E.G.  STRUCTURE S: 1 N NAME SCALAR;  
      DECLARE S S-STRUCTURE(10);  
      NAME(N1) = NAME(N2);
```

- IN REFERENCE CONTEXT, ONLY ONE CAN APPLY:
 - NAME VARIABLE DEFINED IN A STRUCTURE WITH COPIES. SUBSCRIPTING IS EFFECTIVE ON THE NAME VARIABLE ITSELF.
 - NAME VARIABLE POINTING TO A STRUCTURE WITH COPIES. SUBSCRIPTING IS EFFECTIVE ON THE VARIABLE BEING POINTED TO.

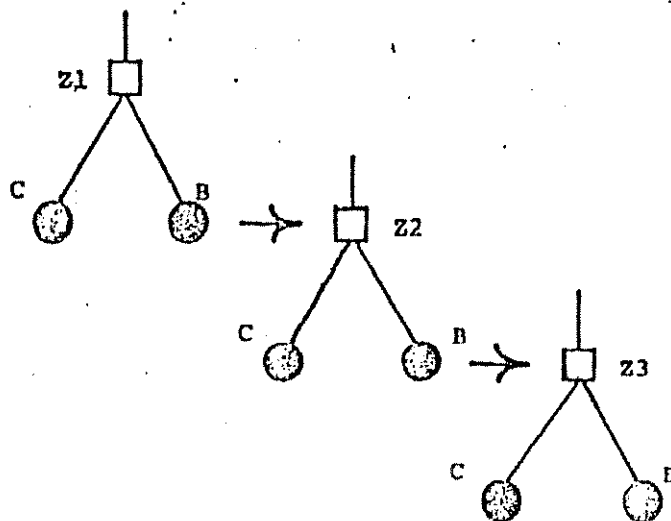
STRUCTURE S: 1 N NAME-STRUCTURE(10); ILLEGAL!

NAME VARIABLES (CONTINUED)

INTERACTION WITH STRUCTURES

```
STRUCTURE A:  
  1 C SCALAR,  
  1 B NAME A-STRUCTURE;  
DECLARE A-STRUCTURE, Z1, Z2, Z3;  
DECLARE Z4 NAME A-STRUCTURE;
```

Let Z1.B point to Z2, and Z2.B point to Z3, as shown pictorially below:



NAME VARIABLES (CONTINUED)

A pointer value to Z3.C can be created by the construct:

NAME(Z3.C)

And also by... NAME(Z2.B.C)

Now Z1.B points to Z2 so that Z3.C is accessed through two levels of indirection by Z1.B.B.C. A third way of creating a pointer value to Z3.C is therefore:

NAME(Z1.B.B.C)

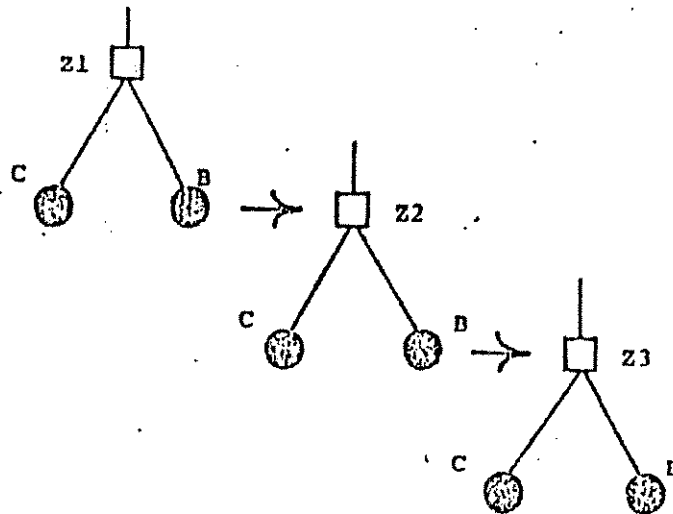
If furthermore, Z4 points to Z1, then

NAME(Z4.B.B.C)

also has the same effect.

NAME VARIABLES (CONTINUED)

```
STRUCTURE A:  
  1 C SCALAR,  
  1 B NAME A-STRUCTURE;  
DECLARE A-STRUCTURE, Z1, Z2, Z3;  
DECLARE Z4 NAME A-STRUCTURE;
```



In each of the above cases, the argument of the NAME pseudo-function is Z3.C which is an ordinary data item, even though indirect access is used. Each of the above instances may therefore only be used in a reference context.

The pointer value of Z2.B can itself be set up by using

NAME(Z2.B)

NAME VARIABLES (CONTINUED)

in an appropriate assignment context to be described. The NAME structure terminal Z2.B may be indirectly accessed by the qualified form Z1.B.B, since Z1.B points to Z2. Hence, the pointer value of Z2.B can also be set up by using:

NAME(Z1.B.B)

in assignment context: With Z4 again pointing to Z1,

NAME(Z4.B.B) .

has the same effect, since Z2.B is again accessed, this time through two levels of indirection.

NAME VARIABLES (CONTINUED)

ARGUMENTS WITH SUBSCRIPTS

Examples:

Given the following declarations:

```
DECLARE V VECTOR(3),  
        NV NAME VECTOR(3),  
        S ARRAY(100) SCALAR,  
        NS NAME ARRAY(100) SCALAR,  
        M ARRAY(5) MATRIX(3,3),  
        NM NAME ARRAY(5) MATRIX(3,3),  
        C CHARACTER(80),  
        NC NAME CHARACTER(80);
```

suppose that $NV \rightarrow V$, $NS \rightarrow S$, $NM \rightarrow M$ and $NC \rightarrow C$.

The following are legal in contexts causing reference of pointer values:

NAME(V ₃)	creates pointer to scalar value which is 3rd element of vector V
NAME(NV ₃)	same as above since $NV \rightarrow V$
NAME(S ₅)	creates pointer to 5th array element of array S

NAME VARIABLES (CONTINUED)

NAME(NS₅) same as above since NS + 5
NAME(M_{3:1,1}) creates pointer to scalar value in row 1,
column 1 of 3rd array element of M
NAME(NM_{3:1,1}) same as above since NM → M
NAME(M_{4:}) creates pointer to 4th array element in M

The following are illegal:

NAME(C₁) } subscripting on character strings
NAME(MC₁) } illegal
NAME(V_{1 TO 2}) } more than one element of V selected
NAME(M_{*:1,1}) } one scalar value selected from more than one
array element

NAME VARIABLES (CONTINUED)

FURTHER RULES:

1. When a NAME pseudo-function is used to assign pointer values, only structure subscripting effective on the pointer copies is legal.
2. For NAME pseudo-functions in reference context, array and component subscripting is always effective on the ordinary data item specified or indirectly accessed. Structure subscripting is effective in the ordinary data item specified or indirectly accessed, or upon the NAME data item indirectly accessing it, depending on which possesses the multiple copies.

NAME VARIABLES (CONTINUED)

Example:

Given the following declarations

```
STRUCTURE A:  
  1 M ARRAY(5) MATRIX(3,3),  
  1 C CHARACTER(80),  
  1 V VECTOR(6),  
  1 B NAME A-STRUCTURE;  
DECLARE Z A-STRUCTURE;  
DECLARE A-STRUCTURE(3), Z1, Z2, Z3 NAME;
```

```
let Z1.B1 + Z22
```

```
Z1.B2 + Z23
```

```
Z1.B3 + Z21
```

```
Z3 + Z1
```

Illustrations for NAME pseudo-functions in a reference context -

(a) Array and component subscripting:

NAME(Z.M _{1,3,3})	creates a pointer to the scalar value in row 3, column 3 of the first array element of Z.M
NAME(Z.M _{*,1,1})	is illegal since the subscript selects a scalar value from more than one array element of Z.M
NAME(Z.C _{10 TO 15})	is illegal since character strings may not possess component subscripts
NAME(Z.V ₁)	creates a pointer to the 1st element of vector Z.V
NAME(Z.V _{1 TO 3})	is illegal since more than one element of Z.V is selected by the subscript

NAME VARIABLES (CONTINUED)

Example:

Given the following declarations

```
STRUCTURE A:  
  1 M ARRAY(5) MATRIX(3,3),  
  1 C CHARACTER(80),  
  1 V VECTOR(6),  
  1 B NAME A-STRUCTURE;  
DECLARE Z A-STRUCTURE;  
DECLARE A-STRUCTURE(3), Z1, Z2, Z3 NAME;
```

```
let Z1.B1 → Z22
```

```
      Z1.B2 → Z23
```

```
      Z1.B3 → Z21
```

```
      Z3 → Z1
```

Illustrations for NAME pseudo-functions in a reference context -

- (b) Structure subscripting effective upon the data item pointed to or directly specified:

NAME(Z1₂)

creates a pointer to the second copy of Z1 since the subscript acts directly on Z1

NAME(Z3₂)

since Z3 is a single pointer, pointing to the whole of Z1, the subscript is effective on Z1 rather than Z3; hence a pointer to the second copy of Z1 is again created

NAME VARIABLES (CONTINUED)

NAME(Z1.M₂;) creates a pointer to the array of matrices M in the second copy of Z1

NAME(Z3.M₂;) as before, the structure subscript is effective on Z1 rather than Z3; hence as before a pointer to the array of matrices M in the second copy of Z1 is created

NAME(Z1.M₁ TO 2;) is illegal since the subscript selects more than one copy of structure Z1

NAME(Z3.M₁ TO 2;) is illegal for the same reason

NAME(Z1.M) is illegal since subscripting to select one copy of Z1.M must be used

NAME VARIABLES (CONTINUED)

Example:

Given the following declarations

```
STRUCTURE A:  
  1 M ARRAY(5) MATRIX(3,3),  
  1 C CHARACTER(80),  
  1 V VECTOR(6),  
  1 B NAME A-STRUCTURE;  
DECLARE Z A-STRUCTURE;  
DECLARE A-STRUCTURE(3), Z1, Z2, Z3 NAME;
```

```
let Z1.B1 + Z22
```

```
Z1.B2 + Z23
```

```
Z1.B3 + Z21
```

```
Z3 + Z1
```

Illustrations for NAME pseudo-functions in a
reference context -

NAME VARIABLES (CONTINUED)

(c) Structure subscripting effective on a pointer value:

The following examples use the fact that $Z1.B_1$ points to $Z2_2$

`NAME(Z1.B1)` references the pointer value $Z1.B_1$, i.e. it creates the pointer to $Z2_2$

`NAME(Z1.B.M1;))` the subscript is effective of $Z1.B$, so that a pointer to the array of matrices in the second copy of $Z2$ is created

`NAME(Z1.B.V1;1)` the structure subscript is effective on $Z1.B$ as before so that a pointer to the first component of the vector in the second copy of $Z2$ is created

Note that there is no restriction on the selection of one pointer only by a structure subscript effective on pointer data:

`NAME(Z1.B)` "simultaneously" references three pointer values

`NAME(Z1.B.M1 TO 2;))` "simultaneously" creates two pointers, to matrices $Z2.M_2$, and $Z2.M_3$, respectively

NAME VARIABLES (CONTINUED)

INITIALIZATION OF NAME VARIABLES

{ INITIAL }
{ CONSTANT } (NAME REFERENCE)

NAME REFERENCE HAS THE FORM:

NAME (ORDINARY VARIABLE)

NAME (NULL) OR NULL

"NULL POINTER" - POINTS TO
NOTHING AT ALL. UNINITIALIZED
NAME VARIABLES ALSO HAVE THIS
VALUE.

UNLESS SUBSCRIPTS
ARE KNOWN AT
COMPILE-TIME.

ORDINARY VARIABLE MUST BE:

- PREVIOUSLY DECLARED
- WITH DATATYPE MATCHING
THAT OF NAME VARIABLE

AND MUST NOT BE:

- SUBSCRIPTED
- DEREFERENCED THROUGH ANOTHER
NAME VARIABLE

NAME VARIABLES (CONTINUED)

INITIALIZATION OF NAME VARIABLES

1. DECLARE S SCALAR,
 NS NAME SCALAR INITIAL(NAME(S));

2. NAME VARIABLES IN STRUCTURES WILL BE DISCUSSED LATER.
 STRUCTURE A:
 1 B SCALAR,
 1 C NAME A-STRUCTURE,
 DECLARE A-STRUCTURE,
 Z1,
 Z2 INITIAL(1.5, NAME(Z1));
 DECLARE NB NAME SCALAR INITIAL(NAME(Z1.B));

3. DECLARE A ARRAY(5) INTEGER INITIAL(0),
 NI NAME INTEGER INITIAL(NAME(A₃));

NAME VARIABLES (CONTINUED)

Examples:

The following are legal initializations of NAME data items:

```
DECLARE S SCALAR,  
      V ARRAY(4) VECTOR DOUBLE;  
DECLARE NS1 NAME SCALAR INITIAL(NAME(S));  
DECLARE NV1 NAME ARRAY(4) VECTOR DOUBLE  
      INITIAL(NAME(V));  
STRUCTURE A:  
  1 C SCALAR,  
  1 B NAME A-STRUCTURE;  
DECLARE Z1 A-STRUCTURE;  
DECLARE Z2 A-STRUCTURE INITIAL(1.5, NAME(Z1));  
DECLARE NA NAME SCALAR INITIAL(NAME(Z1.C));
```

```
DECLARE V VECTOR(4);  
DECLARE TV NAME SCALAR INITIAL(NAME(V ));
```

S

3

NAME VARIABLES (CONTINUED)

The following are illegal initializations of NAME data items:

```
DECLARE T SCALAR;  
DECLARE NT NAME SCALAR DOUBLE  
      INITIAL(NAME(T));
```

NT cannot legally
point to T

```
DECLARE NT1 NAME SCALAR INITIAL(NAME(T1));  
DECLARE T1 SCALAR;
```

T1 is not previously
defined

```
STRUCTURE X:  
  1 Y SCALAR,  
  1 Z NAME X-STRUCTURE;  
DECLARE XX1 X-STRUCTURE;  
DECLARE XX2 X-STRUCTURE INITIAL(1.5, NAME(XX1));  
DECLARE NX NAME SCALAR INITIAL(NAME(XX2.Z.Y));
```

contains implicit
indirection since
XX2,Z → XX1 through
previous initialization

NAME VARIABLES (CONTINUED)

NULL INITIALIZATION

All NAME data items which are not explicitly initialized, are implicitly initialized with null pointer values. The following examples show the explicit initialization to null pointer values.

Examples:

```
DECLARE LV NAME VECTOR INITIAL(NULL);  
STRUCTURE A:  
  1 C SCALAR,  
  1 B NAME A-STRUCTURE  
DECLARE Z A-STRUCTURE(20) INITIAL(20#(7.53, NULL));
```

each copy of B initialized
to a null pointer value

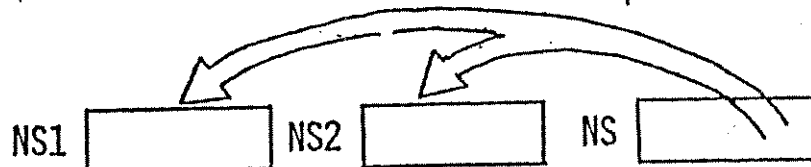
NAME VARIABLES (CONTINUED)

NAME ASSIGNMENTS

$$L^1, L^2, \dots, L^n = R;$$

{ EACH L^i AND R IS A NAME
PSEUDO-FUNCTION.

E.G. NAME(NS1), NAME(NS2) = NAME(NS);



NAME COMPARISONS

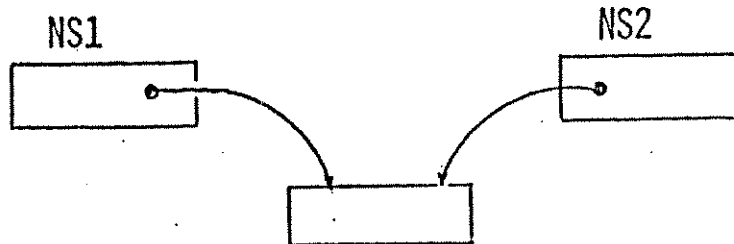
$$L = R$$

$$L \neq R$$

{ L AND R ARE BOTH NAME
PSEUDO-FUNCTIONS.

E.G. IF NAME(NS1) \neq NAME(NS2), THEN ...;

EQUALITY IF BOTH NAME VARIABLES POINT TO THE SAME
ORDINARY VARIABLE!



NAME VARIABLES (CONTINUED)

NAME ASSIGNMENTS

Examples:

Given the declarations

```
DECLARE S SCALAR,  
        NS NAME SCALAR,  
        NSD NAME SCALAR DOUBLE;  
DECLARE V VECTOR(3),  
        NV NAME VECTOR(3);  
STRUCTURE A:  
  1 C SCALAR,  
  1 B NAME A-STRUCTURE;  
DECLARE Z1 A-STRUCTURE,  
        Z2 A-STRUCTURE,  
        NZ NAME A-STRUCTURE;
```

then

NAME(NSD) = NULL;	results in NSD + \emptyset^*
NAME(NS) = NAME(S);	results in NS + S
NAME(NSD) = NAME(NS);	is illegal since NS + S and NSD may not legally point to S itself
NAME(NV) = NAME(V);	results in NV + V
NAME(NS) = NAME(V ₂);	results in NS + V ₂ - note that V ₂ is a <u>scalar</u> value, which is why NS may legally point to it
NAME(NZ) = NAME(Z1);	results in NZ + Z1
NAME(NZ.B) = NAME(Z2);	results in Z1.B + Z2 because of implied indirection in qualified reference NZ.B, in which NZ + Z1
NAME(NS) = NAME(NZ.B.C);	results in NS + Z2.C because of 2 levels of implied indirection in qualified form NZ.B.C, in which NZ + Z1 and Z1.B + Z2

NAME VARIABLES (CONTINUED)

MULTIPLE ASSIGNMENTS

Example:

Given

```
DECLARE S SCALAR,  
        NS NAME SCALAR,  
        NT NAME SCALAR;  
STRUCTURE U:  
  1 US NAME SCALAR,  
  1 UN NAME U-STRUCTURE;  
DECLARE Z U-STRUCTURE;
```

The following is a legal multiple NAME assignment:

```
NAME(NS), NAME(NT), NAME(Z.US) = NAME(S);
```


NAME VARIABLES (CONTINUED)

POINTER ARRAYNESS IN NAME ASSIGNMENTS

Examples:

Given

```
STRUCTURE A:  
  1 B NAME SCALAR,  
  1 C SCALAR;  
DECLARE Z1 A-STRUCTURE(3),  
        Z2 A-STRUCTURE(5);  
DECLARE S SCALAR;
```

then 3 copies of Z1.B exist, and 5 copies of Z2.B exist. Hence in

```
NAME(Z1.B) = NAME(S);
```

the pointer arrayness on the left is {3} whilst the right hand operand has none. The result of this assignment is:

```
Z1.B1; ↘  
Z1.B2; ↘ S  
Z1.B3; ↘
```

NAME VARIABLES (CONTINUED)

NAME COMPARISONS

Examples:

Given

```
DECLARE S SCALAR;  
DECLARE NS NAME SCALAR INITIAL(NAME(S)),  
        NT NAME SCALAR INITIAL(NULL);
```

Then

```
NAME(NS) = NAME(S) is TRUE;  
NAME(NS) = NAME(NULL) is FALSE;  
NAME(NT) != NAME(NULL) is FALSE;  
NAME(NT) != NAME(NS) is TRUE;
```

NAME VARIABLES (CONTINUED)

POINTER ARRAYNESS IN NAME COMPARISONS

Examples:

Given

```
STRUCTURE A:  
  1 D NAME SCALAR,  
  1 C SCALAR;  
DECLARE Z1 A-STRUCTURE(3),  
        Z2 A-STRUCTURE(5);  
DECLARE S SCALAR;
```

After execution of

```
NAME(Z1.D) = NAME(S);
```

then the result of the comparison

```
NAME(Z1.D) = NAME(S) is TRUE
```

since Z1.D₁;
Z1.D₂;
Z1.D₃;

NAME VARIABLES (CONTINUED)

Further,

```
| NAME(Z2.B) = NAME(Z1.B);
```

is illegal since the left and right hand pointer arraynesses are {5} and {3} respectively which do not match. However,

```
| NAME(Z2.B      ) = NAME(Z1.B);  
|S      3 TO 5;
```

is legal since the left hand arrayness has been reduced to {3}. The result of the assignment is

```
Z2.B3 ≡ Z1.B1      (i.e. they both have the same  
Z2.B4 ≡ Z1.B2      pointer value)  
Z2.B5 ≡ Z1.B3;
```

NAME VARIABLES (CONTINUED)

Given

```
STRUCTURE A:  
  1 D NAME SCALAR,  
  1 C SCALAR;  
DECLARE Z1 A-STRUCTURE(3),  
        Z2 A-STRUCTURE(5);  
DECLARE S SCALAR;
```

After subsequent execution of

```
S NAME(Z1.D1) = NULL;  
  1;
```

then the result of the comparison

NAME(Z1.D) = NAME(S) is FALSE

because Z1.D₁ → β

Z1.D₂ → S
Z1.D₃ → S

The comparison

NAME(Z1.D) = NAME(Z2.D)

is illegal because the pointer arraynesses of the left and right operands are (3) and (5) respectively, which do not match. However, the comparison

NAME(Z1.D) = NAME(Z2.D_{3 TO 5})

is legal since the pointer arrayness of the right hand operand has been reduced to (3).

NAME ARGUMENTS AND PARAMETERS

```
P: PROCEDURE(NA) ASSIGN(NB);  
  DECLARE NA NAME INTEGER,  
         NB NAME ARRAY(10) SCALAR;  
  ⋮  
CLOSE P;
```

- ARGUMENT MATCHING NA: NAME PSEUDO-FUNCTION IN REFERENCE CONTEXT (OR NULL).
- ARGUMENT MATCHING NB: NAME PSEUDO-FUNCTION IN ASSIGNMENT CONTEXT.

```
DECLARE I INTEGER;  
DECLARE ARRAY(10) SCALAR, A, N NAME;  
CALL P(NAME(I)) ASSIGN(NAME(N));  
CALL P(NULL) ASSIGN(NAME(N));  
CALL P(NAME(I)) ASSIGN(NAME(A));  
CALL P(NULL) ASSIGN(NAME(N*));
```

LEGAL

ILLEGAL

10-62

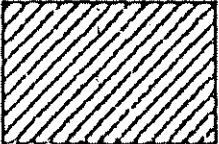
~~NAME(A) = NB~~

NAME VARIABLES (CONTINUED)

INPUT ARGUMENTS

The effect of using a pointer value as an input argument of a procedure or function is as if the pointer value were being assigned to the corresponding NAME input parameter. The attributes of the NAME input parameter must therefore allow legal acceptance of that pointer value.

Examples:

```
DECLARE S SCALAR;  
DECLARE NS NAME SCALAR;  
DECLARE NT NAME TASK;  
:  
:  
F: FUNCTION(A,B) SCALAR;  
  DECLARE A NAME SCALAR,  
         B BOOLEAN;  
   } function body  
CLOSE F;  
:  
:
```

NAME VARIABLES (CONTINUED)

<code>NAME(NS) = NAME(S);</code>	
<code>S = F(NAME(S), TRUE);</code>	invocation results in input parameter A pointing to S
<code>S = F(NAME(NS), FALSE);</code>	has the same effect: A gets same pointer value as NS, i.e. A + S
<code>S = F(NAME(NT), TRUE);</code>	is illegal since pointer values legal for NT are not legal for A
<code>S = F(NULL, FALSE);</code>	results in A + \emptyset

Note that although ordinary input parameters are prevented from appearing in NAME pseudo-functions, NAME input parameters are only prevented from appearing in NAME pseudo-functions in assignment context.

NAME VARIABLES (CONTINUED)

ASSIGN ARGUMENTS

A pointer value may be passed both into and out of a procedure by the appearance of a NAME pseudo-function in the assign argument list of the procedure's invocation. The class of data items which can be pointed to by the NAME data item appearing in the NAME pseudo-function must be the same as that which can be pointed to by the corresponding NAME assign parameter.

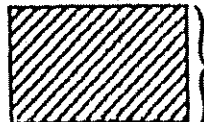
NAME VARIABLES (CONTINUED)

Examples:

```
DECLARE NS NAME SCALAR;  
DECLARE NT NAME TASK;  
STRUCTURE A:  
  1 B NAME A-STRUCTURE,  
  1 C SCALAR;  
DECLARE Z A-STRUCTURE;
```

```
·  
·  
·
```

```
P: PROCEDURE ASSIGN(U,V);  
  DECLARE U NAME TASK,  
         V NAME A-STRUCTURE;
```



} procedure body

```
CLOSE P;
```

```
·  
·  
·
```

```
CALL P ASSIGN(NAME(NT), NAME(Z.B));
```

causes passage of pointer values
between NT and U, and between Z.B
and V.

```
CALL P ASSIGN(NAME(NS), NAME(Z));
```

illegal because
NS points to
scalar data items
but NT points to
tasks

illegal because Z is not a
NAME data item

NAME VARIABLES (CONTINUED)

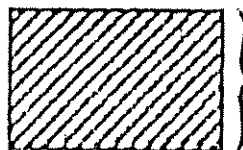
POINTER ARRAYNESS IN ARGUMENTS

Examples:

```
STRUCTURE A:  
  1 B NAME SCALAR;  
DECLARE Z A-STRUCTURE(20);  
DECLARE S1 ARRAY(20) SCALAR,  
        S2 ARRAY(10) SCALAR;
```

.
. .
.

```
P: PROCEDURE(U) ASSIGN(V);  
  DECLARE U NAME SCALAR,  
         V NAME SCALAR;
```

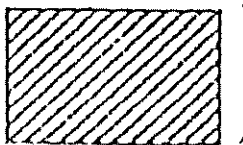


} procedure body

```
CLOSE P;
```

.
. .
.

```
F: FUNCTION(W) SCALAR;  
  DECLARE W NAME SCALAR;
```



} function body

```
CLOSE F;
```

.
. .
.

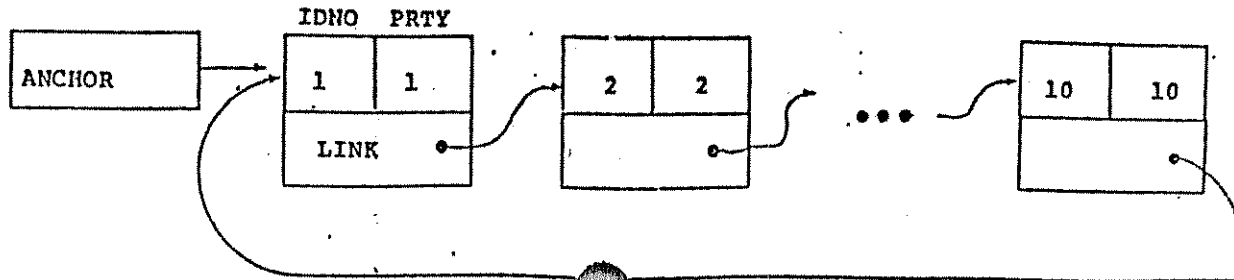
NAME VARIABLES (CONTINUED)

USES OF STRUCTURES AND NAME VARIABLES

① A PRIORITY-ORDERED QUEUE

```
STRUCTURE QUEUE:  
  1 IDNO INTEGER,  
  1 PRTY INTEGER,  
  1 LINK NAME QUEUE-STRUCTURE;  
DECLARE QUEUE QUEUE-STRUCTURE(10),  
        ANCHOR NAME QUEUE-STRUCTURE;  
DECLARE INTEGER, I, NEW_ID, NEW_PRTY;  
  ⋮  
DO FOR I = 1 TO 9;  
  IDNOI, PRTYI = I;  
  NAME(LINKI) = NAME(QUEUEI+1);  
END;  
IDNO10, PRTY10 = 10;  
NAME(ANCHOR), NAME(LINK10) = NAME(QUEUE1);
```

PROBLEM #1:
QUEUE CANNOT BE
UNQUALIFIED BECAUSE
OF NESTED STRUCTURE
REFERENCES.



NAME VARIABLES (CONTINUED)

PROBLEM #2: CANNOT FACTOR
NAME DECLARES

USES OF STRUCTURES AND NAME VARIABLES

DECLARE NAME QUEUE-STRUCTURE, THIS, PREV;

/* FIND IDNO IN QUEUE */

NAME(PREV) = NAME(ANCHOR);
NAME(THIS) = NAME(ANCHOR.LINK);

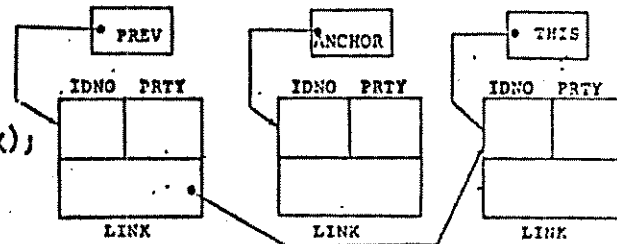
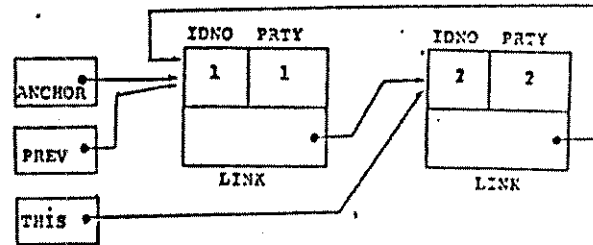
DO WHILE THIS.IDNO ≠ NEW_IDNO;
 NAME(PREV) = NAME(THIS);
 NAME(THIS) = NAME(THIS.LINK);
END;

/* SET NEW PRIORITY, AND TEMPORARILY UNLINK FROM QUEUE */

THIS.PRTY = NEW_PRTY;
NAME(ANCHOR), NAME(PREV.LINK) = NAME(THIS.LINK);

/* FIND PROPER PLACE TO RE-LINK IN QUEUE ACCORDING TO PRIORITY */

DO UNTIL PREV.PRTY ≤ NEW_PRTY AND NEW_PRTY ≤ THIS.PRTY;
 NAME(PREV) = NAME(THIS);
 NAME(THIS) = NAME(THIS.LINK);
END;
NAME(PREV.LINK) = NAME(ANCHOR);
NAME(ANCHOR.LINK) = NAME(THIS);



NAME VARIABLES (CONTINUED)

```
1 M† PRIO+0:
1 M† PROGRAM;
2 M†   STRUCTURE NODE:
2 M†     1 IDNO INTEGER;
2 M†     1 PRTY INTEGER;
2 M†     1 LINK NAME NODE-STRUCTURE;
3 M†   DECLARE QUEUE NODE-STRUCTURE(10);
3 M†     ANCHOR NAME NODE-STRUCTURE;
4 M†   DECLARE THIS NAME NODE-STRUCTURE;
4 M†     PREV NAME NODE-STRUCTURE;
5 M†   DECLARE INTEGER,
5 M†     I, NEW+ID, NEW+PRTY;
6 M†   DO FOR I = 1 TO 9;
7 M†     QUEUE.IDNO , QUEUE.PRTY = I;
8 M†     S†           I           I
8 M†     E†           +           +
8 M†     NAME(QUEUE.LINK ) = NAME(QUEUE );
8 M†     S†           I           I+1
9 M†   END;
10 M†   QUEUE.IDNO , QUEUE.PRTY = 10;
10 M†     S†           10           10
10 M†     E†           +           +
11 M†   NAME(ANCHOR), NAME(QUEUE.LINK ) = NAME(QUEUE );
11 M†     S†           10           1
11 M†     C†
11 M†     C†   FIND IDNO IN QUEUE
```

NAME VARIABLES (CONTINUED)

```

C†
E†
12 N†   NAME(PREV) = NAME(ANCHOR);
E†
13 N†   NAME(THIS) = NAME(ANCHOR.LINK);
14 N†   DO WHILE THIS.IDNO \= NEW.ID;
E†
15 N†   NAME(PREV) = NAME(THIS);
E†
16 N†   NAME(THIS) = NAME(THIS.LINK);
17 N†   END;
C†
C†
C†   SET NEW PRIORITY, AND TEMPORARILY UNLINK FROM QUEUE
18 N†   THIS.PRTY = NEW.PRTY.
E†
19 N†   NAME(ANCHOR), NAME(PREV.LINK) = NAME(THIS.LINK);
C†
C†   FIND PROPER PLACE TO RE-LINK IN QUEUE ACCORDING TO PRIORITY
20 N†   DO UNTIL PREV.PRTY <= NEW.PRTY AND NEW.PRTY <= THIS.PRTY;
E†
21 N†   NAME(PREV) = NAME(THIS);
E†
22 N†   NAME(THIS) = NAME(THIS.LINK);
23 N†   END;
E†
24 N†   NAME(PREV.LINK) = NAME(ANCHOR);
E†
25 N†   NAME(ANCHOR.LINK) = NAME(THIS);
26 N†   CLOSE.

```


NAME VARIABLES (CONTINUED)

```
C↑
E↑
12 M↑ NAME(PREV) = NAME(ANCHOR);
E↑
13 M↑ NAME(THIS) = NAME(ANCHOR.LINK);
14 M↑ DO WHILE THIS.IDNO ≠ NEW.ID;
E↑
15 M↑ NAME(PREV) = NAME(THIS);
E↑
16 M↑ NAME(THIS) = NAME(THIS.LINK);
17 M↑ END;
C↑
C↑ SET NEW PRIORITY, AND TEMPORARILY UNLINK FROM QUEUE
C↑
18 M↑ THIS.PRTY = NEW.PRTY.
E↑
19 M↑ NAME(ANCHOR), NAME(PREV.LINK) = NAME(THIS.LINK);
C↑
C↑ FIND PROPER PLACE TO RE-LINK IN QUEUE ACCORDING TO PRIORITY
C↑
20 M↑ DO UNTIL PREV.PRTY ≤ NEW.PRTY AND NEW.PRTY ≤ THIS.PRTY;
E↑
21 M↑ NAME(PREV) = NAME(THIS);
E↑
22 M↑ NAME(THIS) = NAME(THIS.LINK);
23 M↑ END;
E↑
24 M↑ NAME(PREV.LINK) = NAME(ANCHOR);
E↑
25 M↑ NAME(ANCHOR.LINK) = NAME(THIS);
26 M↑ CLOSE.
```

NAME VARIABLES (CONTINUED)

0000020		ST#15	EQU	*	
00020 1CE2			LR	4,2	
00020 8C18	0007		STH	4,7(1)	PREV
000002E		ST#16	EQU	*	
0002E 9A0R	0002		LH	2,2(2)	
0002F 1CE2			LR	4,2	
00030 8C19	0006		STH	4,6(1)	THIS
0000031		ST#17	EQU	*	
00031 DF2A	0028		BC	7,+9	LBL#6
0000032		LBL#7	EQU	*	
0000032		ST#18	EQU	*	
00032 9D11	0004		LH	5,4(1)	NEW+PRTY
00033 9A19	0006		LH	2,6(1)	THIS
00034 8D26	0001		STH	5,1(2)	
0000035		ST#19	EQU	*	
00035 9A0R	0002		LH	2,2(2)	
00036 1CE2			LR	4,2	
00037 9A1D	0007		LH	2,7(1)	PREV
00038 8C0R	0002		STH	4,2(2)	
00039 8C15	0005		STH	4,5(1)	ANCHOR
000003A		ST#20	EQU	*	
0003A DF24	0044		BC	7,++10	LBL#8
000003B		LBL#9	EQU	*	
0003B 9A1D	0007		LH	2,7(1)	PREV
0003C 9D06	0001		LH	5,1(2)	
0003D 9511	0004		CH	5,4(1)	NEW+PRTY
0003E D914	0044		BC	1,++6	LBL#13
000003F		LBL#12	EQU	*	
0003F 9D11	0004		LH	5,4(1)	NEW+PRTY
00040 9A19	0006		LH	2,6(1)	THIS
00041 9506	0001		CH	5,1(2)	
00042 D904	0044		BC	1,++2	LBL#13
00043 DF1C	0048		BC	7,++8	LBL#14
0000044		LBL#15	EQU	*	
0000044		LBL#13	EQU	*	
0000044		LBL#8	EQU	*	

NAME VARIABLES (CONTINUED)

0000044		ST#21	EQU	*	
00041 9A19	0006		LH	2,6(1)	THIS
00045 1CE2	-		LR	4,2	
00046 BC1D	0007		STH	4,7(1)	PREV
0000047	-	ST#22	EQU	*	
00047 9A0A	0002		LH	2,2(2)	
00048 1CE2	-		LR	4,2	
00049 BC19	0006		STH	4,6(1)	THIS
000004A	-	ST#23	EQU	*	
0004A DF42	003B		BC	7, *-15	LBL#9
000004B	-	LBL#14	EQU	*	
000004B	-	ST#24	EQU	*	
0004B 9A15	0005		LH	2,5(1)	ANCHOR
0004C 1CE2	-		LR	4,2	
0004D 9A1D	0007		LH	2,7(1)	PREV
0004E BC0A	0002		STH	4,2(2)	
000004F	-	ST#25	EQU	*	
0004F 9A19	0006		LH	2,6(1)	THIS
00050 1CE2	-		LR	4,2	
00051 9A15	0005		LH	2,5(1)	ANCHOR
00052 BC0A	0002		STH	4,2(2)	

NAME VARIABLES (CONTINUED)

USES OF STRUCTURES AND NAME VARIABLES

② TREE-STRUCTURED SYMBOL TABLE

STRUCTURE TREE:

```

1 SYMBOL CHARACTER(32),
1 LESS NAME TREE-STRUCTURE,
1 GTR NAME TREE-STRUCTURE;

```

```

DECLARE TREE TREE-STRUCTURE(100),
NEW_SYMBOL CHARACTER(32),
I INTEGER INITIAL(0),
WAS_LESS BOOLEAN;
DECLARE NAME TREE-STRUCTURE,
ROOT INITIAL(NULL), OLD_LEAF, THIS;

```

```

NAME(THIS) = NAME(ROOT);
DO WHILE NAME(THIS) ≠ NAME(NULL);
  NAME(OLD_LEAF) = NAME(THIS);
  IF NEW_SYMBOL < THIS.SYMBOL
    THEN DO; WAS_LESS = TRUE; NAME(THIS) = NAME(THIS.LESS); END;
    ELSE DO; WAS_LESS = FALSE; NAME(THIS) = NAME(THIS.GTR); END;
END;

```

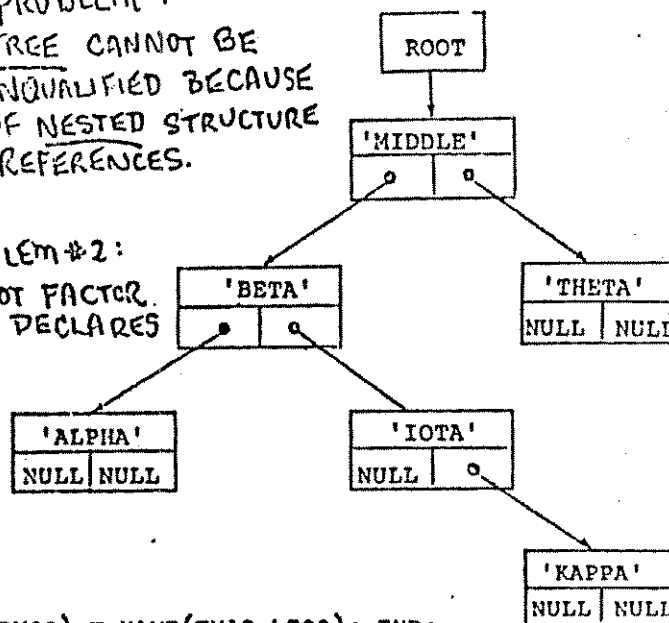
```

I = I+1; SYMBOLI = NEW_SYMBOL;
NAME(LESSI), NAME(GTRI) = NAME(NULL);
IF NAME(ROOT) = NAME(NULL) THEN NAME(ROOT) = NAME(TREEI);
ELSE IF WAS_LESS THEN NAME(OLD_LEAF.LESS) = NAME(TREEI);
ELSE NAME(OLD_LEAF.GTR) = NAME(TREEI);

```

PROBLEM #1:
TREE CANNOT BE
UNQUALIFIED BECAUSE
OF NESTED STRUCTURE
REFERENCES.

PROBLEM #2:
CANNOT FACTOR
NAME DECLARES



NAME VARIABLES (CONTINUED)

```
1 N↑ TREE+PROB·
1 N↑ PROGRAM:
2 N↑   STRUCTURE NODE:
2 N↑     1 SYMBOL CHARACTER(32),
2 N↑     1 LESS NAME NODE-STRUCTURE,
2 N↑     1 OTR NAME NODE-STRUCTURE;
3 N↑   DECLARE TREE NODE-STRUCTURE(100),
3 N↑     NEW+SYMBOL CHARACTER(32),
3 N↑     I INTEGER INITIAL(0),
3 N↑     WAS+LESS BOOLEAN;
4 N↑   DECLARE ROOT NAME NODE-STRUCTURE INITIAL(NULL);
5 N↑   DECLARE OLD+LEAF NAME NODE-STRUCTURE;
6 N↑   DECLARE THIS NAME NODE-STRUCTURE;
   C↑
   E↑
7 N↑     NAME(THIS) = NAME(ROOT);
   E↑
8 N↑     DO WHILE NAME(THIS) ≠ NAME(NULL);
   E↑
9 N↑       NAME(OLD+LEAF) = NAME(THIS);
   E↑
10 N↑       IF NEW+SYMBOL < THIS.SYMBOL THEN
11 N↑         DO,
   E↑
12 N↑           WAS+LESS = TRUE;
   E↑
13 N↑           NAME(THIS) = NAME(THIS.LESS);
14 N↑         END;
15 N↑       ELSE
16 N↑         DO.
```


NAME VARIABLES (CONTINUED)

Address	Label	Value	Operation	Comment
0000003			ST#7 EQU *	
0000004			#OTREEFR CSECT	ESDID= 0001
0000005	0015		LH 2,21(1)	ROOT
0000006			LR 4,2	
0000007	0017		STH 4,23(1)	THIS
0000008			ST#8 EQU *	
0000009			LBL#3 EQU *	
0000010	0017		LH 2,23(1)	THIS
0000011			LR 5,2	
0000012	0015		LH 2,1926(1)	H'0'
0000013			CR 5,2	
0000014	0017		BC 4,#+22	LBL#4
0000015			LBL#5 EQU *	
0000016			LBL#5 EQU *	
0000017			ST#9 EQU *	
0000018	0017		LH 2,23(1)	THIS
0000019			LR 4,2	
0000020	0016		STH 4,22(1)	OLD+LEAF
0000021			ST#10 EQU *	
0000022			LR 3,2	
0000023	0004		LA 2,4(1)	NEW+SYMBOL
0000024			BAL 4,0(3)	CPRC
0000025	0021		BC 5,#+8	LBL#7
0000026			ST#11 EQU *	
0000027			ST#12 EQU *	
0000028	0003		SB 3(1),1	WAS+LESS
0000029			ST#13 EQU *	
0000030	0017		LH 2,23(1)	THIS
0000031	0011		LH 2,17(2)	
0000032			LR 4,2	
0000033	0017		STH 4,23(1)	THIS
0000034			ST#14 EQU *	
0000035			ST#15 EQU *	
0000036	0026		BC 7,#+6	LBL#8
0000037			LBL#7 EQU *	
0000038			ST#16 EQU *	
0000039	0003		ZH 3(1)	WAS+LESS
0000040			ST#17 EQU *	
0000041	0017		LH 2,23(1)	THIS
0000042	0012		LH 2,18(2)	
0000043			LR 4,2	
0000044	0017		STH 4,23(1)	THIS

NAME VARIABLES (CONTINUED)

0000026		ST#18	EQU	*	
0000026		ST#19	EQU	*	
0000026		LBL#8	EQU	*	
00026 DF6E	000C		BC	7, *-26	LBL#3
0000027		LBL#4	EQU	*	
0000027		ST#20	EQU	*	
00027 9D09	0002		LH	5, 2(1)	I
00028 B0E5 0001			AHI	5, 1	
00028 B009	0002		STH	5, 2(1)	I
0000028		ST#21	EQU	*	
00028 9CF9 0785	0785		NIH	5, 1925(1)	H'19'
00028 EA11	0004		LA	2, 4(1)	NEW+SYMBOL
00028 E9F5 8005	0005		LA	1, 5(5, 1)	TREE
00030 E1F3 0000			BAL	4, 0(3)	CAS
0000032		ST#22	EQU	*	
00032 9F09	0002		LH	7, 2(1)	I
00032 9FF9 0785	0785		NIH	7, 1925(1)	H'19'
00032 9CF1 0785	0785		LH	4, 1926(1)	H'0'
00037 BCF5 E017	0017		STH	4, 22(7, 1)	TREE+13
00037 BCF5 E016	0016		STH	4, 22(7, 1)	TREE+17
0000038		ST#23	EQU	*	
00038 9A55	0015		LH	2, 21(1)	ROOT
00038 1DE2			LR	5, 2	
00038 9AF1 0786	0786		LH	2, 1926(1)	H'0'
0003F 15E2			CR	5, 2	
00040 DB1C	0042		BC	3, **8	LBL#9
0000041		LBL#11	EQU	*	
0000041		LBL#10	EQU	*	
0000041		ST#24	EQU	*	
00041 9F09	0002		LH	7, 2(1)	I
00042 9FF9 0785	0785		NIH	7, 1925(1)	H'19'
00044 ECF5 E005	0005		LA	4, 5(7, 1)	TREE
00046 B055	0015		STH	4, 21(1)	ROOT
0000047		ST#25	EQU	*	
00047 DF44	0059		BC	7, **18	LBL#12
0000049		LBL#9	EQU	*	
00048 9D0D	0002		LH	5, 3(1)	NEW+LESS
00049 1C2D	0052		BC	4, **9	LBL#13

10-80

NAME VARIABLES (CONTINUED)

000004F		
0004A 9F09	0002	
0004B 9FF9 0785	0785	
0004D ECF5 E005	0005	
0004F 9A59	0016	
00050 BC46	0011	
0000051		
00051 0F10	0059	
0000052		
00052 9F09	0002	
00053 9FF9 0785	0785	
00055 ECF5 E005	0005	
00057 9A59	0016	
00059 9C4A	0012	

ST#26	EQU	*
	LH	7, 2(1)
	MIH	7, 1925(1)
	LA	4, 5(7, 1)
	LH	2, 22(1)
	STH	4, 17(2)
ST#27	EQU	*
	BC	7, *+8
LBL#13	EQU	*
	LH	7, 2(1)
	MIH	7, 1925(1)
	LA	4, 5(7, 1)
	LH	2, 22(1)
	STH	4, 18(2)

:
H'19'
TREE
OLD+LEAF
LBL#14
I
H'19'
TREE
OLD+LEAF

NAME VARIABLES (CONTINUED)

```

1 N+ CPL:
1 N+ EXTERNAL COMPOOL;
2 N+ STRUCTURE, Q DENSE:
2 N+   1 B1 BIT(3),
2 N+   1 B2 BIT(5),
2 N+   1 S SCALAR,
2 N+   1 V VECTOR,
2 N+   1 M MATRIX,
3 N+ DECLARE Q Q-STRUCTURE(10);
4 N+ CLOSE;

5 N+ PROG:
5 N+ PROGRAM;
6 N+ DECLARE NQ NAME Q-STRUCTURE;
7 N+ DECLARE ARRAY(10),
7 N+   SCAL SCALAR,
7 N+   BITS BIT(5);
8 N+ DO FOR TEMPORARY I = 1 TO 10;
9 N+   SCAL = S + V + M
          1   I;   I; 2   I; 2,3
          ST
          ET
10 N+   BITS = B1 + B2
          -- I;   I;   I; 2 AT 2
          ST
11 N+ END;
12 N+ DO FOR TEMPORARY I = 1 TO 10;
          +
          ET
13 N+   NAME(NQ) = NAME(Q );
          I;
14 N+   SCAL = NO. S + NO. V + NO. M
          I           2           2,3
          ST
          ET
15 N+   BITS = NO. B1 + NO. B2
          I;           2 AT 2
          ST
16 N+ END;

```

NAME VARIABLES (CONTINUED)

0000009		ST#8	EQU	*	
00009 BFE3			LFXI	7,1	
0000A DF92	0021		BC	7,+39	LBL#5
000000B		LBL#6	EQU	*	
000000B		ST#9	EQU	*	
0000B 1EE7			LR	6,7	H'28'
0000C 9EF9 0026	0026		NIH	6,38(1)	
0000E 1DE6			LR	5,6	
0000F F505			SRA	5,1	
00010 F605			SRA	6,1	BASE+REG#2
00011 9A91	0024		LH	2,36(1)	
00012 78F6 C000	0000		LE	0,0(6,2)	
00014 50F6 8004	0004		RE	0,4(5,2)	
00016 1EE7			LR	6,7	
00017 9EF9 0026	0026		NIH	6,38(1)	H'28'
00019 F695			SRA	6,1	
0001A 50F6 C012	0012		RE	0,19(6,2)	
0001C 33F5 E00C	000C	ST#10	STE	0,12(7,1)	SCAL.
000001E			EQU	*	
0001F 1EE7			LR	6,7	
0001F 9F19 0026	0026		NIH	6,38(1)	H'28'
00021 9DF0 C01F	001F		LH	5,26(6,2)	
00023 F516			SRL	5,5	
00024 B6E5 0007			NHI	5,7	
00026 F508			SLL	5,2	
00027 9CF6 C01A	001A		LH	4,26(6,2)	
00029 F40A			SRL	4,2	
0002A B6E4 0003			NHI	4,3	
0002C 2DE4			OR	5,4	
0002D BDF5 E002	0002		STH	5,2(7,1)	BITS
000002F		ST#11	EQU	*	
0002F B0E7 0001			RHI	7,1	
0000031		LBL#5	EQU	*	
00031 BF48	0012		STH	7,18(0)	I
00032 B5E7 000A			CHI	7,10	
00034 DEAA	000B		BC	5,+41	LBL#5
0000035		LBL#7	EQU	*	

36 HW

NAME VARIABLES (CONTINUED)

```

0000075
00005 8FED
00006 0F70      0053
0000037
0000037
00007 9FF9 0026      0026
00009 9A91      0024
0000A ECF6 E000      0000
0000C 5C09      0002
0000030

0000D 9E48      0012
0000E 9A09      0002
0000F 7802      0000
00010 500A      0004
00011 5026      0012
00012 78F5 C00C      000C
0000044
00014 9DCA      001A
00015 F516
00016 B6E5 0007
00018 F508
00019 9C6A      001A
0001A F40A
0001B B6E4 0003
0001D 20E4
0001E BDF5 C002      0002
0000050
00050 1FE6
00051 B0E7 0001
0000053
00053 BF48      0012
00054 B5E7 000A
00056 DE82      0037
    
```

```

ST#12 EQU *
      LFXI 7,1
      BC 7,#+29
LBL#9 EQU *
ST#13 EQU *
      NIH 7,38(1)
      LH 2,36(1)
      LA 4,0(7,2)
      STH 4,2(1)
ST#14 EQU *
      LH 6,18(0)
      LH 2,2(1)
      LE 0,0(2)
      RE 0,4(2)
      RE 0,18(2)
      STE 0,12(6,1)
ST#15 EQU *
      LH 5,26(2)
      SRL 5,5
      NHI 5,7
      SLL 5,2
      LH 4,26(2)
      SRL 4,2
      NHI 4,3
      OR 5,4
      STH 5,2(6,1)
ST#16 EQU *
      LR 7,6
      RHI 7,1
LBL#8 EQU *
      STH 7,18(0)
      CHI 7,10
      BC 6,#+31
    
```

```

LBL#8
H(28)
BASE+REG#2
NO
I
NO
SCAL
BITS
I
LBL#9
    
```

25
HW

NAME VARIABLES (CONTINUED)

%NAMECOPY

%NAMECOPY(α , β);

PERFORMS THE EQUIVALENT OF:

NAME(α) = NAME(β); (*)

EXCEPT

α MUST BE A NAME STRUCTURE AND β IS EITHER A STRUCTURE OR
A NAME STRUCTURE.

Q. SINCE THIS IS MORE RESTRICTIVE THAN A REGULAR NAME ASSIGN
(LIKE (*)), WHAT IS THE ADVANTAGE?

A. NO TEMPLATE, COPYNESS, OR LENGTH CHECKING IS DONE.

%NAMECOPY ALLOWS UTILIZATION OF A SINGLE MEMORY AREA FOR
A NUMBER OF DISTINCT PURPOSES.

NAME VARIABLES (CONTINUED)

NATURALLY, %NAMECOPY MUST BE USED WITH GREAT CAUTION.

EXAMPLE:

```
STRUCTURE Q:
  1 A SCALAR DOUBLE;
DECLARE Q Q-STRUCTURE(50);
STRUCTURE R:
  1 B SCALAR;
DECLARE R NAME R-STRUCTURE(100);
. . .
%NAMECOPY(R,Q);          /* THE A AND B ARRAYS NOW ARE PHYSICALLY
. . .                    COINCIDENT */
A$5 = A$3; }
. . .
B$9 = B$5; }
B$10 = B$6; }
```

BOTH ACCOMPLISH THE SAME THING!

NAME VARIABLES (CONTINUED)

```

1 N↑ CPL:
1 N↑ COMPOOL RIGID;
2 N↑ DECLARE I ARRAY(10) INTEGER INITIAL(1, 3, 5, 7, 9, 11, 13, 15, 17, 19);
3 N↑ DECLARE VECT1 VECTOR INITIAL(10, 20, 30);
4 N↑ DECLARE VECT2 VECTOR INITIAL(40, 50, 60)
5 N↑ STRUCTURE INT RIGID:
5 N↑     1 TYPE INTEGER,
5 N↑     1 OP+CODE INTEGER,
5 N↑     1 VAR+NAME1 NAME INTEGER,
5 N↑     1 VAR+NAME2 NAME INTEGER;
6 N↑ STRUCTURE VECT RIGID:
6 N↑     1 TYPE INTEGER,
6 N↑     1 OP+CODE INTEGER,
6 N↑     1 VAR+NAME1 NAME VECTOR,
6 N↑     1 VAR+NAME2 NAME VECTOR;
7 N↑ DECLARE S1 INT-STRUCTURE INITIAL(1, 1, NAME(I.), NAME(I.));
      S↑
8 N↑ DECLARE S2 INT-STRUCTURE INITIAL(1, 2, NAME(I.), NAME(I.));
      S↑
      E↑
9 N↑ DECLARE S3 VECT-STRUCTURE INITIAL(2, 1, NAME(VECT1), NAME(VECT2));
10 N↑ DECLARE S4 INT-STRUCTURE INITIAL(1, 3, NAME(I.), NAME(I.));
      S↑
      E↑
11 N↑ DECLARE S5 VECT-STRUCTURE INITIAL(2, 2, NAME(VECT1), NAME(VECT2));
12 N↑ CLOSE.

```

NAME VARIABLES (CONTINUED)

```

13 N+ INTERP
13 N+ PROGRAM;
14 N+ DECLARE N+INT NAME INT-STRUCTURE(5);
15 N+ DECLARE N+VECT NAME VEC1-STRUCTURE(5);
    E+
16 N+ XNAMECOPY(N+INT, S1);
    E+
17 N+ XNAMECOPY(N+VECT, S1);
18 N+ DO FOR TEMPORARY I = 1 TO 5;
19 N+   IF N+INT.TYPE = 1 THEN
20 N+     DO;
21 N+       DO CASE N+INT.OP+CODE ;
22 N+         DO;
23 N+           WRITE(6) N+INT.VAR+NAME1 + N+INT.VAR+NAME2 ;
24 N+         END;
25 N+         DO;
26 N+           WRITE(6) N+INT.VAR+NAME1 - N+INT.VAR+NAME2 ;
27 N+         END;
28 N+         DO;
29 N+           WRITE(6) N+INT.VAR+NAME1 * N+INT.VAR+NAME2 ;
30 N+         END;
31 N+       END;
32 N+     END;
33 N+   ELSE
34 N+     DO;
35 N+       DO CASE N+VECT.OP+CODE ;
36 N+         DO;
37 N+           WRITE(6) N+VECT.VAR+NAME1 . N+VECT.VAR+NAME2 ;
38 N+         END;
39 N+         DO;
40 N+           WRITE(6) N+VECT.VAR+NAME1 * N+VECT.VAR+NAME2 ;
41 N+         END;
42 N+       END;
43 N+     END;
10-88 44 N+ END;

```

/* INTEGER TYPE */
/* ADDITION */
/* SUBTRACTION */
/* MULTIPLICATION */
/* VECTOR TYPE */
/* DOT PRODUCT */
/* CROSS PRODUCT */

NAME VARIABLES (CONTINUED)

HAL/S-360 V14.0 START TIME: 22:43:09.61 DAY: 76/075
FARM I.E.D. MCHAN=1

HCP1 HAL/S DRIVER FOR IBM SHUTTLE GPC SIMULATOR "SIM101"

HCP1: LOAD: GO.FND

HCP1: THE FOLLOWING LOAD PARAMETERS ARE IN EFFECT

HCP1: LDNAM= LOADMOD

HCP1: MEMBER= 00

HCP1: REGION=58K BYTES (360 CORE)

HCP1: SIMREGION=4K HALFWORDS (AP101 CORE)

HCP1: SIMPAGE3=4 (CORE RESIDENT 1K PAGES)

HCP1: HAL/S-FC MODULE LOADED

10
-4
3.2000000E+01
3

HCP1: HAL/S DRIVER FOR IBM SHUTTLE GPC SIMULATOR "SIM101"

MESSAGE # 65 DURING SIMULATION INTERRUPT #18 OLD PSW=01E74A00 000

HCP1: SVC OLD PSW = 01E74A00 000101FB

HCP1: INSTRUCTION COUNT 336

HCP1: SIMULATION TIME 7.39199999999999900E-04

HCP1: END OF HAL/S-FC PROGRAM

*** SUMMARY OF ERRORS ***

ERROR SYSTEM (STATEMENT)

10: 0 1.1

-3.0000000E+02 6.0000000E+02 -3.0000000E+02
RETURN CODE 0

HAL/S

21
500-50

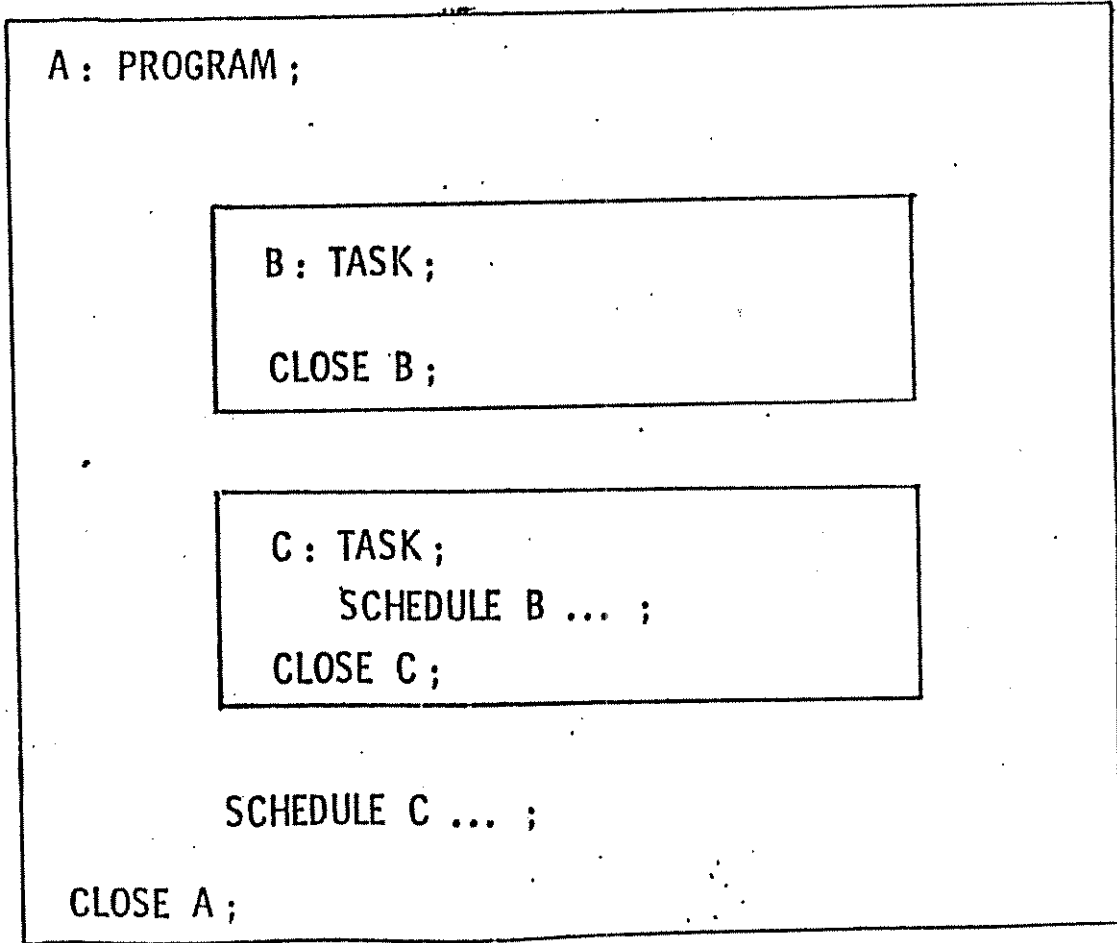
FOR REAL TIME PROCESS CONTROL

- High-Level Problem-Oriented Real Time Features
- Careful Definition of Semantics -- for Easy User Comprehension
- Guaranteed Orderly Access to Shared Data and Code
- Hierarchical Relationships Among Processes

HAL/S

82
500-57

PROGRAM and TASK Blocks



14-2

HAL/S

33
500-52

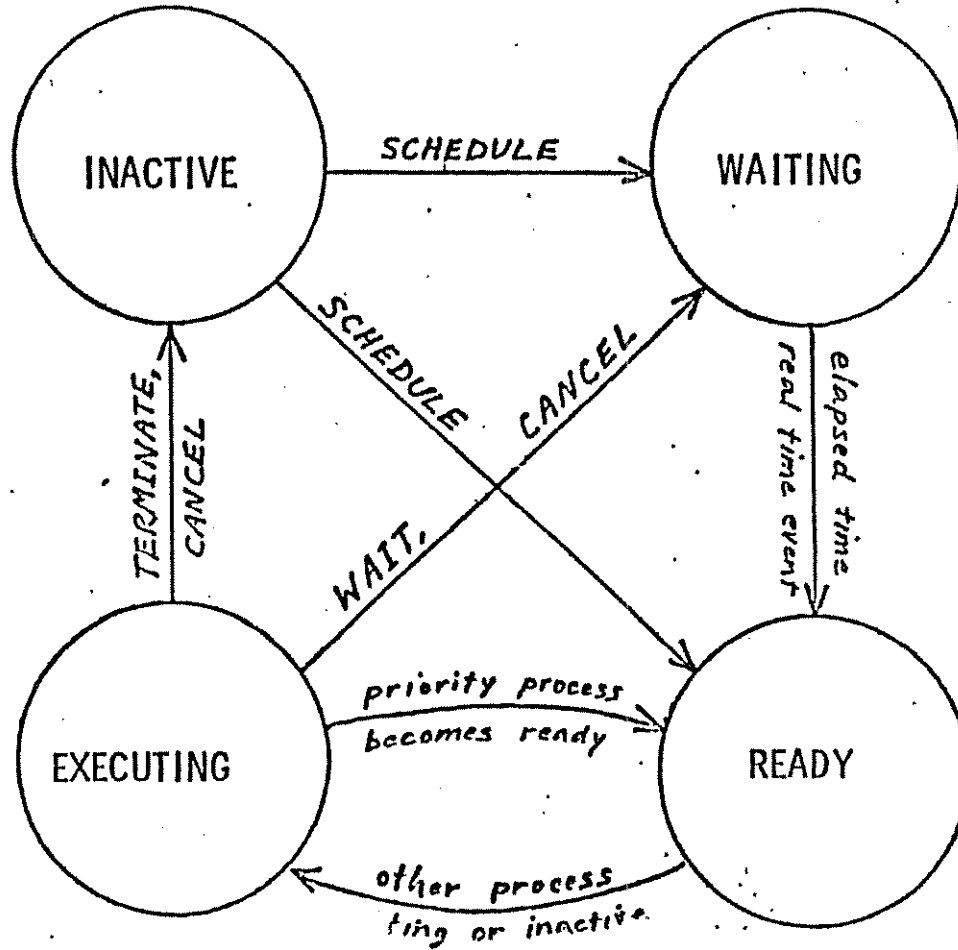
A PROCESS IS :

- The Execution of a PROGRAM or TASK Block
- Made ACTIVE through Execution of a SCHEDULE Statement
- Made INACTIVE through Execution of a CLOSE, CANCEL or TERMINATE Statement

HALS

Process State Transitions

SI
500-53

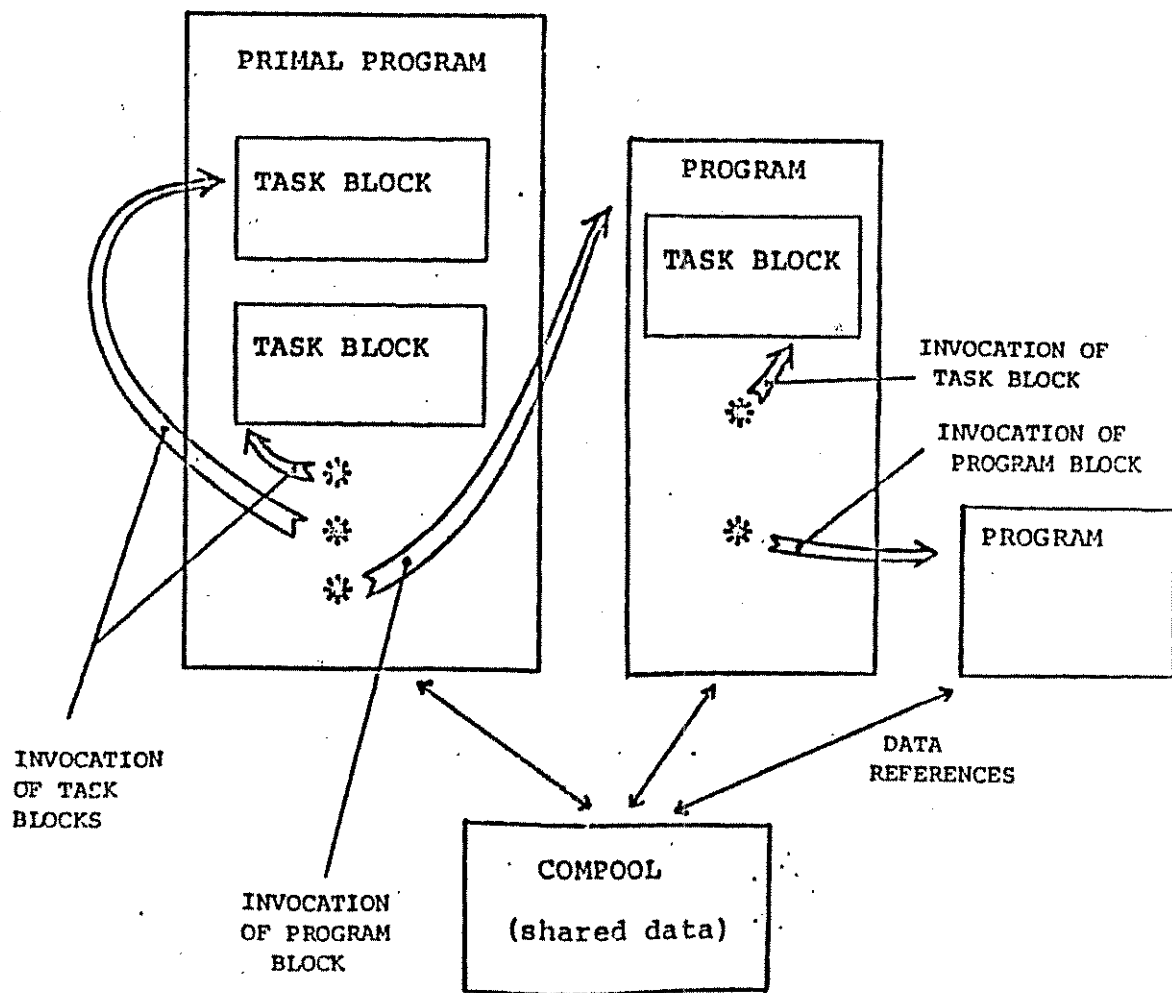


14-20

HAL/S

32
500-54

Scheduling of PROGRAM and TASK Blocks



14-5

HAL/S

52
500-55

EACH PROCESS HAS A PRIORITY
(An Integer Value)

- Initially Specified in SCHEDULE Statement:

SCHEDULE ALPHA PRIORITY (50);

- Dynamically Changed by UPDATE PRIORITY Statement:

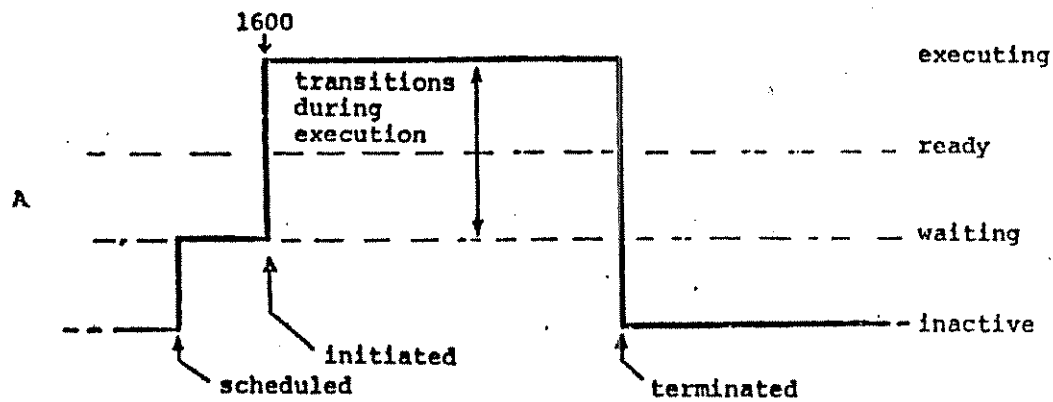
UPDATE PRIORITY ALPHA TO (1 + 5);

14-6

HAL/S

SCHEDULING IN REAL TIME

53
800-52



SCHEDULE A AT 1600 PRIORITY (50);

Alternatively,

SCHEDULE process IN Interval, PRIORITY (n);

HAL/S

S2
500-57

THE WAIT STATEMENT

- Process Is Temporarily Placed in the WAITING State
- WAIT Interval ;
- WAIT UNTIL time ;

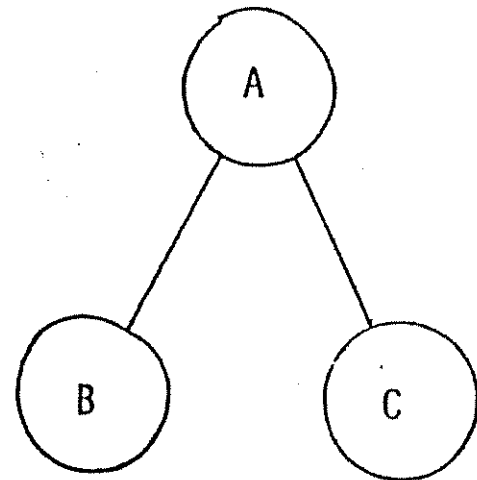
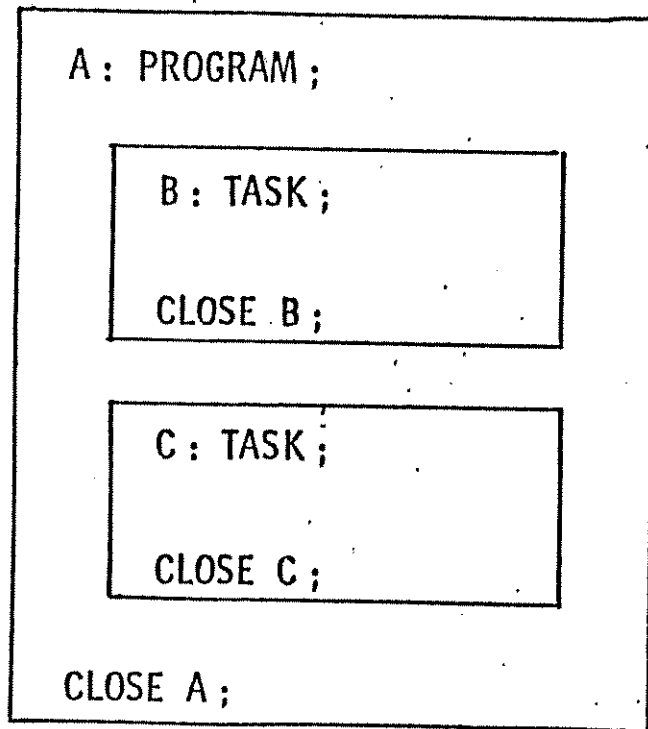
WAIT UNTIL IGNITION + 5 ;

14-0

HAL/S

PROCESS DEPENDENCY

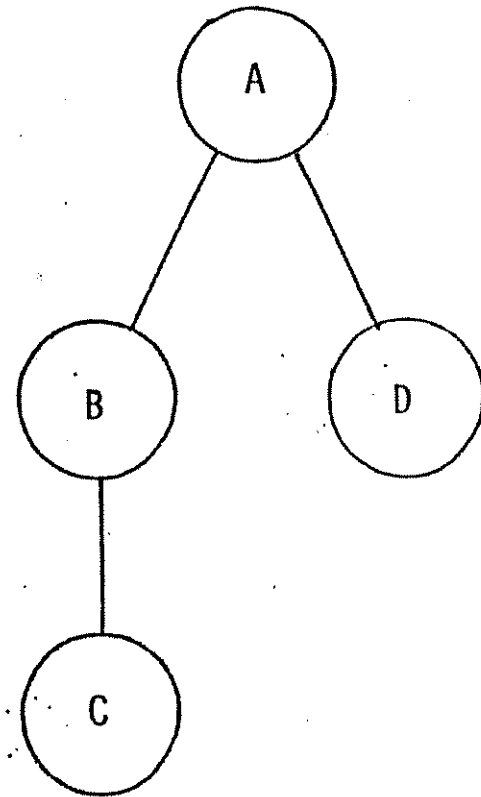
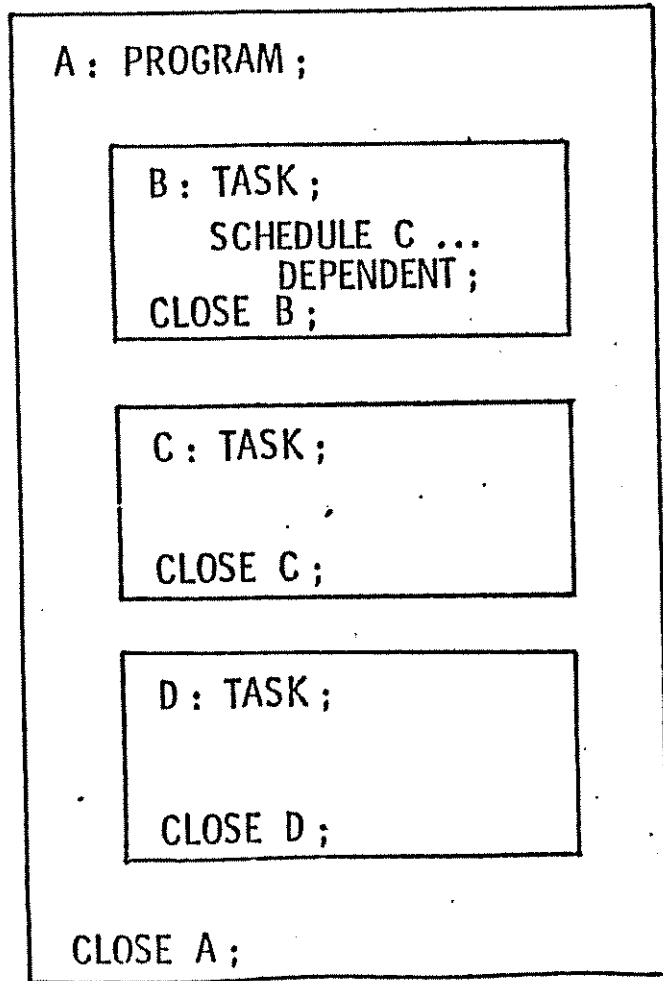
53
500-58



HAL/S

FURTHER PROCESS DEPENDENCY

52
500-59



14-10

HAL/S

CYCLIC SCHEDULING

S3
500-60

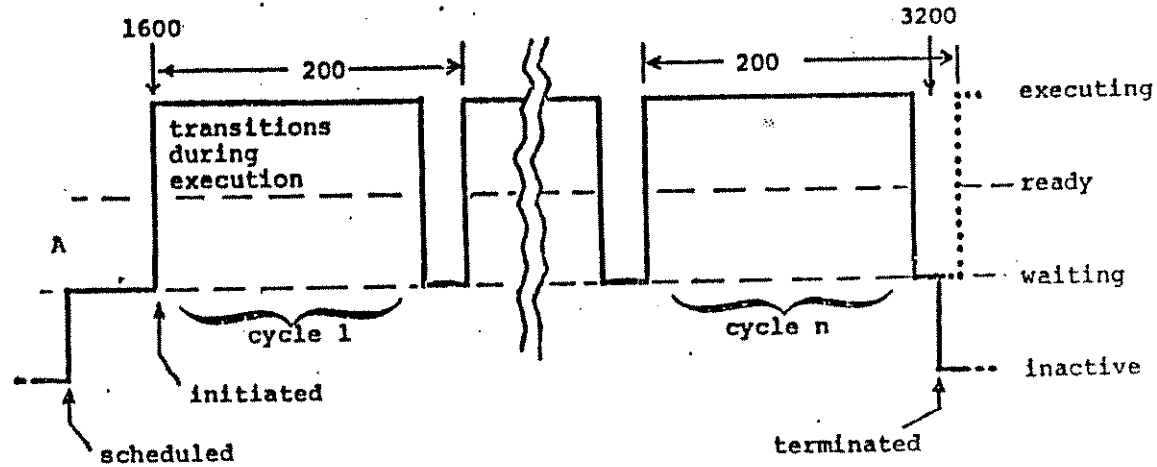
A Process can be Scheduled to Execute
Its PROGRAM or TASK Block Repeatedly,
Until it is :

- Terminated by Execution of a TERMINATE Statement
- "Cancelled" through Execution of a CANCEL Statement, or Because a "Cancellation Condition" has been Met

HAL/S

RECYCLING AT FIXED INTERVALS

52
500-61



SCHEDULE A AT 1600 PRIORITY (50), REPEAT EVERY 200 UNTIL 3200;

SEALS

PROCESS SYNCHRONIZATION

52
500-62

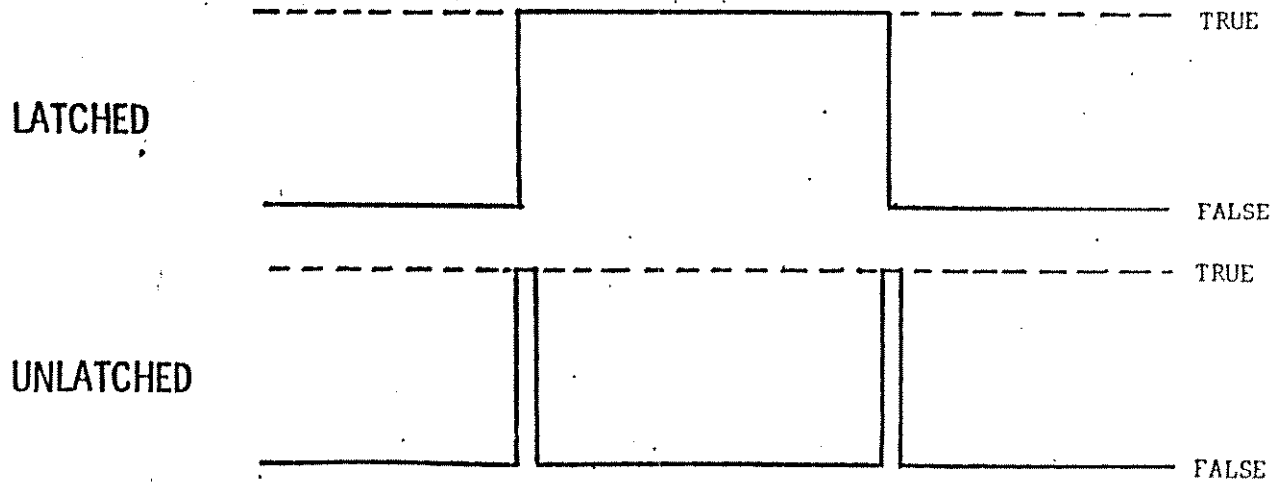
EVENT Variables

- Used for Process Synchronization
- Value TRUE or FALSE
- Combined with Operators AND, OR, NOT to Form EVENT Expressions

HALS

S3
500-63

EVENT Variables May Be "Latched" or "Unlatched"



HAL/S

S2
500-64

SET, SIGNAL, and RESET

- Must Be Used to Change Values of EVENT Variables
- Cause Re-evaluation of EVENT Expressions
- SET and RESET for LATCHED Variables
- SIGNAL for LATCHED or UNLATCHED Variables

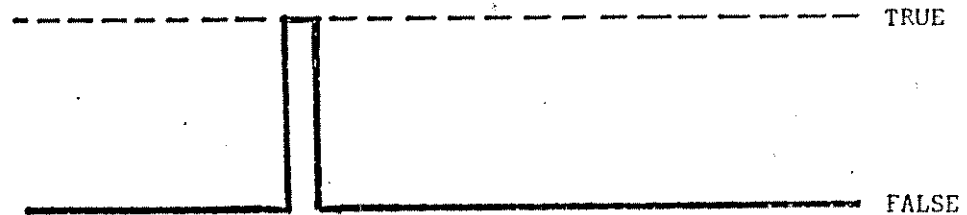
HAL/S

S3
500-65

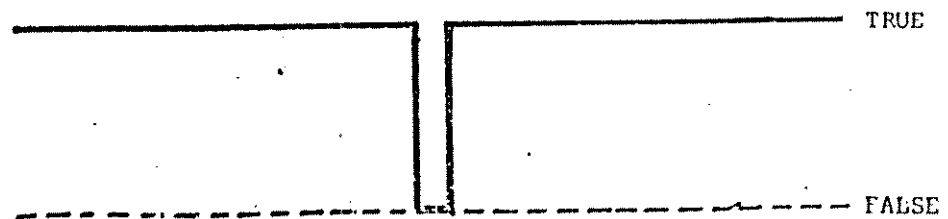
Effect of SIGNAL

DECLARE EV1 EVENT,
EV2 EVENT LATCHED INITIAL (TRUE);

SIGNAL EV1;



SIGNAL EV2;



HALS

23
500-57

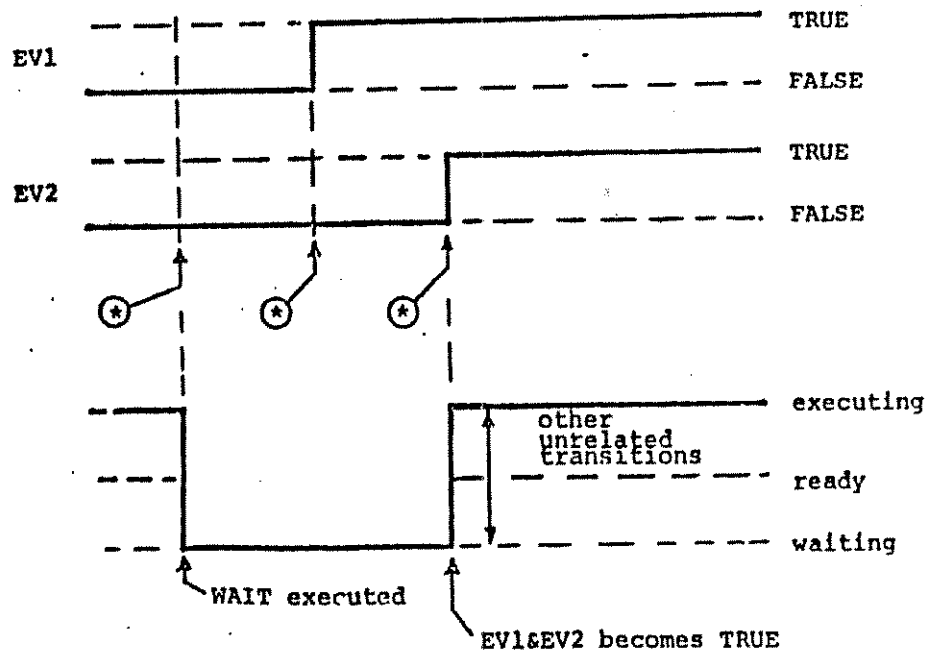
Statement	Event	T = TRUE, F = FALSE	
		Actual Value	Change sensed by RTE
SET	latched	execution ↓	
		T --- T	F + T
		F --- F	
		T --- T	none
RESET	latched	T --- T	none
		F --- F	
		T --- T	T + F
		F --- F	
SIGNAL	latched	T --- T	F + T
		F --- F	
		T --- T	T + F
	unlatched	T --- T	F + T
		F --- F	
		F --- F	F + T

14-17

HAL/S

52
500-66A

WAIT FOR EVENT-Expression ;

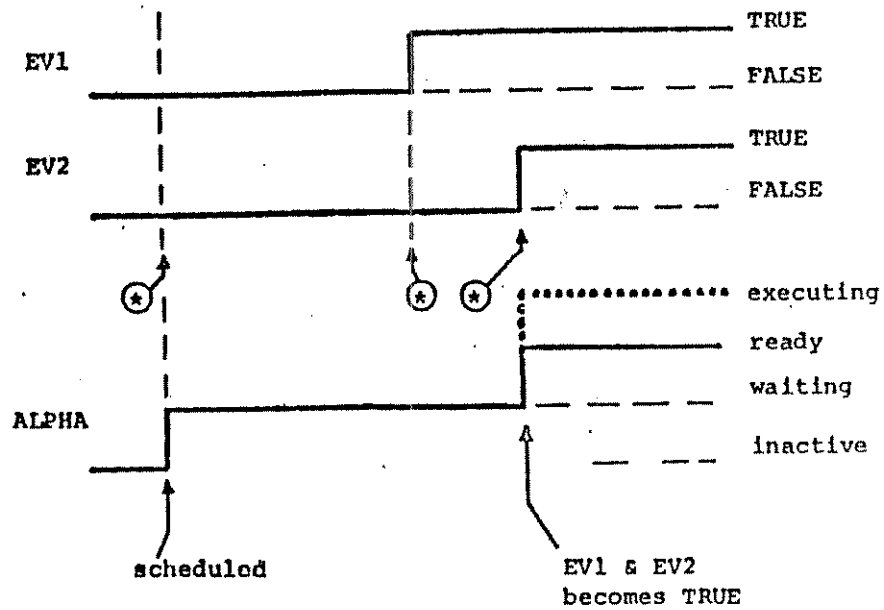


WAIT FOR EV1 & EV2 ;

HAL/S

INITIATION ON AN EVENT CONDITION

52
500-67

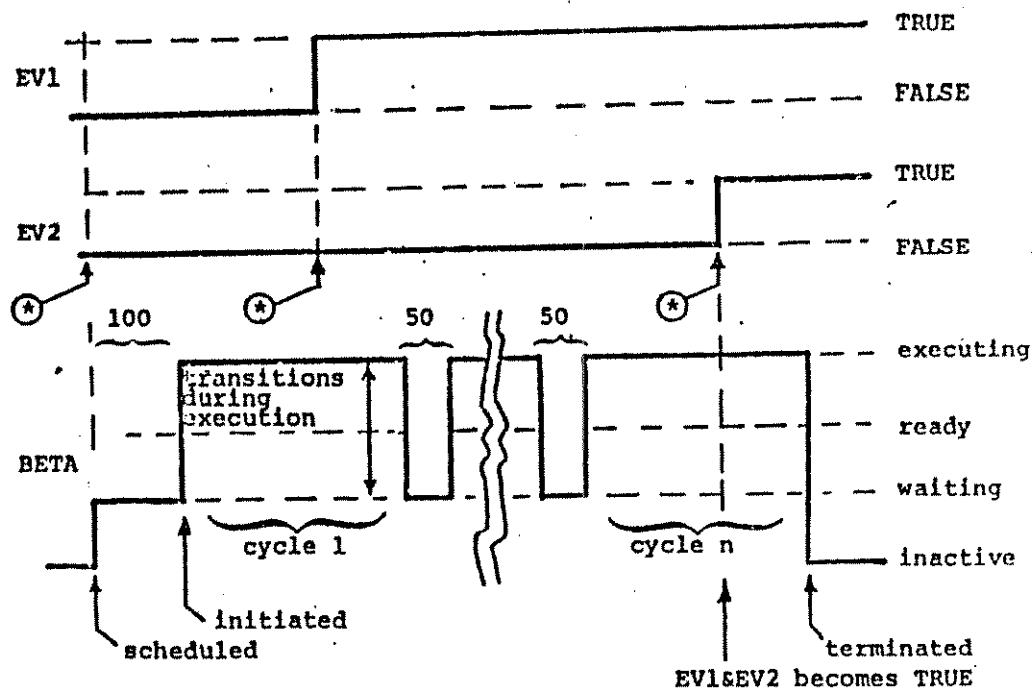


SCHEDULE ALPHA ON EV1 & EV2 PRIORITY (50);

52
500-68

HAL/S

CANCELLATION ON AN EVENT CONDITION



SCHEDULE BETA IN 100 PRIORITY (.50), REPEAT AFTER 50
UNTIL EV1 & EV2;

14-3

3

HAL/S

Shared Data Protection for Dynamic Processes

SI
500-67

DATA MUST BE PROTECTED AGAINST
SIMULTANEOUS ACCESS AND MODIFICATION

- Data Is Accessed In an Orderly Way
- Results are Not Dependent on Timing Coincidences

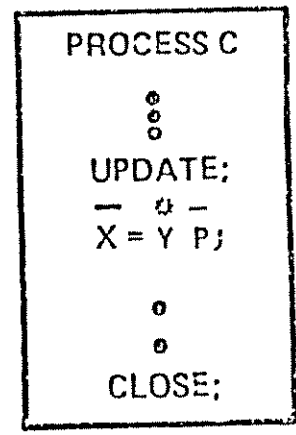
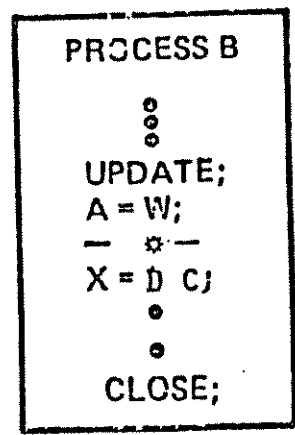
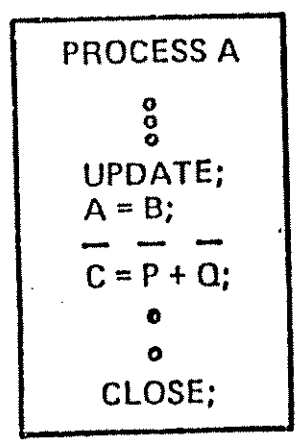
DATA IS PROTECTED BY MEANS OF:

- LOCK Groups
- UPDATE Blocks

HAL/S

Shared Data Protection for Dynamic Processes

DECLARE LOCK (1), A, B INTEGER, C VECTOR, D MATRIX;
 DECLARE LOCK (2), W, X VECTOR, Y MATRIX;



SWAP TO:

DYNAMICS → ACTIVE

	A	B	C
A	-	STALL	RUN
B	STALL	-	STALL
C	RUN	STALL	-

HAL/S

Shared Data Protection for Dynamic Processes

53
500-71

PROTECTION IS GUARANTEED

- Compiler Allows References to LOCKED Variables Only in UPDATE Blocks
- RTE Reserves LOCK Groups at Entry to UPDATE Block
- Simultaneous Reservation of All Required LOCK Groups Avoids Deadlock

14-22

DEALS

SI
500-72

A SHARED BLOCK MAY BE PROTECTED :

- By Problem-Related Synchronization Measures
- By Preventing Concurrent Execution
- By Allowing Concurrent Execution In Independent Environments

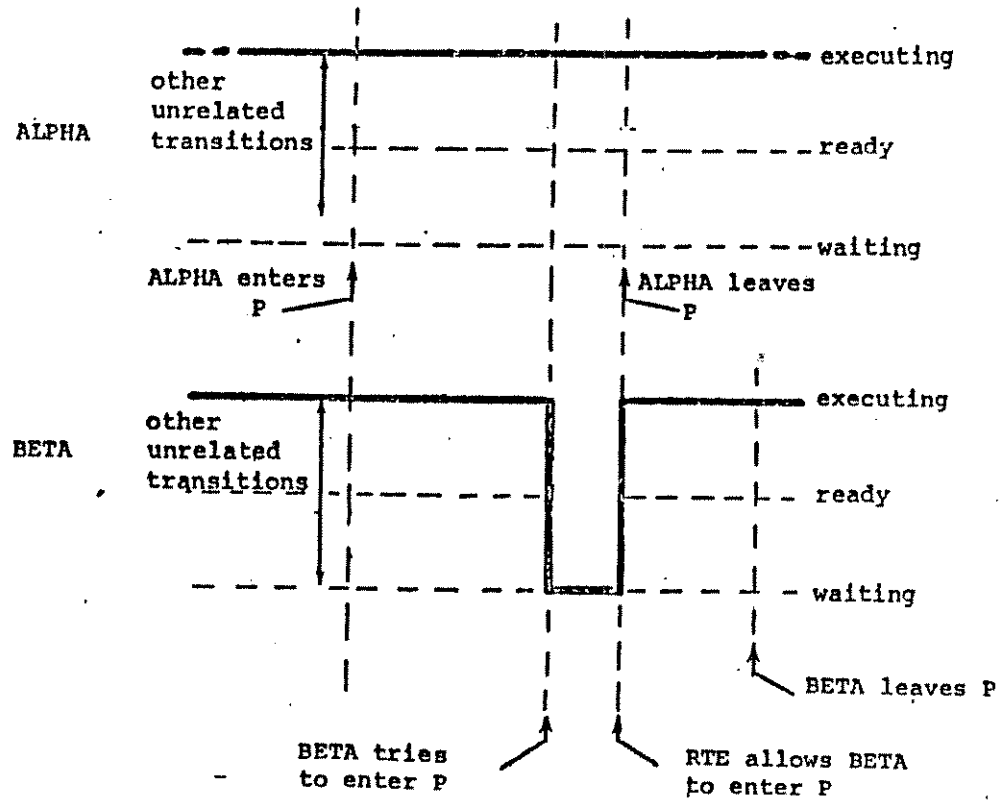
THESE PROTECTION METHODS APPLY TO :

- Ordinary PROCEDURES and FUNCTIONS
- EXCLUSIVE PROCEDURES and FUNCTIONS
- REENTRANT PROCEDURES and FUNCTIONS

HAL/S

Action for EXCLUSIVE Blocks

52
500-73

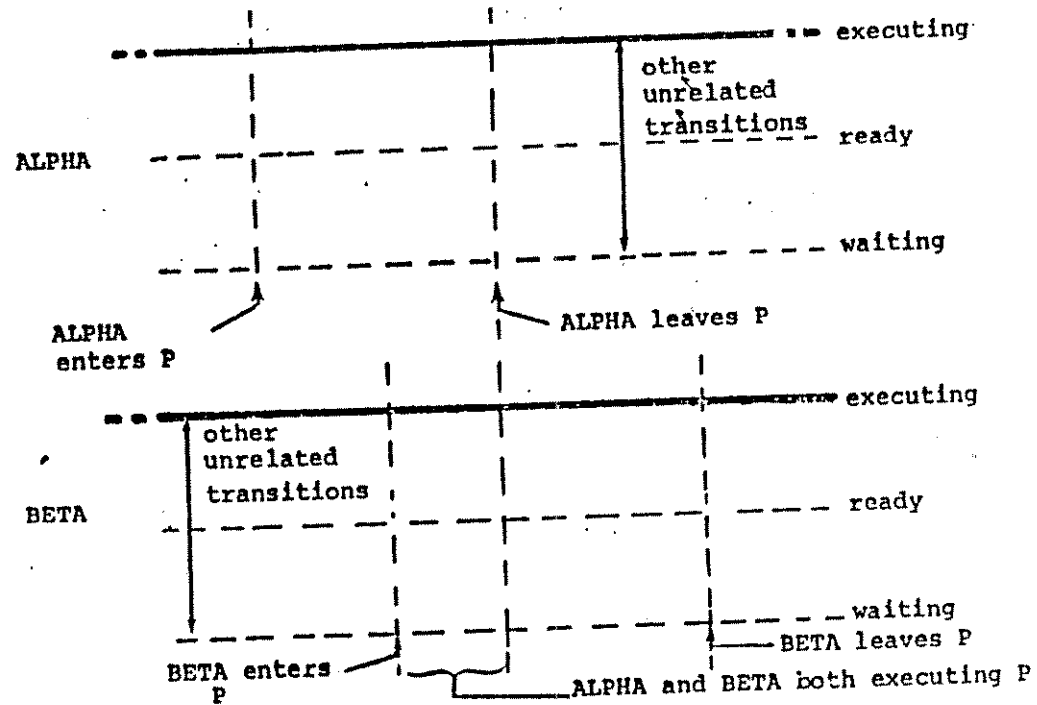


P: PROCEDURE EXCLUSIVE;

HAL/S

Action for REENTRANT Blocks

53
500-74



P : PROCEDURE REENTRANT ;

14-2-74

HAL/S

SI
500-74A

LOCAL DATA IN REENTRANT BLOCKS

- **STATIC** Data Is Unique to the Block and Common to All Invocations
- **AUTOMATIC** Data Is Unique to Each Invocation of the Block
- Parameters and Compiler Temporaries are Effectively **AUTOMATIC**

HAL/S

Conclusions

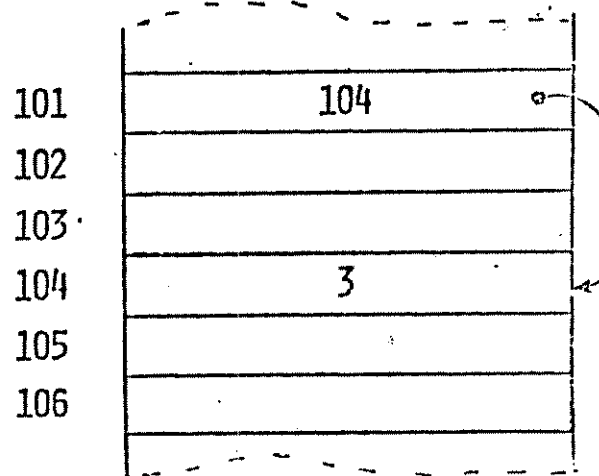
HAL/S Has Been Designed and Implemented to Enhance Software Reliability

SUMMARY OF REAL TIME FEATURES :

- Program Structure and Name Scope
- Process States, Priority, Real Time, Dependency
- Cyclic Scheduling
- EVENT Expressions, SET, SIGNAL, and RESET
- LOCK Groups and UPDATE Blocks
- EXCLUSIVE and REENTRANT Code Blocks

NAME VARIABLES

HARDWARE LEVEL



HAL/S LEVEL

```
DECLARE I INTEGER INITIAL(3),  
        NI NAME INTEGER INITIAL (NAME(I));  
NI = NI + 1;  
/* SAME AS I = I + 1; */
```

USES FOR NAME VARIABLES

- MOVE AROUND A POINTER TO A BLOCK OF DATA (OR CODE) RATHER THAN THE WHOLE BLOCK.
- ACCOMMODATE DATA STRUCTURES OF DYNAMICALLY VARYING SIZE IN A FIXED-SIZE MEMORY.

PROBLEMS WITH UNRESTRICTED POINTERS

```
DECLARE I INTEGER,  
        S SCALAR,  
        N NAME;  
NAME(N) = NAME(I);           /* POINT N AT I */  
L: N = N + 1;                /* WHAT CODE IS COMPILED? */  
NAME(N) = NAME(S);          /* POINT N AT S */  
IF N < S THEN GO TO L;      /* CONVERSION REQUIRED? */
```


SO, A NAME VARIABLE IS DECLARED TO POINT TO VARIABLES
OF A GIVEN DATA TYPE ...

```
DECLARE NI NAME INTEGER,  
        NS NAME SCALAR,  
        NV NAME VECTOR(4),  
        NA NAME ARRAY(2,2) BOOLEAN;
```

```
DECLARE INTEGER DOUBLE,  
        I, J, K,
```

```
        N NAME;
```

```
/* FACTORED ATTRIBURES */
```

HENCE THE COMPILER CAN CHECK FOR TYPE COMPATIBILITY.

```
DECLARE I INTEGER,  
        S SCALAR,  
        N NAME INTEGER;  
NAME(N) = NAME(I),           /* POINT N AT I */  
L: N = N + 1,                /* CODE FOR INTEGER ADDITION */  
NAME(N) = NAME(S),          /* ERROR-TYPE MISMATCH */  
IF N < S THEN GO TO L,     /* N DEFERENCED AND CONVERTED  
                            TO SCALAR */
```

NAME VARIABLES CAN POINT TO DATA (OR CODE) WITH
THE FOLLOWING ATTRIBUTES:

SINGLE OR DOUBLE	OR *	
{	INTEGER	CHARACTER(n)
	SCALAR	BIT(n)
	VECTOR(n)	BOOLEAN
	MATRIX(n,m)	EVENT
		ARRAY(n,m,l)
		α-STRUCTURE(n)
		PROGRAM
		TASK

```

DECLARE MATRIX(2,3), A, B, NB NAME INITIAL(NAME(B));
DECLARE MATRIX(3,3), C, D, ND NAME INITIAL(NAME(D));
A = NB; /* SAME AS A = B; */
A = ND; /* ERROR-DIMENSION MISMATCH */
NAME(NB) = NAME(ND); /* ERROR-DIMENSION MISMATCH */
NB = ND$(2 AT 1,*); /* SAME AS B = D$(2 AT 2,*); */

```

RESTRICTIONS

- NAME VARIABLES CANNOT POINT TO NAME VARIABLES

BUT

 THEY MAY POINT TO STRUCTURES CONTAINING NAME VARIABLES
- CANNOT DECLARE AN ARRAY OF NAME VARIABLES

BUT

 THEY MAY APPEAR IN A STRUCTURE WITH COPIES
- (SUBTLE) A NAME VARIABLE IN A STRUCTURE WITH COPIES CANNOT POINT TO A STRUCTURE WITH COPIES - CONVERSELY, A NAME VARIABLE WHICH POINTS TO A STRUCTURE WITH COPIES CANNOT BE IN A STRUCTURE WITH COPIES.

OTHER ATTRIBUTES FOR NAME VARIABLES

DENSE

ALIGNED

RIGID

STATIC

AUTOMATIC

INITIAL()

CONSTANT()



THESE APPLY TO THE NAME
VARIABLE ITSELF, NOT TO THE
VARIABLE WHICH IT POINTS TO.

TO BE DESCRIBED

15-5

INITIALIZATION OF NAME VARIABLES

{ INITIAL }
{ CONSTANT } (NAME, REFERENCE, NAME REFERENCE, ...)

EACH NAME REFERENCE HAS THE FORM:

NAME(ORDINARY VARIABLE)

NAME(NULL) OR NULL

"NULL POINTER" - POINTS TO
NOTHING AT ALL. UNINITIALIZED
NAME VARIABLES ALSO HAVE THIS
VALUE:

ORDINARY VARIABLE MUST BE:

- PREVIOUSLY DECLARED
- WITH DATATYPE MATCHING
THAT OF NAME VARIABLE

AND MUST NOT BE:

- SUBSCRIPTED
- DEREFERENCED THROUGH ANOTHER
NAME VARIABLE

INITIALIZATION OF NAME VARIABLES

1. DECLARE S SCALAR,
NS NAME SCALAR INITIAL(NAME(S));
2. NAME VARIABLES IN STRUCTURES WILL BE DISCUSSED LATER.
STRUCTURE A:
1 B SCALAR,
1 C NAME A-STRUCTURE,
DECLARE A-STRUCTURE,
Z1,
Z2 INITIAL(1.5, NAME(Z1));
DECLARE NB NAME SCALAR INITIAL(NAME(Z1.B));
3. DECLARE A ARRAY(5) INTEGER INITIAL(0),
NI NAME INTEGER INITIAL(NAME(A₃));

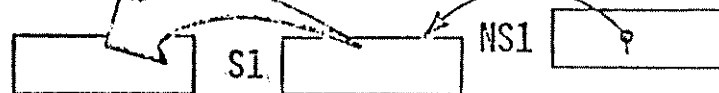
THE NAME PSEUDO-FUNCTION

- ORDINARY REFERENCE TO NAME VARIABLE ACCESSES THE VARIABLE WHICH IT POINTS TO.
- NAME PSEUDO-FUNCTION IS USED TO ACCESS OR CHANGE THE (POINTER) VALUE OF THE NAME VARIABLE ITSELF.

```
DECLARE SCALAR, S1, S2, NS NAME, NS1 NAME INITIAL(NAME(S2));
```

```
S2 = NS1;
```

S2

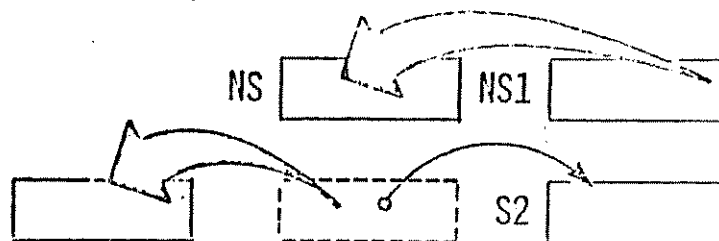


```
NAME(NS) = NAME(NS1);
```

```
..
```

```
NAME(NS) = NAME(S2);
```

NS



```
NAME(NS) = NS1;
```

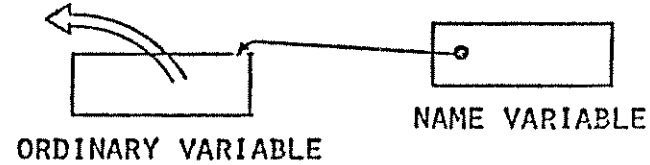
```
NS = NAME(NS1);
```

}

ERROR-TYPE MISMATCH!

IN GENERAL:

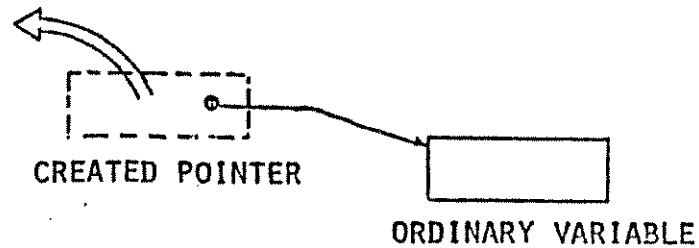
- NAME VARIABLE BY ITSELF DENOTES THE ORDINARY VARIABLE WHICH IT POINTS TO.



- NAME(NAME VARIABLE) OBTAINS THE POINTER CONTENTS OF THE NAME VARIABLE.



- NAME(ORDINARY VARIABLE) CREATES A POINTER TO THAT ORDINARY VARIABLE.

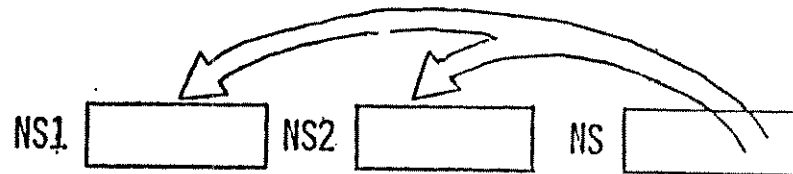


NAME ASSIGNMENTS

$$L^1, L^2, \dots, L^n = R;$$

{ EACH L^i AND R IS A NAME
PSEUDO-FUNCTION.

E.G. NAME(NS1), NAME(NS2) = NAME(NS);



NAME COMPARISONS

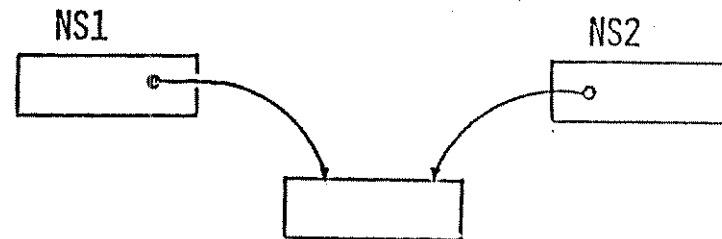
$$L = R$$

$$L \neq R$$

{ L AND R ARE BOTH NAME
PSEUDO-FUNCTIONS.

E.G. IF NAME(NS1) \neq NAME(NS2) THEN ...;

EQUALITY IF BOTH NAME VARIABLES POINT TO THE SAME
ORDINARY VARIABLE!



NAME ARGUMENTS AND PARAMETERS

```
P: PROCEDURE(NA) ASSIGN(NB);  
  DECLARE NA NAME INTEGER,  
         NB NAME ARRAY(10) SCALAR,  
  ⋮  
CLOSE P;
```

- ARGUMENT MATCHING NA: NAME PSEUDO-FUNCTION IN REFERENCE CONTEXT (OR NULL).
- ARGUMENT MATCHING NB: NAME PSEUDO-FUNCTION IN ASSIGNMENT CONTEXT.

```
DECLARE I INTEGER;  
DECLARE ARRAY(10) SCALAR, A, N NAME;  
  
CALL P(NAME(I)) ASSIGN(NAME(N));  
CALL P(NULL) ASSIGN(NAME(N));  
  
CALL P(NAME(I)) ASSIGN(NAME(A));  
CALL P(NULL) ASSIGN(NAME(N*));
```

LEGAL

ILLEGAL

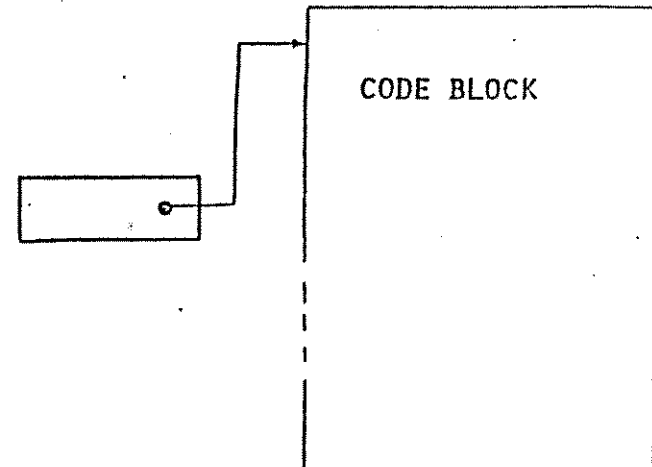
POINTERS TO CODE BLOCKS

```
P1: EXTERNAL PROGRAM;  
CLOSE P1;  
P2: EXTERNAL PROGRAM;  
CLOSE P2;  
P3: EXTERNAL PROGRAM;  
CLOSE P3;
```

TEMPLATES

```
MASTER: PROGRAM;  
  DECLARE NP NAME PROGRAM,  
           I INTEGER;  
  ⋮  
  DO WHILE TRUE;  
    DO FOR I = 1 TO 3;  
      DO CASE I;  
        NAME(NP) = NAME(P1);  
        NAME(NP) = NAME(P2);  
        NAME(NP) = NAME(P3);  
      END;  
      UPDATE PRIORITY NP TO 200;  
      WAIT 1;  
      UPDATE PRIORITY NP TO 50;  
    END;  
  END;  
  ⋮  
CLOSE MASTER;
```

NP



```
/* PROMOTE P1, P2, P3 IN TURN */  
/* TO HIGH PRIORITY FOR 1 SEC */  
/* THEN REVERT TO NORMAL PRI. */
```

NAME VARIABLES IN STRUCTURES

- NAME VARIABLES MAY POINT TO VARIABLES DECLARED WITH STRUCTURE TEMPLATES.
- NAME VARIABLES MAY BE TERMINAL NODES OF A STRUCTURE TEMPLATE DEFINITION.
- A NAME VARIABLE IN A STRUCTURE TEMPLATE MAY POINT TO THE TEMPLATE CURRENTLY BEING DEFINED.

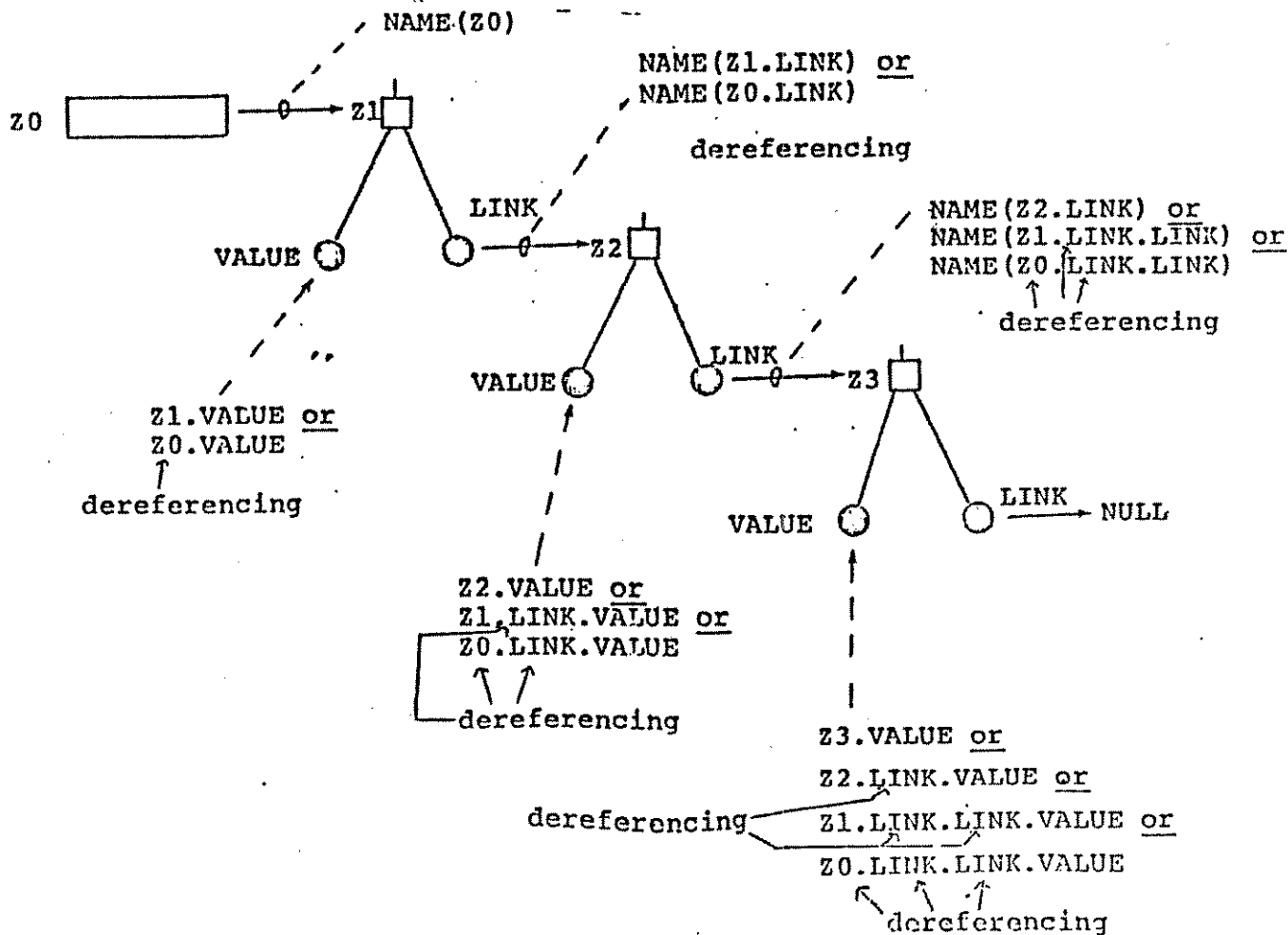
STRUCTURE LIST:

1 VALUE INTEGER,

1-LINK NAME LIST-STRUCTURE;

STRUCTURE LIST:

1 VALUE INTEGER,
 1 LINK NAME LIST-STRUCTURE;
 DECLARE LIST-STRUCTURE,
 Z1, Z2, Z3,
 Z0 NAME;



BEST WAY TO THINK OF THIS:

- ANY APPEARANCE OF A NAME VARIABLE IMPLIES DEREFERENCING.
- APPLICATION OF NAME PSEUDO-FUNCTION IMPLIES ONE LEVEL OF "RE-REFERENCING".

THUS

Z0	IS A REFERENCE TO THE STRUCTURE Z1
NAME(Z0)	"BACKS UP" TO Z0 ITSELF
Z1.LINK.LINK	IS A REFERENCE TO THE STRUCTURE Z3
NAME(Z1.LINK.LINK)	"BACKS UP" TO Z2.LINK (OR Z1.LINK.LINK) ITSELF

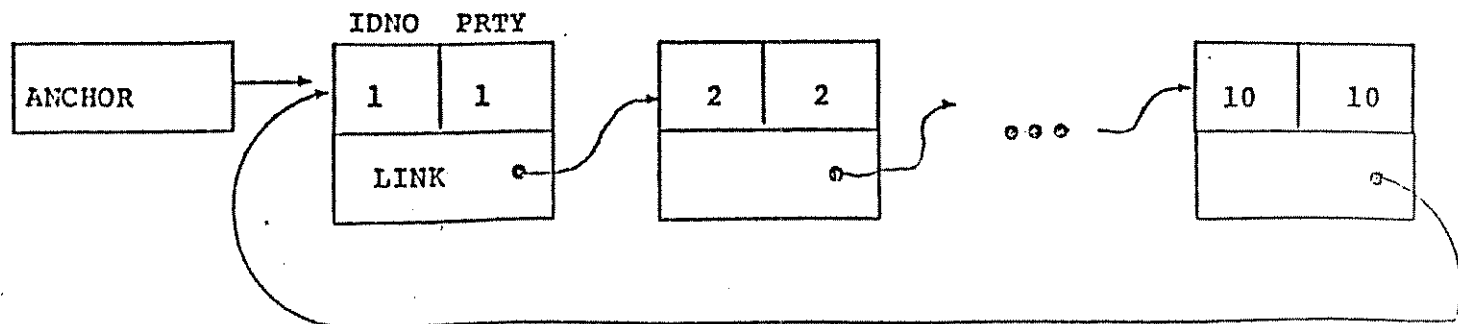
NOTE THAT Z0.VALUE IS THE SAME AS Z1.VALUE SINCE Z0 IS A NAME VARIABLE BUT Z1 IS NOT.

FURTHER, NAME(Z0.VALUE) IS THE SAME AS NAME(Z1.VALUE)
I.E. A POINTER TO THE VALUE FIED IN Z1.

USES OF STRUCTURES AND NAME VARIABLES

① A PRIORITY-ORDERED QUEUE

```
STRUCTURE QUEUE:  
  1 IDNO INTEGER,  
  1 PRTY INTEGER,  
  1 LINK NAME QUEUE-STRUCTURE;  
DECLARE QUEUE QUEUE-STRUCTURE(10),  
        ANCHOR NAME QUEUE-STRUCTURE;  
DECLARE INTEGER, I, NEW_ID, NEW_PRTY;  
  ⋮  
DO FOR I = 1 TO 9;  
  IDNOI, PRTYI = I;  
  NAME(LINKI) = NAME(QUEUEI+1);  
END;  
IDNO10, PRTY10 = 10;  
NAME(ANCHOR), NAME(LINK10) = NAME(QUEUE1);
```



USES OF STRUCTURES AND NAME VARIABLES

```
DECLARE NAME QUEUE-STRUCTURE, THIS, PREV;
```

```
/* FIND IDNO IN QUEUE */
```

```
NAME(PREV) = NAME(ANCHOR);  
NAME(THIS) = NAME(ANCHOR.LINK);
```

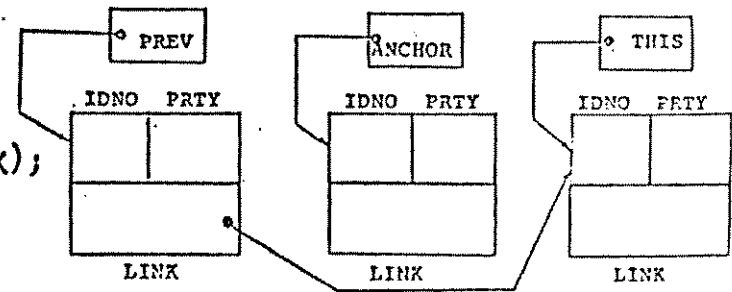
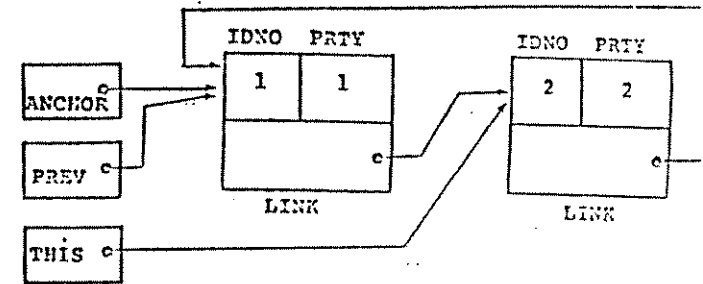
```
DO WHILE THIS.IDNO ≠ NEW_IDNO;  
  NAME(PREV) = NAME(THIS);  
  NAME(THIS) = NAME(THIS.LINK);  
END;
```

```
/* SET NEW PRIORITY, AND TEMPORARILY UNLINK FROM QUEUE */
```

```
THIS.PRTY = NEW_PRTY;  
NAME(ANCHOR), NAME(PREV.LINK) = NAME(THIS.LINK);
```

```
/* FIND PROPER PLACE TO RE-LINK IN QUEUE ACCORDING TO PRIORITY */
```

```
DO UNTIL PREV.PRTY ≤ NEW_PRTY AND NEW_PRTY ≤ THIS.PRTY;  
  NAME(PREV) = NAME(THIS);  
  NAME(THIS) = NAME(THIS.LINK);  
END;  
NAME(PREV.LINK) = NAME(ANCHOR);  
NAME(ANCHOR.LINK) = NAME(THIS);
```



USES OF STRUCTURES AND NAME VARIABLES

② TREE-STRUCTURED SYMBOL TABLE

STRUCTURE TREE:

```

1 SYMBOL CHARACTER(32),
1 LESS NAME TREE-STRUCTURE,
1 GTR NAME TREE-STRUCTURE;
DECLARE TREE TREE-STRUCTURE(100),
NEW_SYMBOL CHARACTER(32),
I INTEGER INITIAL(0),
WAS_LESS BOOLEAN;
DECLARE NAME TREE-STRUCTURE,
ROOT INITIAL(NULL), OLD_LEAF, THIS;

```

```

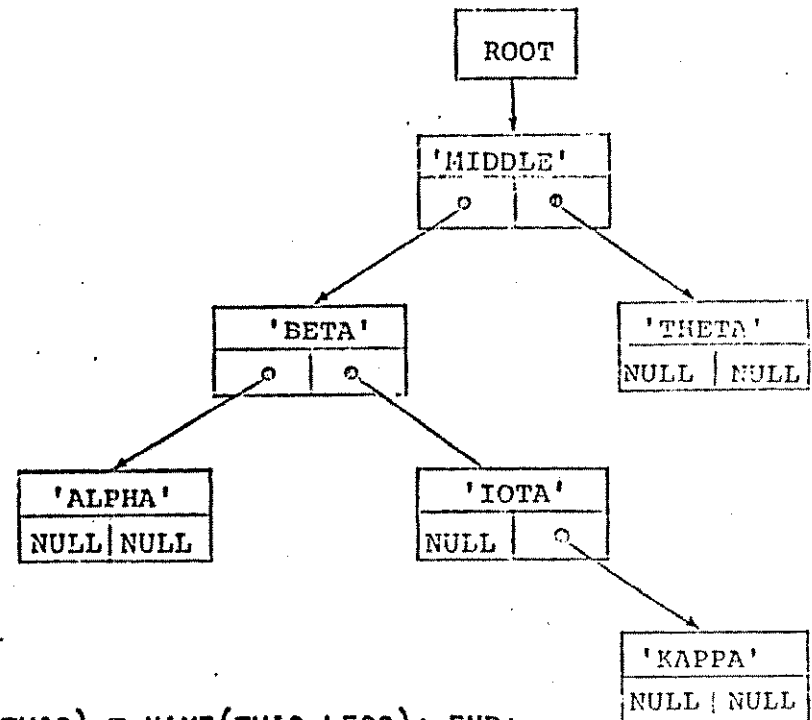
NAME(THIS) = NAME(ROOT);
DO WHILE NAME(THIS) ≠ NAME(NULL);
  NAME(OLD_LEAF) = NAME(THIS);
  IF NEW_SYMBOL < THIS.SYMBOL
    THEN DO; WAS_LESS = TRUE; NAME(THIS) = NAME(THIS.LESS); END;
    ELSE DO; WAS_LESS = FALSE; NAME(THIS) = NAME(THIS.GTR); END;
END;

```

```

I = I+1; SYMBOLI = NEW_SYMBOL;
NAME(LESSI), NAME(GTRI) = NAME(NULL);
IF NAME(ROOT) = NAME(NULL) THEN NAME(ROOT) = NAME(TREEI);
ELSE IF WAS_LESS THEN NAME(OLD_LEAF.LESS) = NAME(TREEI);
ELSE NAME(OLD_LEAF.GTR) = NAME(TREEI);

```

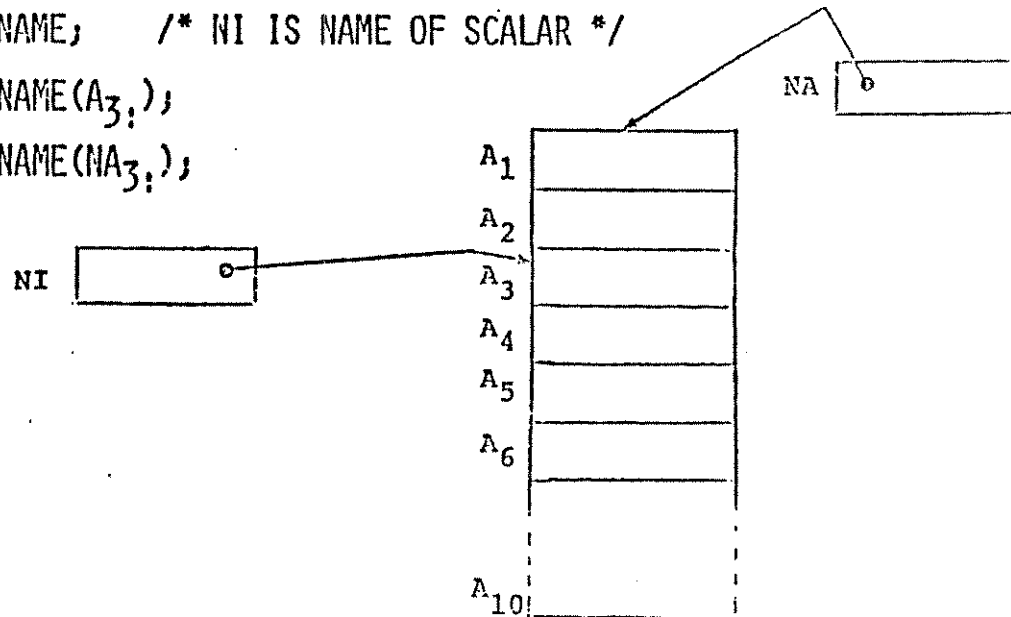


SUBSCRIPTING AND NAME VARIABLES

- SUBSCRIPTING OF DEREFERENCED NAME VARIABLES IS ALLOWED
E.G. DECLARE ARRAY(3) VECTOR, V, NV NAME INITIAL(NAME(V));
 $V_{1:3} = NV_{2:3};$ /* SAME AS $V_{1:3} = V_{2:3};$ */
- SUBSCRIPTING INSIDE A REFERENCE TO THE NAME PSEUDO-FUNCTION CAN ONLY APPEAR IN REFERENCE CONTEXT (NOT IN ASSIGNMENT CONTEXT).
SUBSCRIPTING APPLIES TO THE VARIABLE BEING POINTED TO.

E.G. DECLARE INTEGER

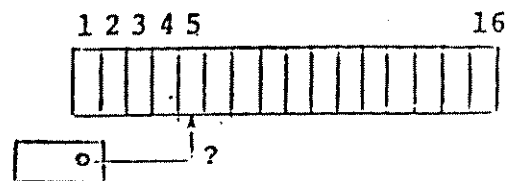
```
A ARRAY(10),  
NA NAME ARRAY(10) INITIAL(NAME(A)),  
NI NAME; /* NI IS NAME OF SCALAR */  
;  
NAME(NI) = NAME(A3);  
NAME(NI) = NAME(NA3);
```



COMPONENT SUBSCRIPTING INSIDE NAME()

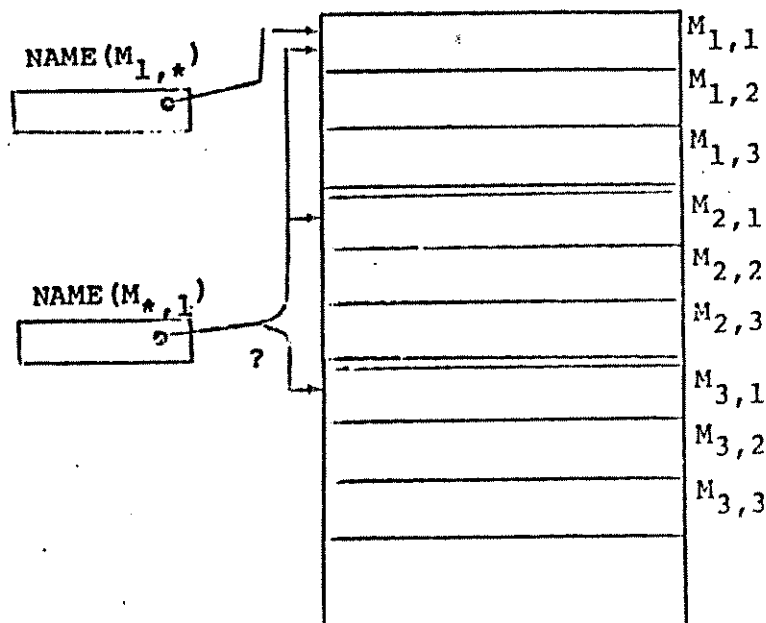
- ILLEGAL FOR BIT AND CHARACTER STRINGS

E.G. DECLARE B BIT(16); NAME(B₅)



- MUST SELECT A SINGLE SCALAR FROM VECTORS AND MATRICES (ALSO SINGLE ELEMENT FROM ARRAYS)

E.G. DECLARE M MATRIX;



STRUCTURE SUBSCRIPTING INSIDE NAME()

- IN ASSIGNMENT CONTEXT, OK ONLY IF NAME() IS APPLIED TO A NAME VARIABLE IN A STRUCTURE WITH MULTIPLE COPIES - THEN IT SELECTS THE APPROPRIATE COPY OF THE NAME VARIABLE ITSELF!

```
E,G.  STRUCTURE S: 1 N NAME SCALAR;  
      DECLARE S S-STRUCTURE(10);  
      NAME(N1) = NAME(N2);
```

- IN REFERENCE CONTEXT, ONLY ONE CAN
- NAME VARIABLE DEFINED IN A STRUCTURE WITH COPIES.
SUBSCRIPTING IS EFFECTIVE ON THE NAME VARIABLE ITSELF.
- NAME VARIABLE POINTING TO A STRUCTURE WITH COPIES.
SUBSCRIPTING IS EFFECTIVE ON THE VARIABLE BEING
POINTED TO.

```
STRUCTURE S: 1 N NAME-STRUCTURE(10); ILLEGAL!
```