```
DDDDDDDDDDD     EEEEEEEEEEEEEEE  BBBBBBBBBBB      UUU         UUU     GGGGGGGGGGG
DDDDDDDDDDDD    EEEEEEEEEEEEEEE  BBBBBBBBBBBB     UUU         UUU     GGGGGGGGGGGG
DDDDDDDDDDDD    EEEEEEEEEEEEEEE  BBBBBBBBBBBB     UUU         UUU     GGGGGGGGGGGG
DDD         DDD EEE              BBB         BBB  UUU         UUU GGG
DDD         DDD EEE              BBB         BBB  UUU         UUU GGG
DDD         DDD EEE              BBB         BBB  UUU         UUU GGG
DDD         DDD EEE              BBB         BBB  UUU         UUU GGG
DDD         DDD EEE              BBB         BBB  UUU         UUU GGG
DDD         DDD EEEEEEEEEEE      BBBBBBBBBBBB     UUU         UUU GGG
DDD         DDD EEEEEEEEEEE      BBBBBBBBBBBB     UUU         UUU GGG
DDD         DDD EEEEEEEEEEE      BBBBBBBBBBBB     UUU         UUU GGG
DDD         DDD EEE              BBB         BBB  UUU         UUU GGG   GGGGGGGGG
DDD         DDD EEE              BBB         BBB  UUU         UUU GGG   GGGGGGGGG
DDD         DDD EEE              BBB         BBB  UUU         UUU GGG   GGGGGGGGG
DDD         DDD EEE              BBB         BBB  UUU         UUU GGG         GGG
DDD         DDD EEE              BBB         BBB  UUU         UUU GGG         GGG
DDD         DDD EEE              BBB         BBB  UUU         UUU GGG         GGG
DDDDDDDDDDDD    EEEEEEEEEEEEEEE  BBBBBBBBBBBB      UUUUUUUUUUUUUUUUU     GGGGGGGGG
DDDDDDDDDDDD    EEEEEEEEEEEEEEE  BBBBBBBBBBBB      UUUUUUUUUUUUUUUUU     GGGGGGGGG
DDDDDDDDDDDD    EEEEEEEEEEEEEEE  BBBBBBBBBBBB      UUUUUUUUUUUUUUUUU     GGGGGGGGG
```

FILEID**DBGLIB

```
DDDDDDDD    BBBBBBBB      GGGGGGGG  LL            IIIIII    BBBBBBBB
DDDDDDDD    BBBBBBBB      GGGGGGGG  LL            IIIIII    BBBBBBBB
DD      DD  BB      BB  GG          LL              II      BB      BB
DD      DD  BB      BB  GG          LL              II      BB      BB
DD      DD  BB      BB  GG          LL              II      BB      BB
DD      DD  BB      BB  GG          LL              II      BB      BB
DD      DD  BBBBBBBB    GG          LL              II      BBBBBBBB
DD      DD  BBBBBBBB    GG          LL              II      BBBBBBBB
DD      DD  BB      BB  GG  GGGGGG  LL              II      BB      BB
DD      DD  BB      BB  GG  GGGGGG  LL              II      BB      BB
DD      DD  BB      BB  GG      GG  LL              II      BB      BB      ....
DD      DD  BB      BB  GG      GG  LL              II      BB      BB      ....
DDDDDDDD    BBBBBBBB      GGGGGG    LLLLLLLLLL    IIIIII    BBBBBBBB      ....
DDDDDDDD    BBBBBBBB      GGGGGG    LLLLLLLLLL    IIIIII    BBBBBBBB      ....
```

```
RRRRRRRR    EEEEEEEEEE    QQQQQQ
RRRRRRRR    EEEEEEEEEE    QQQQQQ
RR      RR  EE            QQ    QQ
RR      RR  EE            QQ    QQ
RR      RR  EE            QQ    QQ
RR      RR  EE            QQ    QQ
RRRRRRRR    EEEEEEEE      QQ    QQ
RRRRRRRR    EEEEEEEE      QQ    QQ
RR  RR      EE            QQ  QQ  QQ
RR  RR      EE            QQ  QQ  QQ
RR    RR    EE            QQ    QQ
RR    RR    EE            QQ    QQ
RR      RR  EEEEEEEEEE    QQQQ  QQ
RR      RR  EEEEEEEEEE    QQQQ  QQ
```

```
!----------------------------------------------------------------------
!
!       DBGLIB -- COMMON DEFINITION FILE FOR THE VAX DEBUGGER
!
!----------------------------------------------------------------------
!
! Version:      'V04-000'
!
!****************************************************************************
!*                                                                          *
!*  COPYRIGHT (c) 1978, 1980, 1982, 1984 BY                                 *
!*  DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS.                  *
!*  ALL RIGHTS RESERVED.                                                    *
!*                                                                          *
!*  THIS SOFTWARE IS FURNISHED UNDER A LICENSE AND MAY BE USED AND COPIED   *
!*  ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE          *
!*  INCLUSION OF THE ABOVE COPYRIGHT NOTICE. THIS SOFTWARE OR ANY OTHER     *
!*  COPIES THEREOF MAY NOT BE PROVIDED OR OTHERWISE MADE AVAILABLE TO ANY   *
!*  OTHER PERSON. NO TITLE TO AND OWNERSHIP OF THE SOFTWARE IS HEREBY       *
!*  TRANSFERRED.                                                            *
!*                                                                          *
!*  THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE    *
!*  AND SHOULD NOT BE CONSTRUED AS A COMMITMENT BY DIGITAL EQUIPMENT        *
!*  CORPORATION.                                                            *
!*                                                                          *
!*  DIGITAL ASSUMES NO RESPONSIBILITY FOR THE USE OR RELIABILITY OF ITS     *
!*  SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DIGITAL.                 *
!*                                                                          *
!*                                                                          *
!****************************************************************************
!
!
! WRITTEN BY
!       Bert Beander    June, 1980.
!
! MODIFIED BY
!       John Francis
!       Ping Sager
!       Rich Title
!       Vicki Holt
!       Sid Maxwell
!
! MODULE FUNCTION:
!       This REQUIRE file contains all literal, macro, and data structure defin-
!       itions used by new Bliss modules in the Debugger.  Old Bliss modules do
!       not use this file for historical reasons.
```

T A B L E O F C O N T E N T S

```
!                G E N E R A L   D E B U G G E R   D E F I N I T I O N S
!
!
!       The declarations in this section define symbols of general utility
!       throughout the Debugger.  This includes widely used literals and
!       various utility macros.
!
!
!***
!       DATA STRUCTURE DEFINITION AND ACCESS
!
!       Here we declare all the macros used to define and acccess BLISS data
!       structures.  These are the names L_, W_, B_, W0_, W1_, and so forth.
!       The actual definitions are in a separate REQUIRE file, but they are
!       required for the LIBRARY compilation of the present REQUIRE file.
!
LIBRARY 'LIB$:STRUCDEF.L32';
!
!**-
!
!       BLISS-32 LANGUAGE EXTENSIONS
!
!***
!       Here we declare various minor extensions to the Bliss-32 langauge.
!
LITERAL
        TRUE    = 1,                    ! Define TRUE and FALSE
        FALSE   = 0;                    !
!**-
!
MACRO
        REPEAT          = WHILE 1 DO%,  ! Infinite Loop
!
! ZEROCOR zeroes an area of memory.  The two parameters are ADDRESS and COUNT,
! which are the address of the area to be zeroed and the number of contiguous
! longwords to be zeroed, respectively.
!
        ZEROCOR (ADDRESS, COUNT) =
                CH$FILL (0, (COUNT) * 4, CH$PTR (ADDRESS))%,

        ONEOF (X) [] =
                (MASK (%REMAINING)) ^ (X) LSS 0%,

        OR_OP (X) [] =
                OR%,

        MASK (X) [] =
                1 ^ (31-X) OR_OP (%REMAINING) MASK (%REMAINING)%,
```

```
        CH$SEQUENCE (N) =
                VECTOR [CH$ALLOCATION (N)]%,

        CONSECUTIVE [X] = LITERAL X = %COUNT%;
```

```
!       ERROR REPORTING
!
!
!       The following macro is used for reporting internal DEBUG coding errors
!       symbolically.  The macro is used as follows:
!
!                       $DBG_ERROR('MODNAME\ROUTNAME number');
!
!       This signals the internal DEBUG error message DBG$_INTERR which prints
!       the text of the macro parameter.  The convention is that this parameter
!       should include the module name and the routine name of the signal loca-
!       tion so that the point where the error was signalled can be easily
!       determined by DEBUG developers.  In addition, a number may be included
!       in the signal text to make the text unique when there is more than one
!       $DBG_ERROR invocation in the same routine.

MACRO $DBG_ERROR (STRING) =
    SIGNAL (DBG$_INTERR, 1, UPLIT BYTE (%ASCIC STRING)
    %IF %LENGTH GTR 1
    %THEN
        , %REMAINING)
    %ELSE
        )
    %FI %;
```

```
!       DEBUG OUTPUT MACROS
!
!       The following MACROs are used in several places in the debugger
!       to do formatted ASCII output ($FAO).
!
MACRO

        ! $FAO_STG_COUNT makes a counted byte string out of an ASCII string.
        ! This macro is useful to transform an FAO control string into the
        ! address of such a string, whose first byte contains the length of
        ! the string in bytes.
        !
        $FAO_STG_COUNT (STRING) =
                %PLIT BYTE (%CHARCOUNT (STRING), %ASCII STRING)%,


        ! $FAO_TT_OUT constructs a call to FAO with a control string, and some
        ! arguments to the control string.  This formatted string is then
        ! output to the output device.
        !
        $FAO_TT_OUT (CTL_STRING) =
                DBG$FAO_OUT ($FAO_STG_COUNT (CTL_STRING)
                %IF %LENGTH GTR 1
                %THEN
                  , %REMAINING
                %FI
                )%;
```

```
!       MISCELLANEOUS LITERALS
!
!
!       This section defines literals which are used throughout the Debugger but
!       which do not belong with any specific data structures.
!
! These four literals are used to index into the DBG$GB_RADIX byte vector.
!
LITERAL
        DBG$B_RADIX_INPUT       = 0,
        DBG$B_RADIX_OUTPUT      = 1,
        DBG$B_RADIX_OUTPUT_OVER = 2;

! Ascii character literals
!
LITERAL
        DBG$K_LCBIAS            = 32,           ! Lower case bias
        DBG$K_TAB               = 9,            ! Horizontal tab
        DBG$K_SEMICOLON         = 59,           ! ';'
        DBG$K_CAR_RETURN        = 13,           ! <cr>
        DBG$K_LINE_FEED         = 10,           ! <lf>
        DBG$K_NULL              = 0,            ! Null character
        DBG$K_EX_POINT          = 33,           ! '!'
        DBG$K_BLANK             = 32,           ! ' '
        DBG$K_AMPERSAND         = 38,           ! '&'
        DBG$K_AT_SIGN           = 64,           ! '@'
        DBG$K_SLASH             = 47,           ! '/'
        DBG$K_BACKSLASH         = 92,           ! '\'
        DBG$K_EQUAL             = 61,           ! '='
        DBG$K_COMMA             = 44,           ! ','
        DBG$K_DOT               = 46,           ! '.'
        DBG$K_COLON             = 58,           ! ':'
        DBG$K_UPARROW           = 94,           ! '^'
        DBG$K_QUOTE             = 39,           ! '''
        DBG$K_LEFT_PARENTHESIS  = 40,           ! '('
        DBG$K_RIGHT_PARENTHESIS = 41,           ! ')'
        DBG$K_MULTIPLY          = 42,           ! '*'
        DBG$K_PLUS              = 43,           ! '+'
        DBG$K_MINUS             = 45,           ! '-'
        DBG$K_DIVIDE            = DBG$K_SLASH,  ! '/'
        DBG$K_DBLQUOTE          = 34,           ! '"'

! Define unary operators too

        DBG$K_UNARY_PLUS        = 230,
        DBG$K_UNARY_MINUS       = 231;


! Object type literals.  These values are used by the  ddress Expression
! Interpreter and the Command Language Interpreter to  dentity the objects
! returned by the Address Expression Interpreter.  Some of these values also
! appear in certain descriptors including the Primary, Value, and Address
! Expression descriptors.
!
LITERAL
```

```
        DBG$K_MIN_DESCR_TYPE    = 120,  ! Used in FROM clause of CASE
        DBG$K_LITERAL           = 120,  ! Literal value
        DBG$K_PRIMARY_DESC      = 121,  ! Primary descriptor
        DBG$K_VALUE_DESC        = 122,  ! Value descriptor
        DBG$K_PERM_DESC         = 123,  ! Permanent Symbol descriptor
        DBG$K_INSTRUCTION       = 124,  ! Instruction
        DBG$K_NC_INSTRUCTION    = 125,  ! Named constant, instruction type
        DBG$K_NC_OTHER          = 126,  ! Named constant, not instruction
        DBG$K_OTHER             = 127,  ! Language-specific data type
        DBG$K_NOTYPE            = 128,  ! Virtual address without any type
        DBG$K_EXTERNAL_DESC     = 129,  ! External (printable) format
        DBG$K_VAX_DESC          = 130,  ! VAX standard descriptor
        DBG$K_V_VALUE_DESC      = 131,  ! Volatile Value descriptor
        DBG$K_AED               = 132,  ! This code appears in the DBG$B_DHDR_TYPE
                                        ! field of the new primary descriptor
                                        ! and also in the corresponding place
                                        ! in the AED to tell that the descriptor
                                        ! is an AED and not a new-style descriptor.
        DBG$K_MAX_DESCR_TYPE    = 132;  ! Used in TO clause of CASE.

! Define the Radix Literals.
!
LITERAL
        DBG$K_DEFAULT   = 1,            ! Default source language radix
        DBG$K_BINARY    = 2,            ! Binary radix
        DBG$K_OCTAL     = 8,            ! Octal radix
        DBG$K_DECIMAL   = 10,           ! Decimal radix
        DBG$K_HEX       = 16;           ! Hexadecimal radix


! Define Pseudo-Symbol codes.
!
LITERAL
        DBG$K_CURRENT_LOC       = 220,  ! Current location "."
        DBG$K_PREDECESSOR       = 221,  ! Previous location "^"
        DBG$K_SUCCESSOR         = 222,  ! Next location <CR>
        DBG$K_LAST_VALUE        = 223,  ! Last value "\"
        DBG$K_INDIRECTION       = DBG$K_CURRENT_LOC;
                                        ! An alternative meaning of "."
                                        !       is indirection


! Token type literals. These are used by the Pathname Parser and scanners.
!
LITERAL
        DBG$K_TOK_NULL          = 0,    ! Null or EOL token
        DBG$K_TOK_INVAL         = 1,    ! Invalid token
        DBG$K_TOK_LINE          = 2,    ! '%LINE' token
        DBG$K_TOK_LABEL         = 3,    ! '%LABEL' token
        DBG$K_TOK_BS            = 4,    ! '\' token
        DBG$K_TOK_ID            = 5,    ! ID token
        DBG$K_TOK_INT           = 6,    ! integer token
        DBG$K_TOK_DOT           = 7,    ! '.' token
        DBG$K_TOK_REG           = 8,    ! '%REGISTER' token
        DBG$K_TOK_QNAME         = 9,    ! '%NAME' token
        DBG$K_TOK_LOWEST        =DBG$K_TOK_NULL,
```

```
        DBG$K_TOK_HIGHEST       =DBG$K_TOK_QNAME;

! Return state values for the Pathname Parser
!
LITERAL
        DBG$K_PN                = 0,    ! Data reference or lexical reference
        DBG$K_REG               = 1,    ! Register reference
        DBG$K_LINE              = 2,    ! '%LINE' reference
        DBG$K_LABEL             = 3;    ! '%LABEL' reference

! Command verb literals. These are used by the Command Line Interpreter.
!
LITERAL
        DBG$K_ALLOCATE_VERB     = 22,   ! ALLOCATE
        DBG$K_AT_SIGN_VERB      = 1,    ! @
        DBG$K_ATTACH_VERB       = 26,   ! ATTACH
        DBG$K_CALL_VERB         = 2,    ! CALL
        DBG$K_CANCEL_VERB       = 3,    ! CANCEL
        DBG$K_DECLARE_VERB      = 20,   ! DECLARE
        DBG$K_DEFINE_VERB       = 4,    ! DEFINE
        DBG$K_DELETE_VERB       = 32,   ! DELETE
        DBG$K_DEPOSIT_VERB      = 5,    ! DEPOSIT
        DBG$K_DISPLAY_VERB      = 28,   ! DISPLAY
        DBG$K_DUMP_VERB         = 27,   ! DUMP  (Only for developer use)
        DBG$K_EDIT_VERB         = 33,   ! EDIT
        DBG$K_EVALUATE_VERB     = 6,    ! EVALUATE
        DBG$K_EXAMINE_VERB      = 7,    ! EXAMINE
        DBG$K_EXIT_VERB         = 8,    ! EXIT
        DBG$K_EXITLOOP_VERB     = 19,   ! EXITLOOP
        DBG$K_FOR_VERB          = 25,   ! FOR
        DBG$K_GO_VERB           = 9,    ! GO
        DBG$K_HELP_VERB         = 13,   ! HELP
        DBG$K_IF_VERB           = 16,   ! IF
        DBG$K_REPEAT_VERB       = 18,   ! DO
        DBG$K_SAVE_VERB         = 31,   ! SAVE
        DBG$K_SCROLL_VERB       = 29,   ! SCROLL
        DBG$K_SEARCH_VERB       = 15,   ! SEARCH
        DBG$K_SELECT_VERB       = 30,   ! SELECT
        DBG$K_SET_VERB          = 10,   ! SET
        DBG$K_SHOW_VERB         = 11,   ! SHOW
        DBG$K_SPAWN_VERB        = 21,   ! SPAWN
        DBG$K_STEP_VERB         = 12,   ! STEP
        DBG$K_SYMBOLIZE_VERB    = 23,   ! SYMBOLIZE
        DBG$K_TYPE_VERB         = 14,   ! TYPE
        DBG$K_UNDEFINE_VERB     = 24,   ! UNDEFINE
        DBG$K_WHILE_VERB        = 17,   ! WHILE

        DBG$K_FIRST_VERB        = DBG$K_AT_SIGN_VERB,
        DBG$K_LAST_VERB         = DBG$K_EDIT_VERB,

        EVENT$K_SET_BREAK       =  1,   ! Also SET_BREAK in DBGNSET
        EVENT$K_SET_BREAK_EXC   =  3,   ! Also SET_EXCEPTION_BREAK in DBGNSET
        EVENT$K_SET_TRACE       = 14,   ! Also SET_TRACE in DBGNSET
        EVENT$K_SET_WATCH       = 17,   ! Also SET_WATCH in DBGNSET
```

```
        EVENT$K_SET_STEP        = 11,   ! Also SET_STEP in DBGNSET
        EVENT$K_STEP            = 12,   ! Also DBG$K_STEP_VERB

        EVENT$K_SHOW_BREAK      = 1,    ! Also SHOW_BREAK in DBGNSHOW
        EVENT$K_SHOW_TRACE      = 11,   ! Also SHOW_TRACE in DBGNSHOW
        EVENT$K_SHOW_WATCH      = 14,   ! Also SHOW_WATCH in DBGNSHOW

        EVENT$K_CANCEL_BREAK    = 2,    ! Also CANCEL_BREAK in DBGNCANCL
        EVENT$K_CANCEL_BREAK_EXC= 4,    ! Also CANCEL_EXCEPTION_BREAK in DBGNCANCL
        EVENT$K_CANCEL_TRACE    = 9,    ! Also CANCEL_TRACE in DBGNCANCL
        EVENT$K_CANCEL_WATCH    = 14;   ! Also CANCEL_WATCH in DBGNCANCL


! Value kind literals.  These values are returned as "value kinds" by routines
! DBG$STA_SYMVALUE and DBG$STA_VALSPEC.
!
LITERAL
        DBG$K_VAL_NOVALUE       = 0,    ! Symbol is a type and has no value
        DBG$K_VAL_LITERAL       = 1,    ! Value is a literal
        DBG$K_VAL_ADDR          = 2,    ! Value is an address
        DBG$K_VAL_DESCR         = 3,    ! Value is a descriptor pointer
        DBG$K_VAL_UNALLOC       = 4;    ! Symbol was never allocated and
                                        !      hence has no value


! Debug specific DTYPE codes which are used in DBGNEXMNE.
! These codes are TEMPORARY!
!
LITERAL
        DBG$K_DTYPE_AD          = 56;   ! ASCID string

! Maximum number of digits in a packed decimal number.
!
LITERAL
        DBG$K_LARGEST_PACKED    = 31;


! PL/I specific data type codes.  These are referenced in DBGPERMOP and in
! DBGEVALOP in calls to the PL/I RTL during type conversions and calculations.
!
LITERAL
        DBG$K_PLI_PIC           = 1,
        DBG$K_PLI_FIX_BIN       = 2,
        DBG$K_PLI_FLO_BIN       = 3,
        DBG$K_PLI_FIX_DEC       = 4,
        DBG$K_PLI_FLO_DEC       = 5,
        DBG$K_PLI_CHAR          = 10,
        DBG$K_PLI_CHAR_VAR      = 11,
        DBG$K_PLI_UBIT          = 12,
        DBG$K_PLI_ABIT          = 14;


! Debug Print Control Code.
!
LITERAL
```

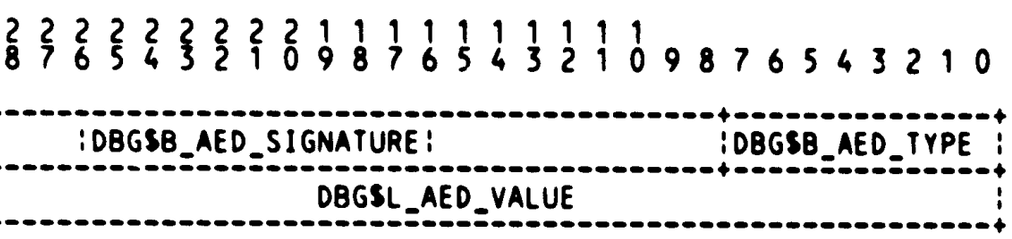
```
        DBG$K_PRTSET_LMARGIN    = 1,    ! Set left margin for DBG$PRINT
        DBG$K_PRTSET_RLMARGIN   = 2,    ! Set relative left margin
        DBG$K_PRTBRK_ON_BLANKS  = 3,    ! Set break on blanks flag
        DBG$K_PRTSET_CONTINUE   = 4,    ! Set indentation length for continuation
        DBG$K_PRT_RESET         = 5;    ! Reset to default settings
!***-
```

```
! Input string descriptor definition. This definition is compatable with the
! use of the DSC$B_DTYPE, DSC$B_CLASS, DSC$W_LENGTH, and DSC$A_POINTER macros.
!
LITERAL
        DBG$K_STG_DESC_SIZE = 12;       ! Length of descriptor in bytes

MACRO
        DBG$STG_DESC = BLOCK [DBG$K_STG_DESC_SIZE, BYTE] %;
```

```
!                A D D R E S S   D E S C R I P T O R S


!       The Address Descriptor is used to define an address where a bit offset
!       is needed in addition to a byte address.

!        3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
!        1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
!       +---------------------------------------------------------------+
!   0   !                     DBG$L_ADDRESS_BYTE_ADDR                   !
!       +---------------------------------------------------------------+
!   1   !                     DBG$L_ADDRESS_BIT_OFFSET                  !
!       +---------------------------------------------------------------+

!       A pointer to an Address Descriptor is declared as follows:

!                       ADDPTR: REF DBG$ADDRESS_DESC;

! Define the fields in the Address Descriptor.  Also define the declaration
! macro.
!***
FIELD DBG$ADDRESS_DESC_FIELDS =
        SET
        DBG$L_ADDRESS_BYTE_ADDR  = [ 0, L_ ],   ! Byte address
        DBG$L_ADDRESS_BIT_OFFSET = [ 1, L_ ]    ! Bit offset
        TES;

LITERAL
        DBG$K_ADDRESS_DESC_SIZE = 2;            ! Size of descriptor in longwords

MACRO
        DBG$ADDRESS_DESC = BLOCK[DBG$K_ADDRESS_DESC_SIZE]
                           FIELD(DBG$ADDRESS_DESC_FIELDS) %;
!***
```

A D D R E S S E X P R E S S I O N D E S C R I P T O R S

The Address Expression Descriptor is used to define an address expression. It has the following structure:



A pointer to an Address Expression Descriptor is declared as follows:

```
AEDPTR: REF DBG$AED
```

Define the fields in the Address Expression Descriptor. Also define the declaration macro.

```
FIELD DBG$AED_FIELDS =
        SET
        DBG$B_AED_TYPE  = [ 0, B0_ ],   ! The type of the value field
        DBG$B_AED_SIGNATURE = [0, B2_], ! Always contains a code DBG$K_AED
                                        !    saying this is an AED.
        DBG$L_AED_VALUE = [ 1, L_ ]     ! L-value or pointer to descriptor
        TES;

LITERAL
        DBG$K_AED_SIZE  = 2;            ! Size of descriptor in longwords

MACRO
        DBG$AED = BLOCK[DBG$K_AED_SIZE] FIELD(DBG$AED_FIELDS) %;
```

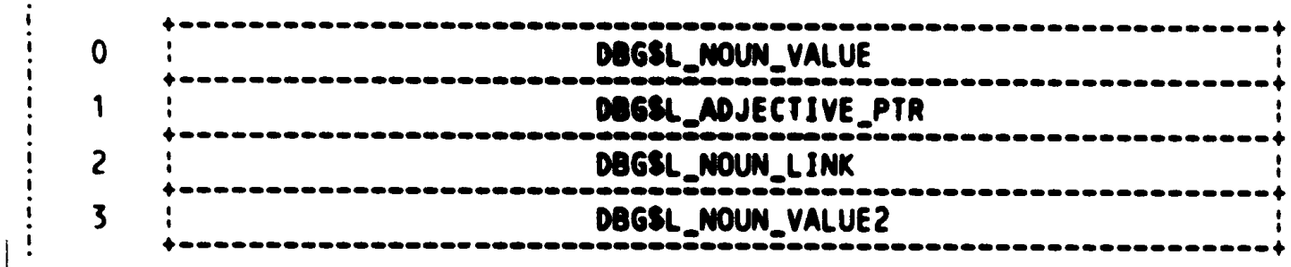
The following are the legal values for the DBG$B_AED_TYPE field.

DBG$K_PRIMARY_DESC	The value field contains the address of a primary descriptor
DBG$K_PERM_DESC	The value field contains the address of a permanent symbol descriptor
DBG$K_INSTRUCTION	The value field contains a PC value
DBG$K_NOTYPE	The value field contains an untyped L-value

```
!                C O M M A N D   E X E C U T I O N   T R E E S
!
! This section contains the definitions for the nodes that are part of a
! command execution tree. A command execution tree is built when a debug
! command is parsed. The tree contains a verb node representing the command
! (e.g., EXAMINE, DEPOSIT, etc.) as the header node. Linked to the verb node
! are chains of adverb nodes and noun nodes. The noun node chain hangs off
! the OBJECT_PTR field and the adverb node chain hangs off the ADVERB_PTR
! field. The exact structure of the tree depends on the command.
!
! VERB NODES
!
!       +-----------------------------------------------+---------------+
!   0   !                                 !$B_VERB_COMPOSITE!$B_VERB_LITERAL!
!       +-----------------------------------------------+---------------+
!   1   !                           DBG$L_VERB_ADVERB_PTR                 !
!       +-------------------------------------------------------------- -+
!   2   !                           DBG$L_VERB_OBJECT_PTR                 !
!       +-------------------------------------------------------------- -+

FIELD
    DBG$VERB_NODE_FIELDS =
        SET

        DBG$B_VERB_LITERAL      = [0, 0, 8, 0],         ! First level verb
        DBG$B_VERB_COMPOSITE    = [0, 8, 8, 0],         ! Second or third level verb
        DBG$L_VERB_ADVERB_PTR   = [1, 0, 32, 0],        ! Pointer to adverb node
        DBG$L_VERB_OBJECT_PTR   = [2, 0, 32, 0]         ! Pointer to object (noun node)

        TES;

LITERAL
        DBG$K_VERB_NODE_SIZE    = 3;                    ! Length in longwords

MACRO
        DBG$VERB_NODE = BLOCK [DBG$K_VERB_NODE_SIZE] FIELD (DBG$VERB_NODE_FIELDS) %;

! ADVERB NODES
!
!       +-----------------------------------------------+---------------+
!   0   !                                               !_ADVERB_LITERAL!
!       +-----------------------------------------------+---------------+
!   1   !                           DBG$L_ADVERB_VALUE                    !
!       +-------------------------------------------------------------- -+
!   2   !                           DBG$L_ADVERB_LINK                     !
!       +-------------------------------------------------------------- -+

FIELD
    DBG$ADVERB_NODE_FIELDS =
        SET

        DBG$B_ADVERB_LITERAL    = [0, 0, 8, 0],         ! Adverb id literal
        DBG$L_ADVERB_VALUE      = [1, 0, 32, 0],        ! Value (integer)
        DBG$L_ADVERB_LINK       = [2, 0, 32, 0]         ! Link to next adverb node
```

```
        TES;

LITERAL
        DBG$K_ADVERB_NODE_SIZE  = 3;                    ! Length in longwords

MACRO
        DBG$ADVERB_NODE = BLOCK [DBG$K_ADVERB_NODE_SIZE] FIELD (DBG$ADVERB_NODE_FIELDS) %;
```

! NOUN NODES

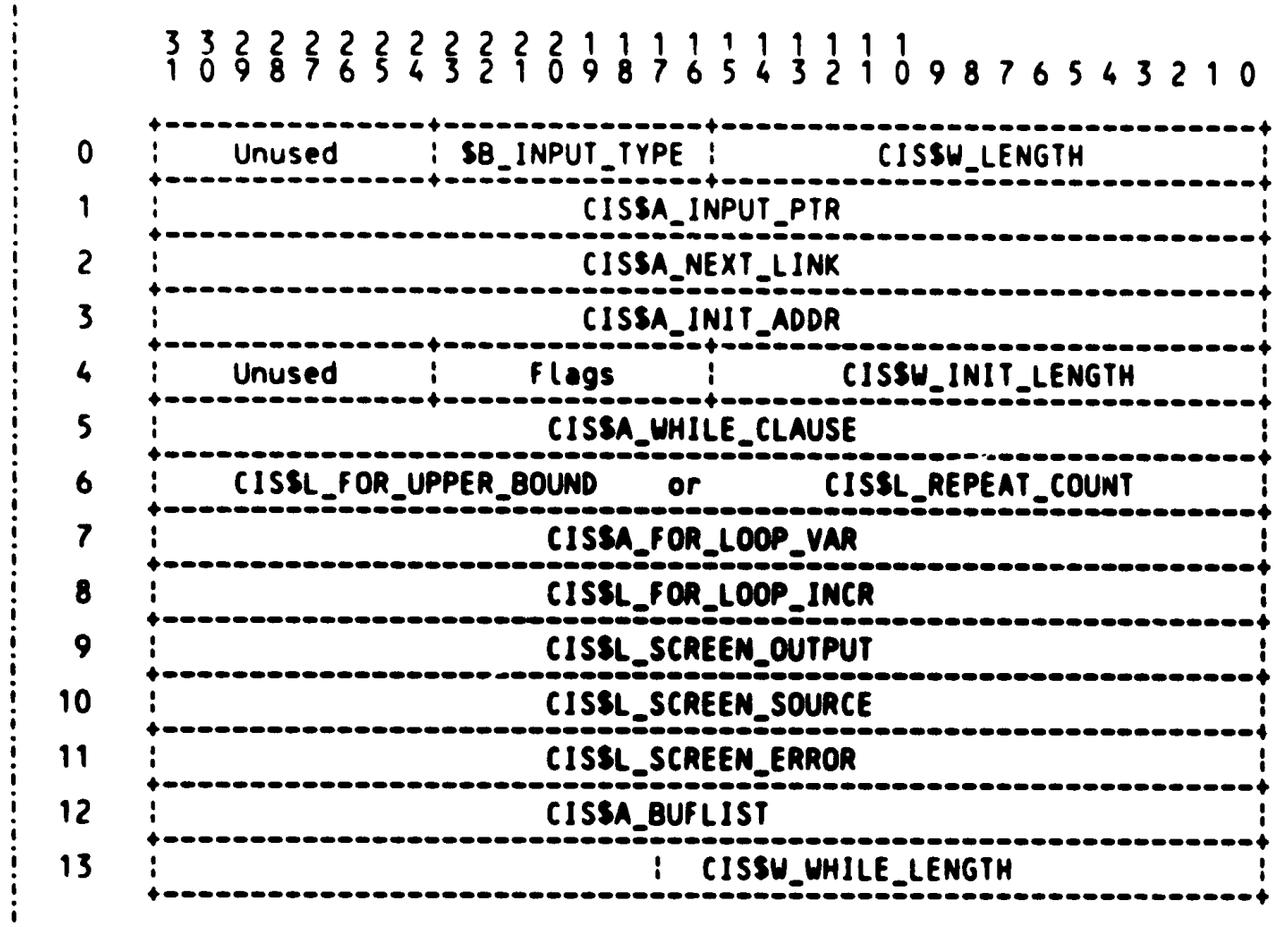



```
FIELD DBG$NOUN_NODE_FIELDS =
        SET
        DBG$L_NOUN_VALUE     = [ 0, L_ ],! Noun value
        DBG$L_ADJECTIVE_PTR  = [ 1, L_ ],! Pointer to adjective (unused)
        DBG$L_NOUN_LINK      = [ 2, L_ ],! Pointer to next noun
        DBG$L_NOUN_VALUE2    = [ 3, L_ ],! Additional Noun value
        DBG$L_NOUN_VALUE3    = [ 4, L_ ],! More Noun value (long nodes only)
        DBG$L_NOUN_VALUE4    = [ 5, L_ ] ! More Noun value (long nodes only)
        TES;

LITERAL
        DBG$K_NOUN_NODE_SIZE      = 4,  ! Length of normal Noun Node in longwords
        DBG$K_NOUN_NODE_SIZE_LONG = 6;  ! Length of long Noun Node in longwords

MACRO
    DBG$NOUN_NODE = BLOCK [DBG$K_NOUN_NODE_SIZE] FIELD (DBG$NOUN_NODE_FIELDS) %;
```

C O M M A N D I N P U T S T R E A M

The Command Input Stream (CIS) is a linked list that the debugger maintains to describe where it is currently taking input from.

The format of a Command Input Stream Descriptor is:



A pointer to a command input stream descriptor block is declared with:

```
CIS_PTR: REF CIS$LINK
```

Define the fields, the descriptor size, and the declaration macro.

```
FIELD CIS_FIELDS =
        SET
        CIS$B_INPUT_TYPE  = [ 2, B_ ],  ! Type of CIS link
        CIS$W_LENGTH      = [ 0, W_ ],  ! Length of input buffer
        CIS$A_INPUT_PTR   = [ 4, L_ ],  ! Pointer to input buffer
```

```
        CIS$A_NEXT_LINK    = [ 8, L_ ],  ! Ptr to next cis link
        CIS$A_INIT_ADDR    = [ 12, L_ ], ! Ptr to start of input buffer
        CIS$W_INIT_LENGTH  = [ 16, W_ ], ! Allocated buffer size
        CIS$V_REM_FLAG     = [ 18, V_(0)],! Is link flagged for removal?
        CIS$V_WHILE_FLAG   = [ 18, V_(1)],! TRUE when condition is true
        CIS$V_SCREEN_NOGO  = [ 18, V_(2)],! Saved NOGO flag to turn off STEPs
                                          !     and GOs in CIS_SCREEN entries
        CIS$A_WHILE_CLAUSE =[ 20, L_ ], ! Pointer to WHILE command
        CIS$L_REPEAT_COUNT =[ 24, L_ ], ! Count for REPEAT loops
        CIS$L_FOR_UPPER_BOUND=[24, L_ ],! Upper bound for FOR loops
        CIS$A_FOR_LOOP_VAR =[ 28, L_ ], ! Pointer to the FOR loop var name
        CIS$L_FOR_LOOP_INCR=[ 32, L_],  ! Increment for FOR loops
        CIS$L_SCREEN_OUTPUT=[ 36, L_ ], ! Reset value for DBG$GL_SCREEN_OUTPUT
        CIS$L_SCREEN_SOURCE=[ 40, L_ ], ! Reset value for DBG$GL_SCREEN_SOURCE
        CIS$L_SCREEN_ERROR =[ 44, L_ ], ! Reset value for DBG$GL_SCREEN_ERROR
        CIS$A_BUFLIST      =[ 48, L_ ], ! List of buffers to free in CIS_REMOVE
        CIS$W_WHILE_LENGTH =[ 52, W_ ]
        TES;

LITERAL
        CIS_ELEMENTS    = 54;           ! CIS block size (bytes)

MACRO
        CIS$LINK = BLOCK [CIS_ELEMENTS, BYTE] FIELD(CIS_FIELDS) %;

! The following are legal types for CIS links.
!
LITERAL
        DBG$K_CIS_MINIMUM       =  0,           ! Lowest type CIS
        DBG$K_CIS_DBG$INPUT     =  0,           ! Type DBG$INPUT
        DBG$K_CIS_RAB           =  1,           ! Type RAB
        DBG$K_CIS_INPBUF        =  2,           ! Type input buffer
        DBG$K_CIS_ACBUF         =  3,           ! Type action buffer
        DBG$K_CIS_REPEAT        =  4,           ! Type repeat buffer
        DBG$K_CIS_WHILE         =  5,           ! Type while buffer
        DBG$K_CIS_IF            =  6,           ! Type if buffer
        DBG$K_CIS_FOR           =  7,           ! Type for buffer
        DBG$K_CIS_SCREEN        =  8,           ! Type for Screen Display
        DBG$K_CIS_MAXIMUM       =  8;           ! Highest type CIS

! The definitions below should eventually be eliminated when all uses of
! them are converted to the DBG$K_ form.
LITERAL
        CIS_DBG$INPUT   =  0,           ! Type DBG$INPUT
        CIS_RAB         =  1,           ! Type RAB
        CIS_INPBUF      =  2,           ! Type input buffer
        CIS_ACBUF       =  3,           ! Type action buffer
        CIS_REPEAT      =  4,           ! Type repeat buffer
        CIS_WHILE       =  5,           ! Type while buffer
        CIS_IF          =  6,           ! Type if buffer
        CIS_FOR         =  7,           ! Type for buffer
        CIS_SCREEN      =  8;           ! Type for Screen Display
```

```
!               C O N T R O L   F L A G S
!
!       Control flags are global DEBUG status flags used to control the
!       overall flow of DEBUG execution. They represent the major state
!       of DEBUG.

MACRO DBG$CONTROL_FLAGS = BITVECTOR[]%;

LITERAL
        DBG$V_CONTROL_TDBG      =  0,   ! Set if this is a testable DEBUG
        DBG$V_CONTROL_SDBG      =  1,   ! Set if this is SUPERDEBUG
        DBG$V_CONTROL_KDBG      =  2,   ! Reserved for future development
        DBG$V_CONTROL_URUN      =  3,   ! Set if user program has been run to
                                        !     stop DEBUG allocating memory
        DBG$V_CONTROL_EXIT      =  4,   ! Set if DEBUG is about to EXIT
        DBG$V_CONTROL_FAIL      =  5,   ! Set by DEBUG internal errors
        DBG$V_CONTROL_DONE      =  6,   ! Set if user execution complete
        DBG$V_CONTROL_ALLOCATE  =  7,   ! Set if OK to allocate more memory
                                        !     (e.g., SET MODULE/ALLOCATE)
        DBG$V_CONTROL_USER      =  8,   ! Set if user program is running
        DBG$V_CONTROL_STOP      =  9,   ! Set by ^Y,DEBUG sequence to abort
                                        !     processing DEBUG command files
        DBG$V_CONTROL_TBIT      = 10,   ! Used by DBGEVENT and DBGSTART to
                                        !     control AST/TBIT interactions.
        DBG$V_CONTROL_SCREEN    = 11,   ! Set if screen displays must be updated
                                        !     because user program has run
        DBG$V_CONTROL_VERSION_4 = 12;   ! Set if VMS 3B or 4.0 is running


! We also define a macro $ABORT_ON_CONTROL_Y. The purpose of this macro is
! to allow premature termination of commands which can take a large amount
! of time to complete but to allow the individual command routines to make
! sure that they leave all DEBUGs internal data structures in a consistent
! state. Execution of this macro will have no effect if the Control-Y flag
! is clear, but will SIGNAL back to DEBUG command level if the flag is set

MACRO $ABORT_ON_CONTROL_Y =
        BEGIN
        EXTERNAL DBG$GV_CONTROL : BITVECTOR[];

        IF .DBG$GV_CONTROL[DBG$V_CONTROL_STOP] THEN SIGNAL(DBG$_ABORTED);
        END%;
```

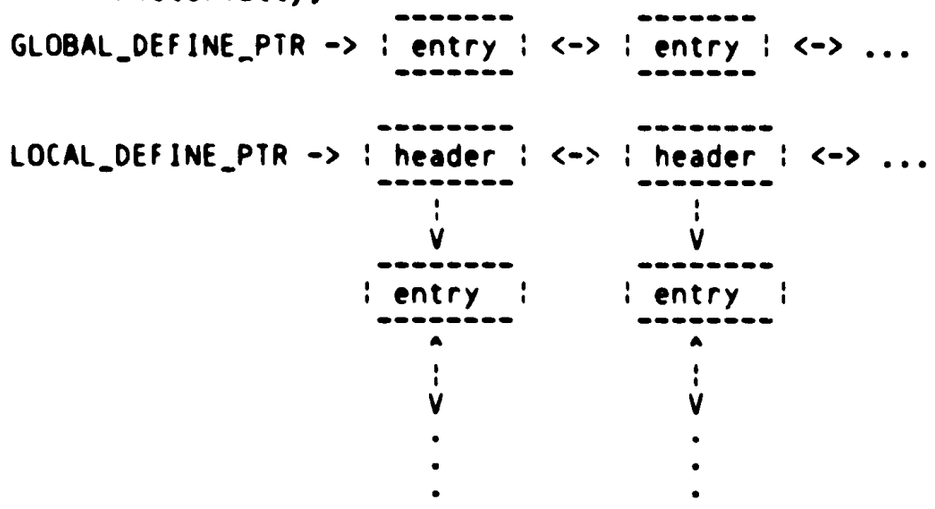
D E F I N E S Y M B O L T A B L E

The Define Symbol Table contains information about the symbols that are currently defined with the DEFINE command.

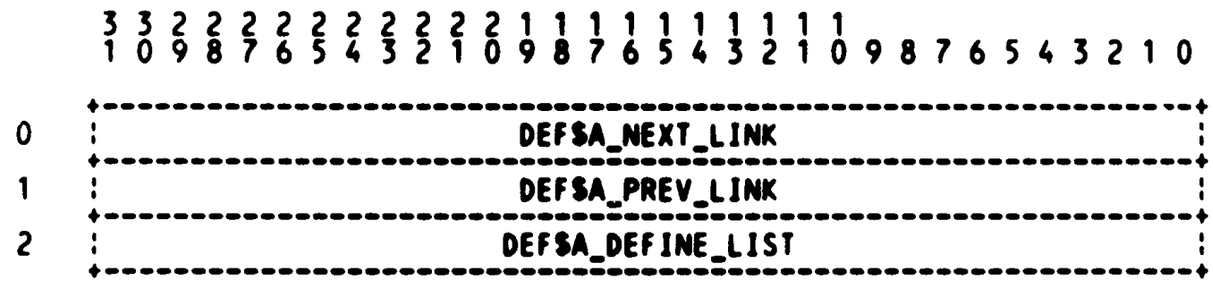
There is a doubly linked list for all globally-defined symbols. (I.e., symbols defined with DEFINE <symbol> == <definition>) The global variable DBG$GL_GLOBAL_DEFINE_PTR points to this list.

For each command procedure that is active, there may also be a define list that is local to the procedure. The variable DBG$GL_LOCAL_DEFINE_PTR points to a doubly-linked list of header blocks. Each header block points to a doubly-linked list of symbols.

Pictorially,



The format of a header block is:



The format of an entry is:

```
!        1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
!      +---------------------------------------------------------------+
!   0  !                    DEF$A_ENTRY_NEXT_LINK                      !
!      +---------------------------------------------------------------+
!   1  !                    DEF$A_ENTRY_PREV_LINK                      !
!      +---------------------------------------------------------------+
!   2  !                         DEF$A_NAME                            !
!      +---------------------------------------------------------------+
!   3  !                         DEF$A_VALUE                           !
!      +-------------------------------------------+-------------------+
!   4  !                    Unused                 !    DEF$B_KIND     !
!      +-------------------------------------------+-------------------+


! Define the fields of a DEFINE entry.

FIELD DEFINE_ENTRY_FIELDS =
        SET
        DEF$A_ENTRY_NEXT_LINK = [ 0, L_ ],      ! Forward link
        DEF$A_ENTRY_PREV_LINK = [ 4, L_ ],      ! Backward link
        DEF$A_NAME            = [ 8, L_ ],      ! Points to a counted string
                                                !    with the name being defined
        DEF$A_VALUE           = [ 12, L_ ],     ! Points to a descriptor or
                                                !     counted string with the
                                                !     "value" of the symbol.
        DEF$B_KIND            = [ 16, B0_ ]     ! Contains a code for the "kind"
                                                !    of definition. The codes
                                                !    are defined below.
        TES;

! Define the fields of a header block.
!
FIELD DEFINE_HEADER_FIELDS =
        SET
        DEF$A_NEXT_LINK       = [ 0, L_ ],      ! Forward link
        DEF$A_PREV_LINK       = [ 4, L_ ],      ! Backward link
        DEF$A_DEFINE_LIST     = [ 8, L_ ]       ! Points to the linked list
                                                !      of DEFINE entries.
        TES;

! Define the size in bytes and in words of the above structures.
!
LITERAL
    DBG$K_DEFINE_ENTRY_SIZE_B  = 17,
    DBG$K_DEFINE_ENTRY_SIZE_W = 5,
    DBG$K_DEFINE_HEADER_SIZE_B = 12,
    DBG$K_DEFINE_HEADER_SIZE_W = 3;

! Define the codes that will be used for the possible kinds of
! symbol definitions.
```

```
!
LITERAL
    DEFINE_LOWEST       = 1,    ! Lowest value of DEFINE codes
    DEFINE_ADDRESS      = 1,
    DEFINE_COMMAND      = 2,
    DEFINE_PROCEDURE    = 3,
    DEFINE_STRING       = 4,
    DEFINE_VALUE        = 5,
    DEFINE_PARAMETER    = 6,    ! This is used to temporarily store strings
                                !      representing actual parameters, until
                                !      they are bound to formals by the
                                !      DECLARE command.
    DEFINE_KEY          = 7,
    DEFINE_HIGHEST      = 7;    ! Highest value of DEFINE codes

! Macros for declaration of DEFINE entries and headers.
!
MACRO
    DEFINE$ENTRY =
        BLOCK [DBG$K_DEFINE_ENTRY_SIZE_B, BYTE] FIELD (DEFINE_ENTRY_FIELDS)%,
    DEFINE$HEADER =
        BLOCK [DBG$K_DEFINE_HEADER_SIZE_B, BYTE] FIELD (DEFINE_HEADER_FIELDS)%;
```

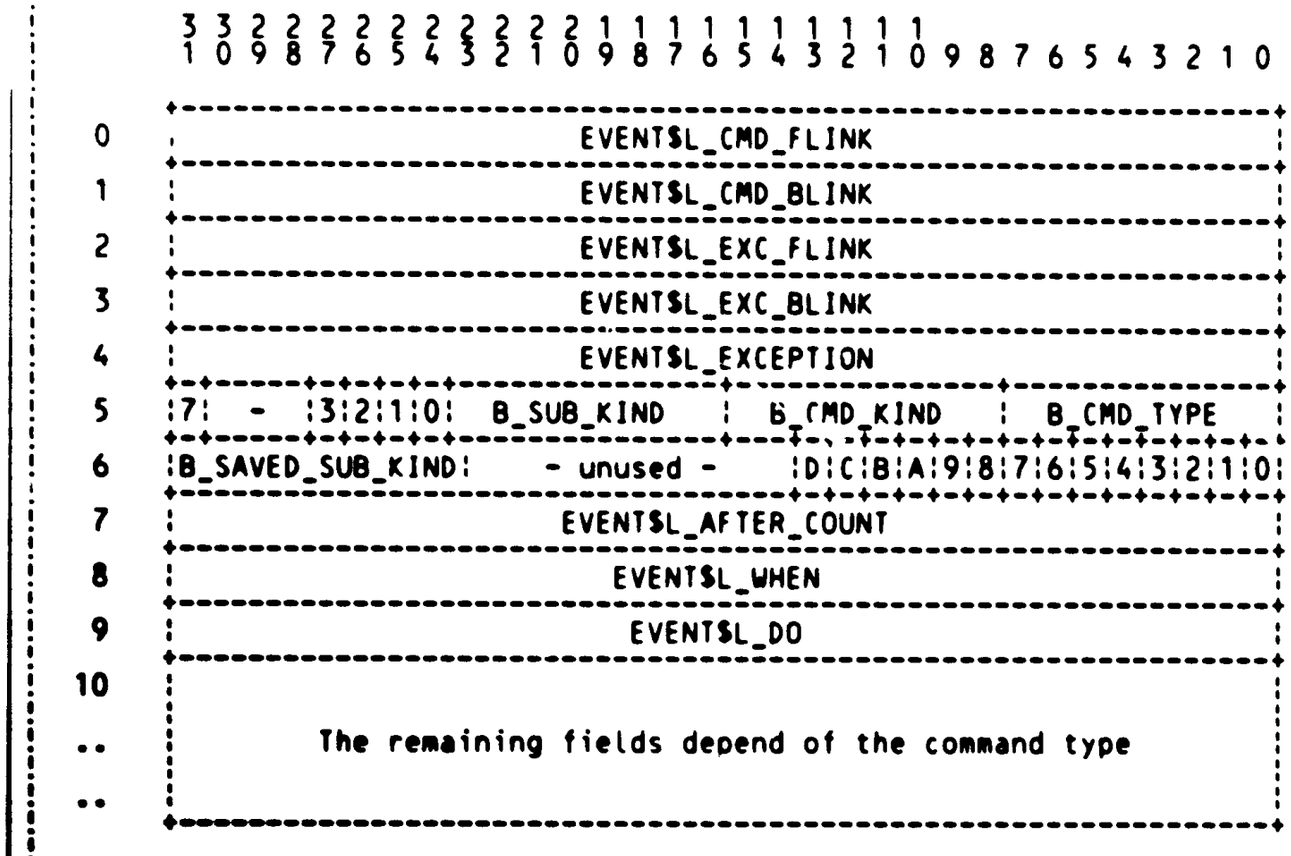
E R R O R S T A T U S C O D E S

Debugger status codes are defined in the MDL file DBGMES.MDL--new ones should therefore be entered in that file. All that is found here are definitions relating to components of status codes.

```
! Define the possible values of the status severity field.

LITERAL
        SYS$K_INFO      = 3;            ! Informational status severity
```

E V E N T T A B L E E N T R Y

The Event Table contains all needed information about all set breakpoints, tracepoints, and watchpoints. For each set eventpoint, there is one Event Table Entry. The exact format of the Event Table Entry depends on the nature of the eventpoint itself, but all Event Table Entries have the following general format:




```
0 :     [5] EVENT$V_SILENT
1 :     [5] EVENT$V_THREADED
2 :     [5] EVENT$V_ONCE_ONLY
3 :     [5] EVENT$V_HAS_VAL_DSCR
4 :     [5] EVENT$V_ROUTINE
5 :     [5] EVENT$V_RET_AT_PC

7 :     [5] EVENT$V_DELETED

0 :     [6] EVENT$V_OVRD_LINE
1 :     [6] EVENT$V_STEP_LINE
2 :     [6] EVENT$V_OVRD_OVER
3 :     [6] EVENT$V_STEP_OVER
```

```
                        4 :     [6] EVENT$V_OVRD_SOURCE
                        5 :     [6] EVENT$V_STEP_SOURCE
                        6 :     [6] EVENT$V_OVRD_NOSYSTEM
                        7 :     [6] EVENT$V_STEP_NOSYSTEM
                        8 :     [6] EVENT$V_OVRD_NOSILENT
                        9 :     [6] EVENT$V_STEP_NOSILENT

                        A :     [6] EVENT$V_STEP_BPT
                        B :     [6] EVENT$V_STEP_TICK
                        C :     [6] EVENT$V_STEP_LINE_OK

                        D :     [6] EVENT$V_BREAK_ADDRESS

        A pointer to an Event Table Entry is declared as follows:

                        EVENTPTR: REF EVENT$EVENT_DESCRIPTOR


! Define the fields of the Event Table Entry.  Also declare the maximum
! size of an Event Table Entry and define the declaration macro.

FIELD EVENT$EVENT_DESCRIPTOR_FIELDS =
        SET
        EVENT$A_EVENT_DESCRIPTOR = [0,A_],  ! Event Descriptor address

        EVENT$L_CMD_FLINK    =   [0,L_],    ! Command queue foward link
        EVENT$L_CMD_BLINK    =   [1,L_],    ! Command queue backward link

        EVENT$L_EXC_FLINK    =   [2,L_],    ! Exception queue forward link
        EVENT$L_EXC_BLINK    =   [3,L_],    ! Exception queue backward link

        EVENT$L_EXCEPTION    =   [4,L_],    ! Exception code we expect to see

        EVENT$B_CMD_TYPE     =   [5,B0_],   ! Command type
        EVENT$B_CMD_KIND     =   [5,B1_],   ! Command kind
        EVENT$B_SUB_KIND     =   [5,B2_],   ! Command sub-kind

        EVENT$B_CMD_FLAGS    =   [5,B3_],   ! Command flags

        EVENT$V_SILENT       =   [5,V3_(0)], ! Flag set if /SILENT qualifier set
        EVENT$V_THREADED     =   [5,V3_(1)], ! Flag set if this is threaded code
        EVENT$V_ONCE_ONLY    =   [5,V3_(2)], ! Flag set for temporary event entry
        EVENT$V_HAS_VAL_DSCR=    [5,V3_(3)], ! Flag set if there is an Event
                                             !     Value Descriptor
        EVENT$V_ROUTINE      =   [5,V3_(4)], ! Flag set if ROUTINE EventPoint
        EVENT$V_RET_AT_PC    =   [5,V3_(5)], ! Flag set if SET BREAK/RET (or TRACE)
                                             !     address is at current PC
        EVENT$V_DELETED      =   [5,V3_(7)], ! Flag set if this entry is deleted

        EVENT$L_STEP_FLAGS   =   [6,L_],     ! 'Stepping' flags

        EVENT$V_OVRD_LINE    =   [6,V_( 0)], ! Flag set if override step/line
        EVENT$V_STEP_LINE    =   [6,V_( 1)], ! Flag set if step/line
        EVENT$V_OVRD_OVER    =   [6,V_( 2)], ! Flag set if override step/over
        EVENT$V_STEP_OVER    =   [6,V_( 3)], ! Flag set if step/over
        EVENT$V_OVRD_SOURCE  =   [6,V_( 4)], ! Flag set if override step/source
```

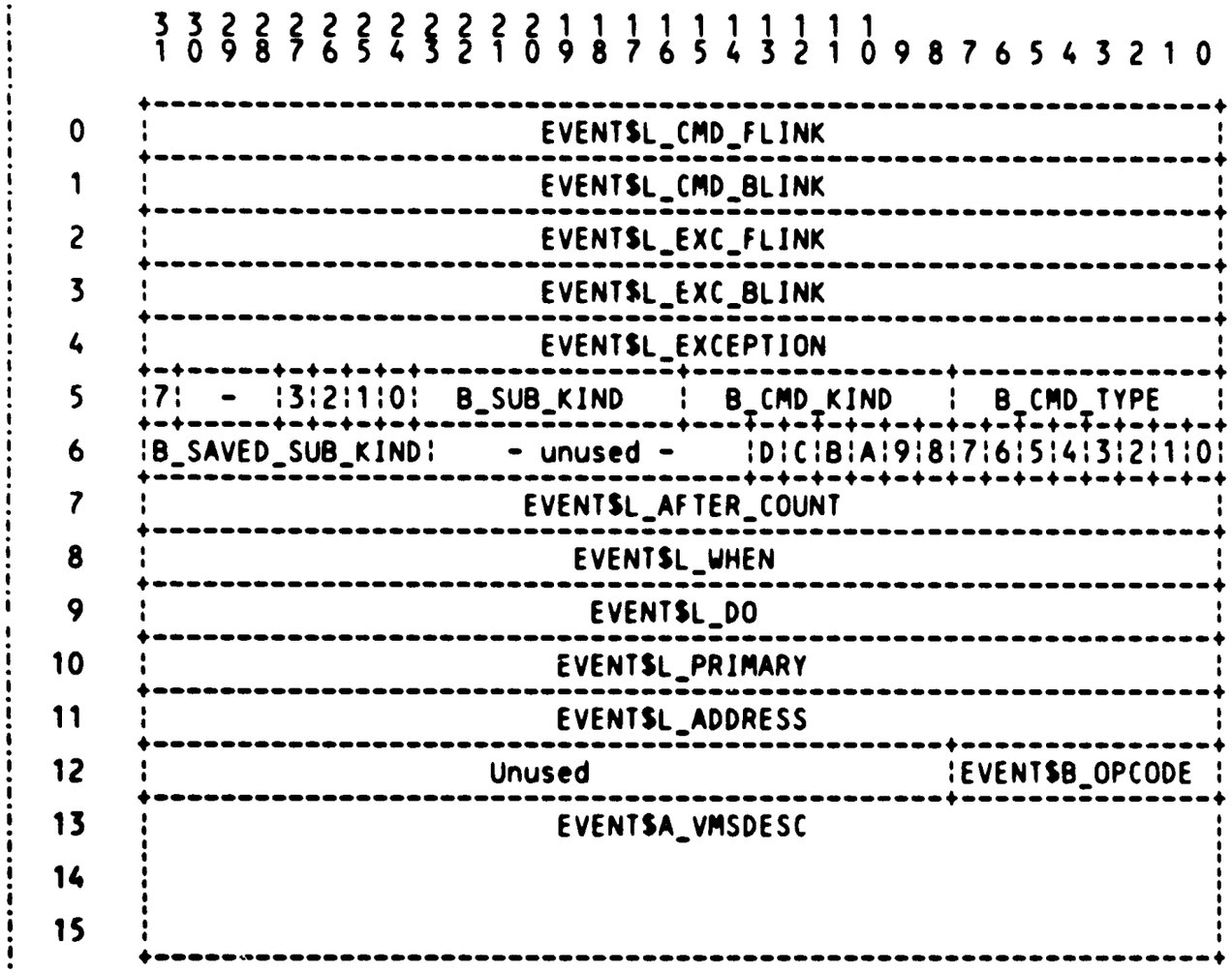
```
        EVENT$V_STEP_SOURCE =   [6,V_( 5)], ! Flag set if step/source
        EVENT$V_OVRD_NOSYSTEM = [6,V_( 6)], ! Flag set if override step/nosystem
        EVENT$V_STEP_NOSYSTEM = [6,V_( 7)], ! Flag set if step/nosystem
        EVENT$V_OVRD_NOSILENT = [6,V_( 8)], ! Flag set if override step/nosilent
        EVENT$V_STEP_NOSILENT = [6,V_( 9)], ! Flag set if step/nosilent

        EVENT$V_STEP_BPT    =   [6,V_(10)], ! Flag set if step over BPT active
        EVENT$V_STEP_TICK   =   [6,V_(11)], ! Flag set if new step
        EVENT$V_STEP_NOLINE  =  [6,V_(12)], ! Flag set if line not found

        EVENT$V_BREAK_ADDRESS = [6,V_(13)], ! Flag set if the prolog address is
                                            !  used
        EVENT$B_SAVED_SUB_KIND= [6,B3_],    ! Saved value of SUB_KIND

        EVENT$L_AFTER_COUNT =   [7,L_],     ! After count
        EVENT$L_STEP_COUNT  =   [7,L_],     ! Step count
        EVENT$L_SSV_COUNT   =   [7,L_],     ! Address of a count which keeps
                                            ! track of the number of SSV calls
                                            ! originated from one SSV

        EVENT$L_WHEN        =   [8,L_],     ! WHEN expression pointer
        EVENT$L_USERS_PC    =   [8,L_],     ! User's PC for STEP/RETURN

        EVENT$L_DO          =   [9,L_],     ! DO command list pointer
        EVENT$L_USERS_FP    =   [9,L_],     ! User's FP for STEP/RETURN and
                                            ! [BREAK:TRACE]/RETURN skips

        EVENT$L_PRIMARY     =   [10,L_],    ! Primary pointer
        EVENT$L_OPCODE_LIST =   [10,L_],    ! Instruction opcode list

        EVENT$L_ADDRESS     =   [11,L_],    ! Byte address
        EVENT$L_FP_RET_CNT  =   [11,L_],    ! Address of a count which keeps
                                            ! track of RET for SSV return levels

        EVENT$L_VALDESC     =   [12,L_],    ! Value descriptor pointer

        EVENT$B_OPCODE      =   [12,B0_],   ! Opcode byte

        EVENT$L_THREAD      =   [12,L_],    ! Thread

        EVENT$A_VMSDESC     =   [13,A_],    ! VMS Descriptor

        EVENT$L_STEP_LO_PC  =   [13,L_],    ! Step line lo PC
        EVENT$L_STEP_HI_PC  =   [14,L_],    ! Step line hi PC

        EVENT$L_CALL_FRAME   =  [15,L_]     ! Pointer to Saved value from FP

        TES;

LITERAL
    EVENT$K_EVENT_DESCRIPTOR_SIZE = 16;     ! Descriptor size in longwords

MACRO
    EVENT$EVENT_DESCRIPTOR =
        BLOCK [EVENT$K_EVENT_DESCRIPTOR_SIZE]
        FIELD (EVENT$EVENT_DESCRIPTOR_FIELDS) %;
```

```
! Macro to assign an increasing value to each member of a <name> list,
! starting with <constant>.
!
MACRO
    ENUMERATE (CONSTANT, NAME) [] =
        NAME = CONSTANT
        %IF %NULL (%REMAINING)
        %THEN %EXITMACRO
        %FI
        , ENUMERATE (CONSTANT + 1, %REMAINING) %;

! The following are the allowed values of the EVENT$B_CMD_TYPE field.
!
LITERAL
    ENUMERATE(0,                            ! Starting with 0...
          EVENT$K_TYPE_BREAK,               ! Break
          EVENT$K_TYPE_TRACE,               ! Trace
          EVENT$K_TYPE_WATCH,               ! Watch
          EVENT$K_TYPE_STEPS,               ! Steps
          EVENT$K_TYPE_SKIPS,               ! Skips
          EVENT$K_TYPE_IGNOR,               ! Ignored activating an event
          EVENT$K_TYPE_SSINT                ! System service Intercept Event
        );

! The following are the allowed values of the EVENT$B_CMD_KIND field.
!
LITERAL
    ENUMERATE(0,                            ! Starting with 0...
          EVENT$K_KIND_ACC,                 ! Access (/READ, etc.)
          EVENT$K_KIND_INS,                 ! Instruction
          EVENT$K_KIND_EXC,                 ! Exception
          EVENT$K_KIND_SKP                  ! Skips
        );

! The following are the allowed values of the EVENT$B_SUB_KIND field.
!
LITERAL
    ENUMERATE(0,                            ! Starting with 0...
          EVENT$K_ACC_READ,                 ! Access Read
          EVENT$K_ACC_WRIT,                 ! Access Write
          EVENT$K_ACC_MDFY,                 ! Access Modify
          EVENT$K_ACC_EXEC,                 ! Access Execute
          EVENT$K_ACC_RTRN,                 ! Access Return
          EVENT$K_INS_CALL,                 ! Instruction Call
          EVENT$K_INS_BRAN,                 ! Instruction Branch
          EVENT$K_INS_LINE,                 ! Instruction Line
          EVENT$K_INS_EVRY,                 ! Instruction Every
          EVENT$K_INS_USER,                 ! Instruction User
          EVENT$K_SKP_READ,                 ! Skipping Read
          EVENT$K_SKP_WRIT,                 ! Skipping Write
          EVENT$K_SKP_MDFY,                 ! Skipping Modify
```

```
        EVENT$K_SKP_EXEC,               ! Skipping Execute
        EVENT$K_SKP_RTRN,               ! Skipping Return
        EVENT$K_EXC_EXC
       );
```

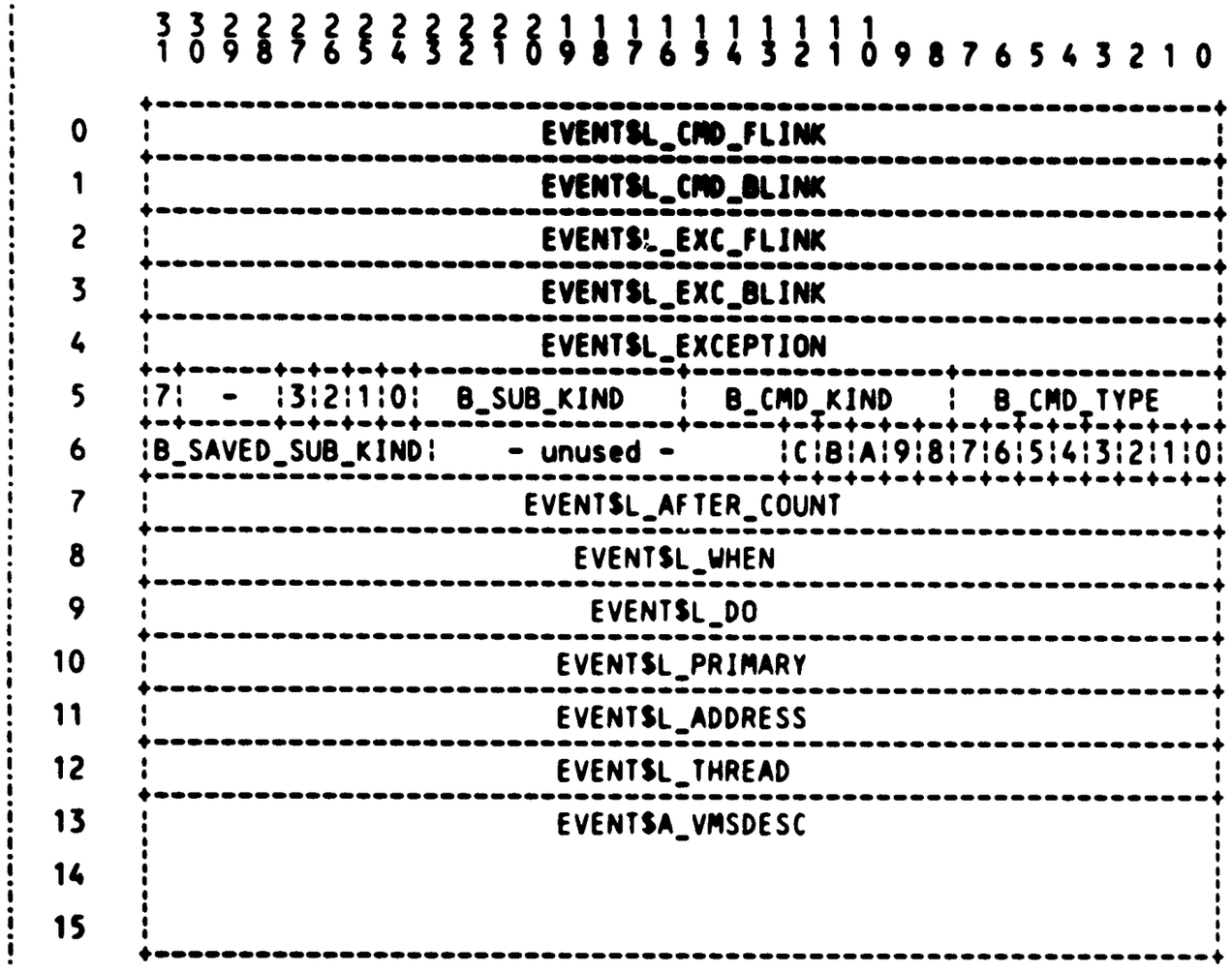
EVENT TABLE ENTRY FOR [BREAK!TRACE] [/RETURN] POINTS

This form of the Event Table Entry is used for ordinary and /RETURN breakpoints and tracepoints. It is thus used whenever the implementation of the eventpoint is to store a BPT instruction at the event location. This form is not used for threaded code (used only for COBOL-74), however.



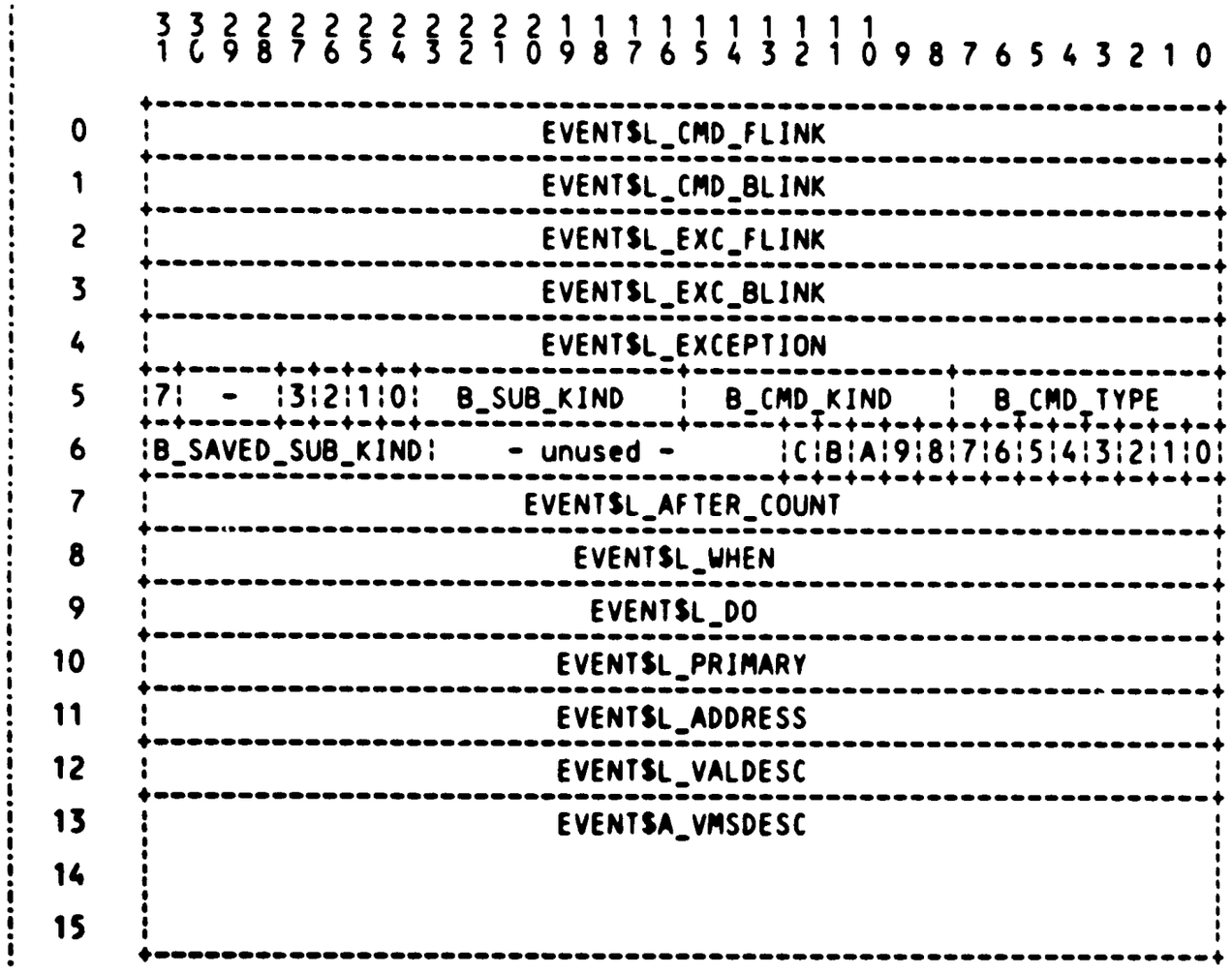
EVENT TABLE ENTRY FOR TREADED CODE EVENT POINTS

This form of the Event Table Entry is used for breakpoints and tracepoints in threaded code. Threaded code is only used by COBOL-74 and is thus obsolescent, but we still support it.



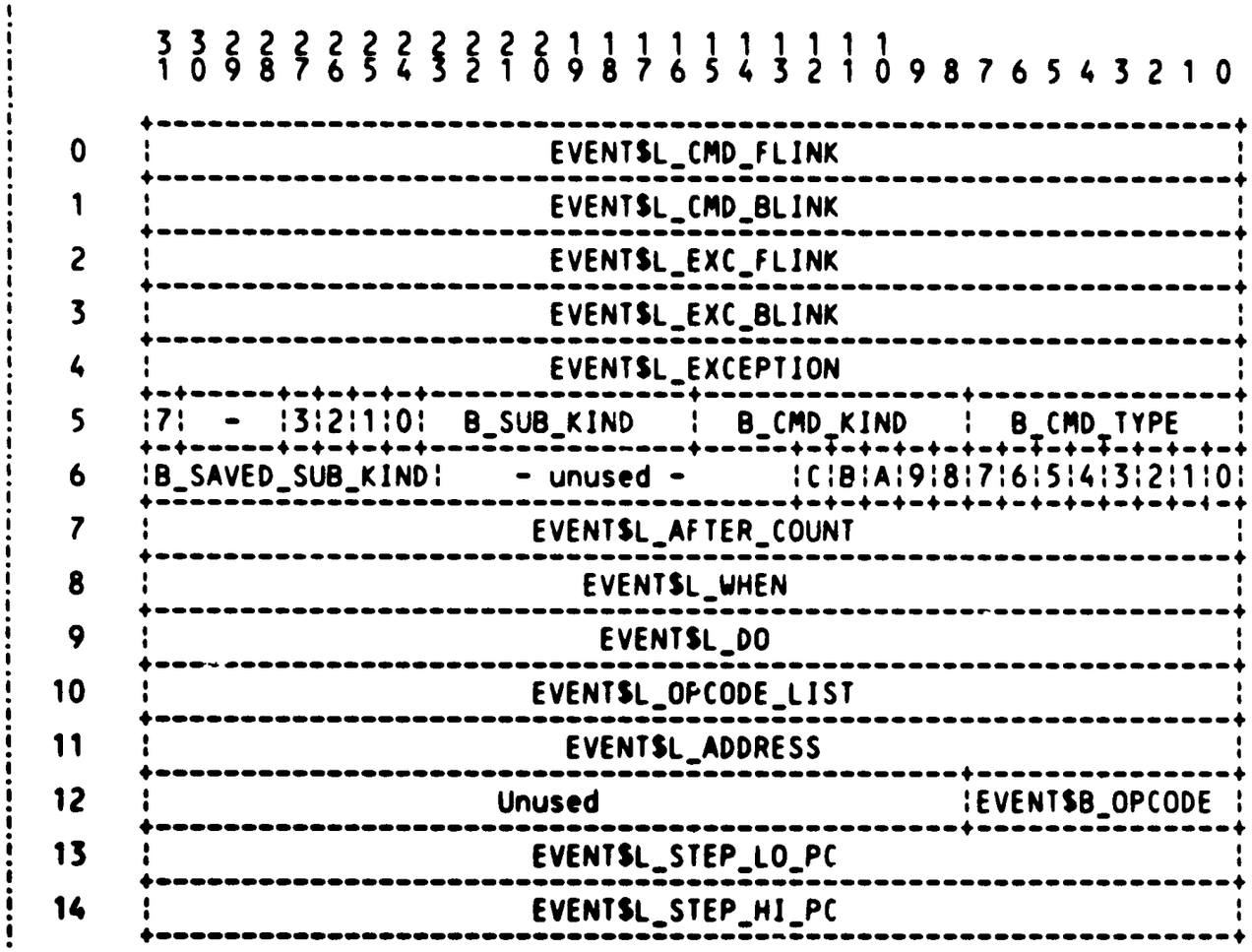
EVENT TABLE ENTRY FOR /READ, /WRITE, /MODIFY AND WATCHPOINTS

This form of the Event Table Entry is used in all cases when data locations are watched for read accesses, write accesses, or modifying accesses. It is thus used both for SET BREAK/MODIFY, for example, and for the SET WATCH command.



EVENT TABLE ENTRY FOR /CALL, /BRANCH, /INSTRUCTION

This form of Event Table Entry is used for the SET BREAK and SET TRACE commands when the /CALL, /BRANCH, or /INSTRUCTION qualifier is present. It is thus used in all cases where instructions must be traced (T-bitted), one instruction at a time, and the event is activated when an instruction of a certain class is encountered.



EVENT TABLE ENTRY FOR STEP

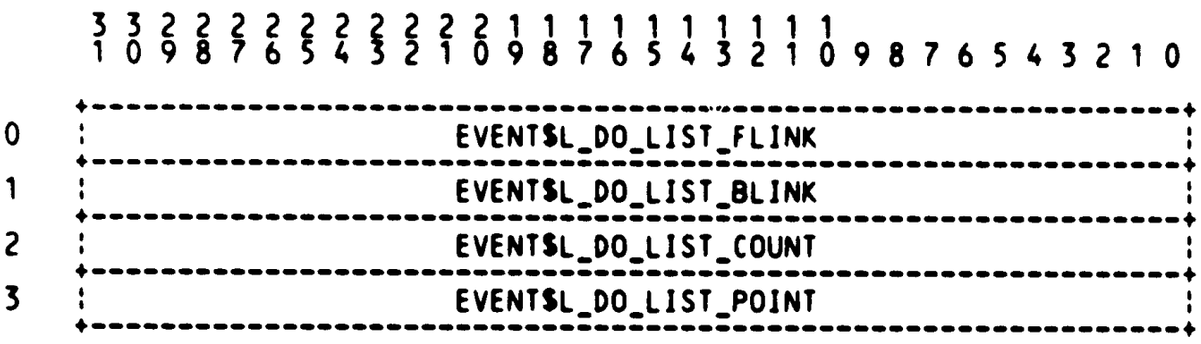
This form of Event Table Entry is used for the STEP command.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	EVENT$L_CMD_FLINK																															
1	EVENT$L_CMD_BLINK																															
2	EVENT$L_EXC_FLINK																															
3	EVENT$L_EXC_BLINK																															
4	EVENT$L_EXCEPTION																															
5	7	-	3	2	1	0	B_SUB_KIND								B_CMD_KIND								B_CMD_TYPE									
6	B_SAVED_SUB_KIND						- unused -								C	B	A	9	8	7	6	5	4	3	2	1	0					
7	EVENT$L_STEP_COUNT																															
8	EVENT$L_USERS_PC																															
9	EVENT$L_USERS_FP																															
10	EVENT$L_OPCODE_LIST																															
11	EVENT$L_ADDRESS																															
12	Unused																								EVENT$B_OPCODE							
13	EVENT$L_STEP_LO_PC																															
14	EVENT$L_STEP_HI_PC																															
15	EVENT$L_CALL_FRAME																															

```
!                E V E N T   S T E P P I N G   D E S C R I P T O R
!
!       This structure defines the current step type(s).
!
!        3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
!        1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
!       +-----------------------------------------+-+-+-+-+---------------+
!   0   !XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX!^!%!$!#!@!       *       !
!       +-----------------------------------------+-+-+-+-+---------------+
!   1   !                    EVENT$L_STEPPING_OP_LIST                     !
!       +-----------------------------------------------------------------+
!
!       * :     [0] EVENT$B_STEPPING_KIND
!       @ :     [0] EVENT$V_STEPPING_LINE
!       # :     [0] EVENT$V_STEPPING_OVER
!       $ :     [0] EVENT$V_STEPPING_SOURCE
!       % :     [0] EVENT$V_STEPPING_NOSYSTEM
!       ^ :     [0] EVENT$V_STEPPING_NOSILENT
!
!       A pointer to an Step Descriptor Entry is declared as follows:
!
!                       STEPPINGPTR: REF EVENT$STEPPING_DESCRIPTOR
!
! Define the fields of the Step Descriptor Entry.  Also declare the maximum
! size of an Step Descriptor Entry and define the declaration macro.
!
FIELD EVENT$STEPPING_DESC_FIELDS =
        SET
        EVENT$A_STEPPING_DESCRIPTOR =   [0,A_],     ! Event Descriptor address
        EVENT$B_STEPPING_KIND   =       [0,B0_],    ! Step kind (as EVENT$B_SUB_KIND)
        EVENT$V_STEPPING_LINE   =       [0,V_(8)],  ! Stepping /LINE
        EVENT$V_STEPPING_OVER   =       [0,V_(9)],  ! Stepping /OVER
        EVENT$V_STEPPING_SOURCE =       [0,V_(10)], ! Stepping /SOURCE
        EVENT$V_STEPPING_NOSYSTEM =     [0,V_(11)], ! Stepping /NOSYSTEM
        EVENT$V_STEPPING_NOSILENT =     [0,V_(12)], ! Stepping /NOSILENT
        EVENT$L_STEPPING_OP_LIST =      [1,L_]      ! /CALL, /BRANCH, etc.
        TES;

LITERAL
    EVENT$K_STEPPING_DESC_SIZE = 2;         ! Descriptor size in longwords

MACRO
    EVENT$STEPPING_DESCRIPTOR =
        BLOCK [EVENT$K_STEPPING_DESC_SIZE]
        FIELD (EVENT$STEPPING_DESC_FIELDS) %;
```

EVENT DO LIST DESCRIPTOR

An Event DO List Descriptor is created for each event-point which has an associated DO-action. This descriptor points to the text of the DO-action itself and is thus what allows the DO-action to be accessed when the event-point is taken. This is the format of the Event DO List Descriptor:



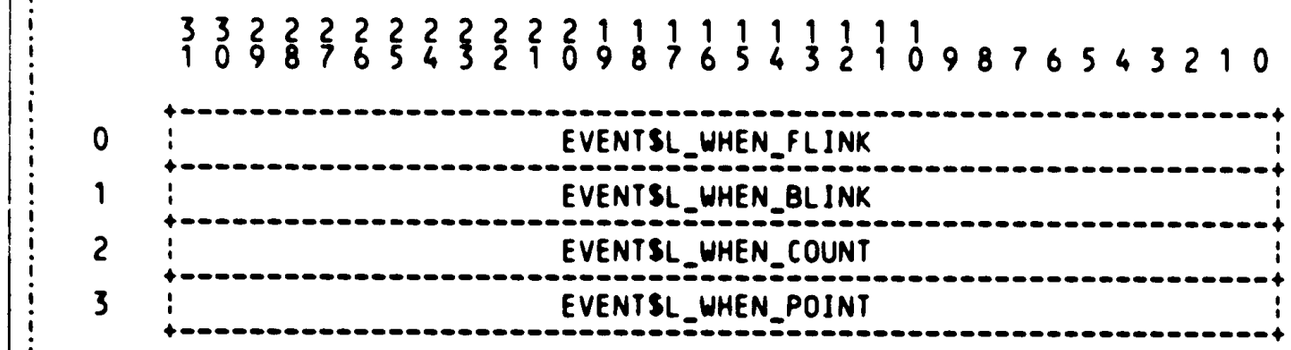
A pointer to an Event DO List Descriptor is declared as follows:

```
DOPTR: REF EVENT$DO_LIST_DESCRIPTOR
```

```
! Define the fields of the Event DO List Descriptor.  Also declare the
! descriptor size and define the declaration macro.

FIELD   EVENT$DO_LIST_DESCRIPTOR_FIELDS =
        SET
        EVENT$A_DO_LIST_DESCRIPTOR = [0,A_],! DO List Descriptor address

        EVENT$L_DO_LIST_FLINK   =   [0,L_], ! DO queue forward
        EVENT$L_DO_LIST_BLINK   =   [1,L_], !    and backward links

        EVENT$L_DO_LIST_COUNT   =   [2,L_], ! DO reference count

        EVENT$L_DO_LIST_POINT   =   [3,L_]  ! DO list pointer
        TES;

LITERAL
    EVENT$K_DO_LIST_DESCRIPTOR_SIZE =   4;  ! Descriptor size (longwords)

MACRO
    EVENT$DO_LIST_DESCRIPTOR =
        BLOCK [EVENT$K_DO_LIST_DESCRIPTOR_SIZE]
        FIELD (EVENT$DO_LIST_DESCRIPTOR_FIELDS) %;
```

EVENT WHEN DESCRIPTOR

An Event WHEN Descriptor is created each time a SET BREAK, SET TRACE, or SET WATCH statement has a WHEN clause. The Event WHEN Descriptor points to the location of the boolean WHEN-expression and allows this expression to be evaluated each time the X-point is reached. This is the format of the Event WHEN Descriptor:



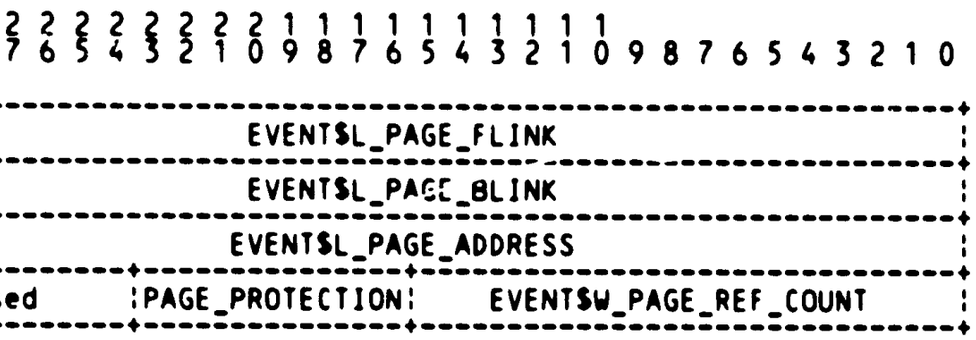
A pointer to an Event WHEN Descriptor is declared as follows:

```
WHENPTR: REF EVENT$WHEN_DESCRIPTOR
```

```
! Define the fields of the Event WHEN Descriptor, its size, and its
! declaration macro.

FIELD EVENT$WHEN_DESCRIPTOR_FIELDS =
        SET
        EVENT$A_WHEN_DESCRIPTOR =   [0,A_], ! WHEN Descriptor address
        EVENT$L_WHEN_FLINK      =   [0,L_], ! WHEN queue forward link
        EVENT$L_WHEN_BLINK      =   [1,L_], ! WHEN queue backward link
        EVENT$L_WHEN_COUNT      =   [2,L_], ! WHEN reference count
        EVENT$L_WHEN_POINT      =   [3,L_]  ! WHEN list pointer
        TES;

LITERAL
    EVENT$K_WHEN_DESCRIPTOR_SIZE =      4;  ! Descriptor size (longwords)

MACRO
    EVENT$WHEN_DESCRIPTOR =
        BLOCK [EVENT$K_WHEN_DESCRIPTOR_SIZE]
        FIELD (EVENT$WHEN_DESCRIPTOR_FIELDS) %;
```

EVENT PAGE DESCRIPTOR

An Event Page Descriptor is created for each page which is write-protected to watch-point one or more variables on that page. Event Page Descriptors are linked together on a doubly linked list which is pointed to by variable XXXXXX. This is the format of an Event Page Descriptor:



A pointer to an Event Page Descriptor is declared as follows:

```
PAGEDESC: REF EVENT$PAGE_DESCRIPTOR
```

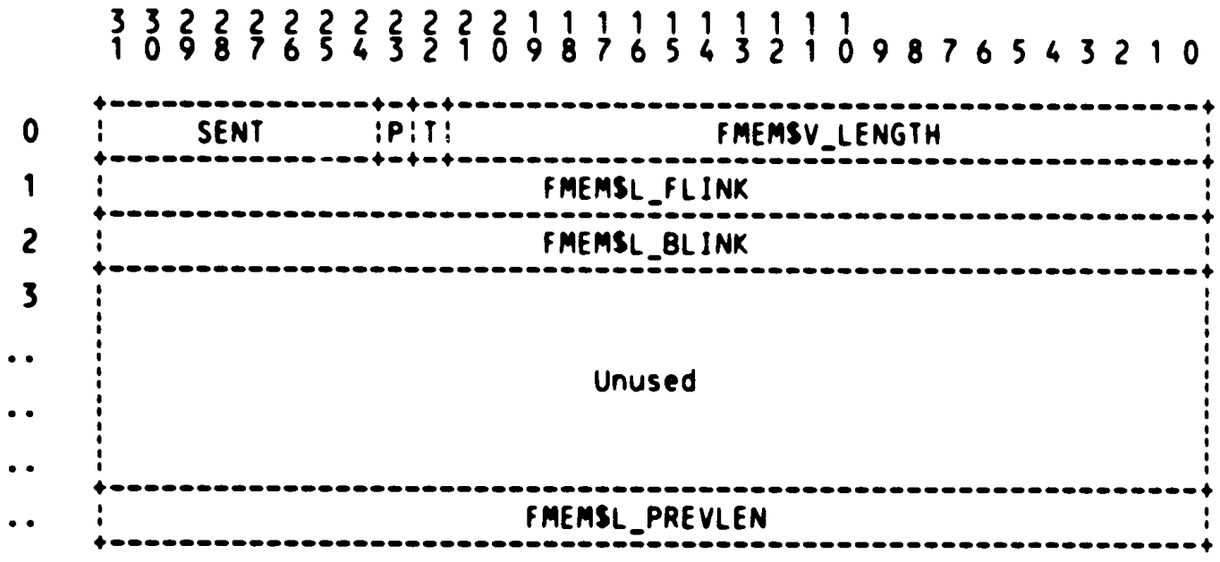
```
! Define the fields of the Event Page Descriptor.  Also define the descriptor
! size and the declaration macro.

FIELD EVENT$PAGE_DESCRIPTOR_FIELDS =
        SET
        EVENT$A_PAGE_DESCRIPTOR =   [0,A_], ! Page Descriptor address
        EVENT$L_PAGE_FLINK      =   [0,L_], ! Page queue forward link
        EVENT$L_PAGE_BLINK      =   [1,L_], ! Page queue backward link
        EVENT$L_PAGE_ADDRESS    =   [2,L_], ! Page byte address
        EVENT$W_PAGE_REF_COUNT  =   [3,W0_],! Page referecne count
        EVENT$B_PAGE_PROTECTION =   [3,B2_] ! Page old protection
        TES;

LITERAL
    EVENT$K_PAGE_DESCRIPTOR_SIZE =      4;  ! Descriptor size (longwords)

MACRO
    EVENT$PAGE_DESCRIPTOR =
        BLOCK [EVENT$K_PAGE_DESCRIPTOR_SIZE]
        FIELD (EVENT$PAGE_DESCRIPTOR_FIELDS) %;
```

F R E E M E M O R Y M A N A G E M E N T

The Free Memory Manager manages the Debugger's free memory pool. Its routines are thus called throughout the Debugger to allocate and deallocate memory blocks used for RST entries, Static Address Table entries, and numerous other kinds of descriptors and records. The Free Memory Manager has two kinds of blocks in its memory pool: free blocks and allocated blocks. The formats of both kinds of blocks and the overall structure of the memory pool is illustrated below.

FORMAT OF A FREE MEMORY BLOCK

The format of a free memory block is shown here:

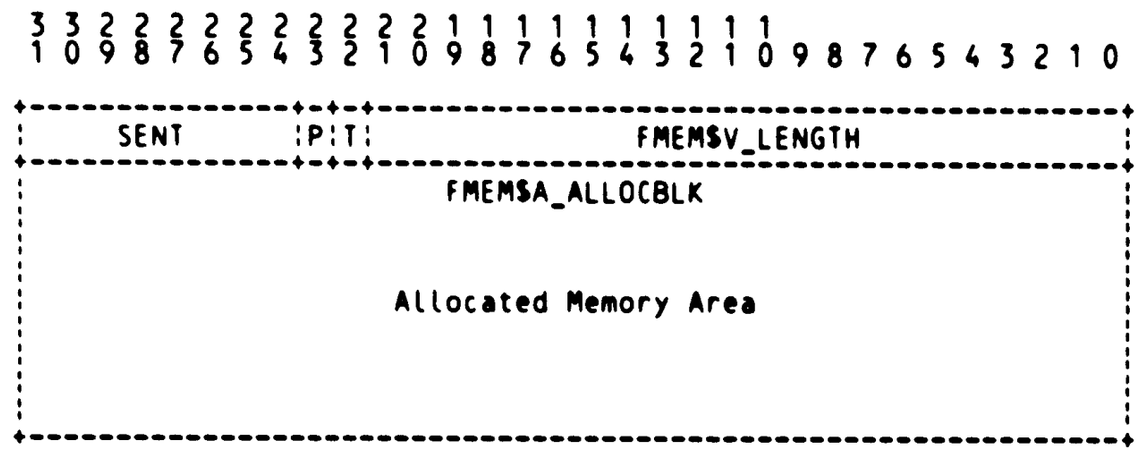


Here SENT represents FMEM$B_SENTINEL, P is FMEM$V_PREVALLOC, and T is FMEM$V_THISALLOC as defined on the next page.

A free block has its length (in longwords) stored both at the beginning of the block (FMEM$V_LENGTH) and at the end (FMEM$L_PREVLEN). The minimum length of a free block is four longwords. Free blocks are chained together on a doubly linked free-list. This list has a list head which is never allocated--the list is thus never empty. The address of the free-list list head is always stored in the OWN variable DBG$FREE_LIST.

FORMAT OF AN ALLOCATED MEMORY BLOCK

The format of an allocated block is as follows:



Here SENT represents FMEM$B_SENTINEL, P is FMEM$V_PREVALLOC, and T is FMEM$V_THISALLOC as defined below. In an allocated block, the address returned to the requestor is represented by FMEM$A_ALLOCBLK.

A pointer to a memory block (free or allocated) is declared as follows:

```
BLKPTR: REF FMEM$BLOCK
```

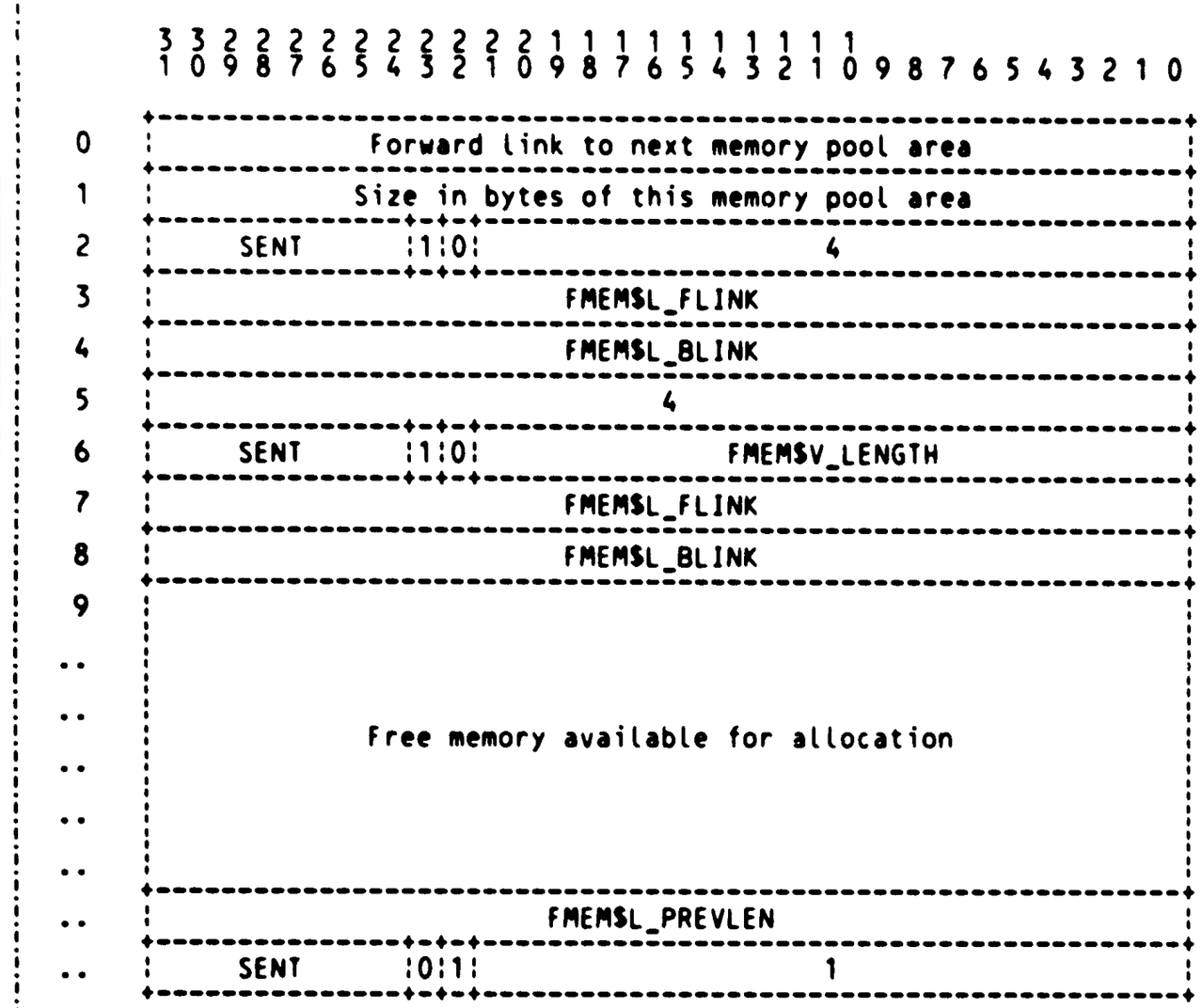
This declaration is used in the Free Memory Manager only.

```
! Define the fields in the two kinds of memory blocks.

FIELD FMEM$FLD_DEF =
        SET
        FMEM$V_LENGTH    = [ 0, V_(0,22) ],! The length of the block in longwords
        FMEM$V_THISALLOC= [ 0, V_(22) ],! Set to TRUE if this block is allocated
        FMEM$V_PREVALLOC= [ 0, V_(23) ],! Set to TRUE if the previous block is
                                            allocated (block at smaller addr)
        FMEM$B_SENTINEL = [ 0, B3_ ],    ! Sentinel value--always FMEM$K_SENTINEL
        FMEM$L_FLINK    = [ 1, L_ ],     ! Forward link on free list
        FMEM$L_BLINK    = [ 2, L_ ],     ! Backward link on free list
        FMEM$L_PREVLEN  = [ -1, L_ ],    ! Length of this free block relative to
                                            the start of the next block
        FMEM$A_ALLOCBLK = [ 1, A_ ],     ! Address returned to caller requesting
                                            a new memory block
        FMEM$A_HEADER   = [ -1, A_ ]     ! Address of block's header relative to
                                            the FMEM$A_ALLOCBLK location
        TES;
```

```
! Declare the sentinel value and the declaration macro.
!
LITERAL
        FMEM$K_SENTINEL = %X'B2';       ! Magic sentinel value--used for error
                                        !       checking only

MACRO
        FMEM$BLOCK = BLOCK[,LONG] FIELD(FMEM$FLD_DEF) %;
```

OVERALL STRUCTURE OF MEMORY POOL

The memory pool consists of one or more memory pool areas. The first such area is initialized to have the format shown here:

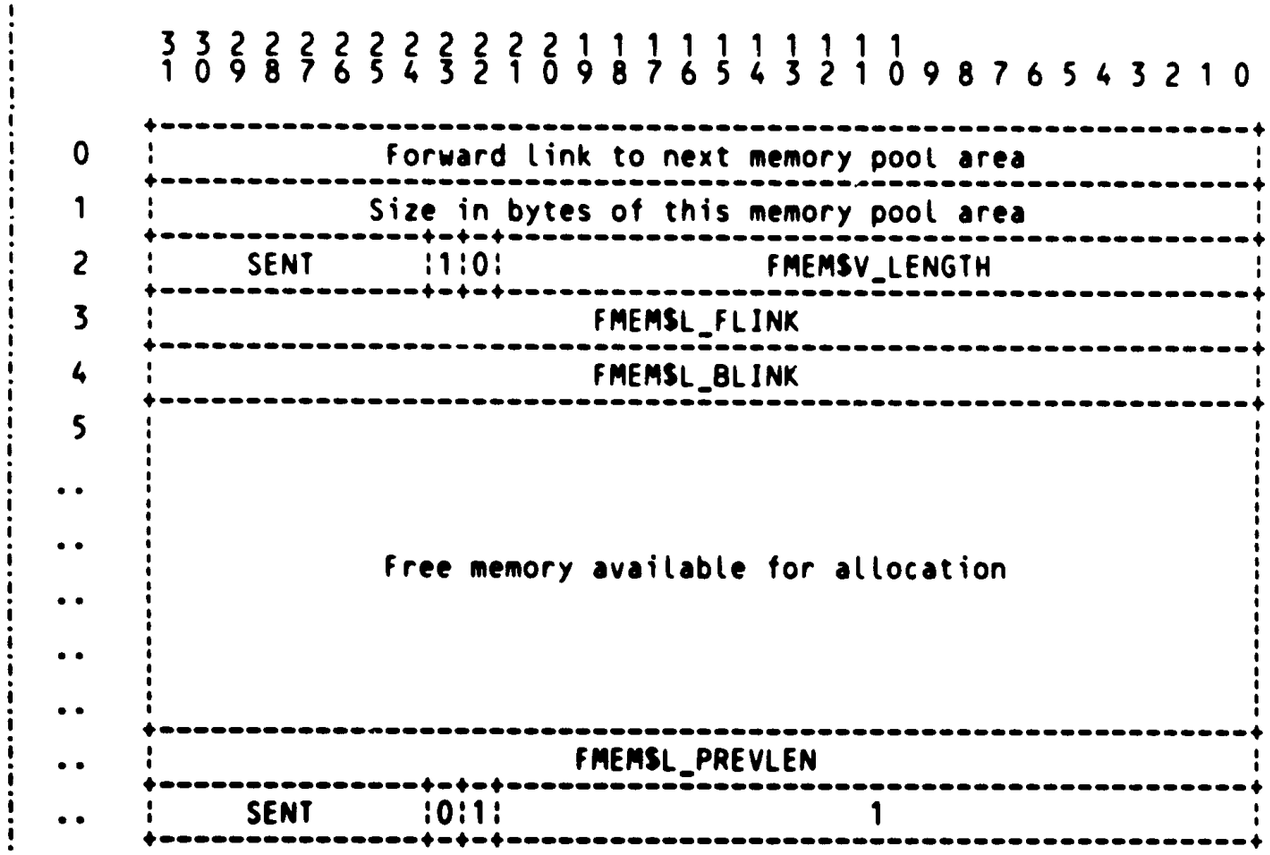


Here longword 0 contains a pointer to the next memory pool area (for this initial area the pointer is always zero) and longword 1 contains the size of this area in bytes. These two fields are present for internal error checking purposes only.

Longwords 2 - 5 contain the free-list list head. This is a free block which is always at the start of the doubly linked free list and which

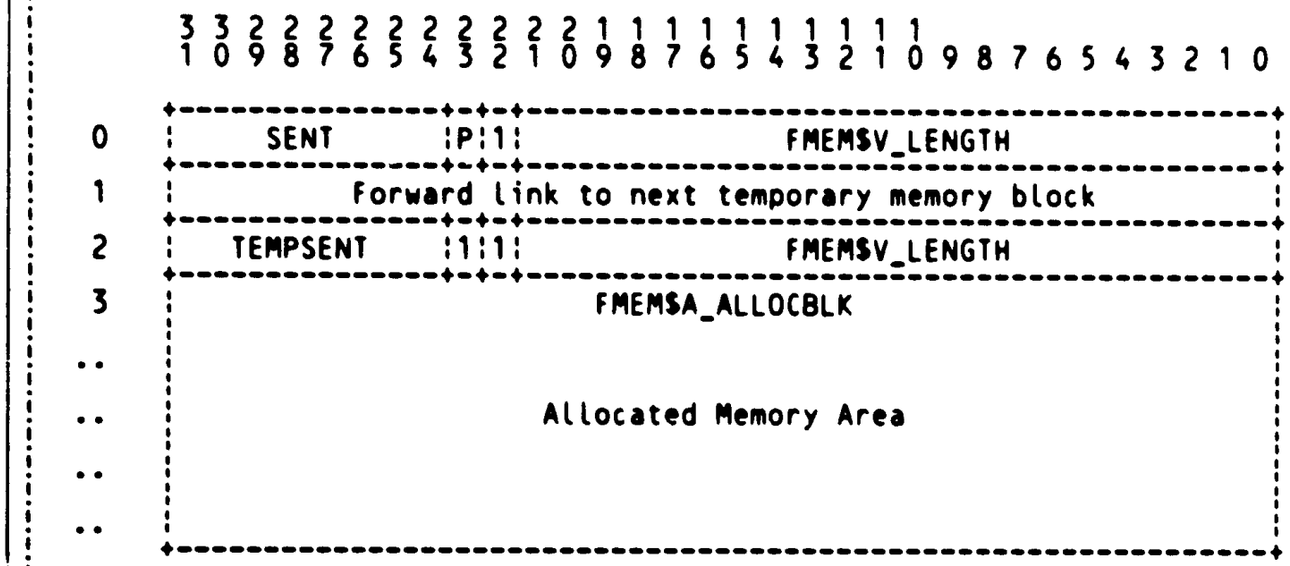
is never coalesced with any other free block--the code thus never has to worry about a completely empty free list. The very last longword in the memory pool area is the header of an allocated block. It prevents any attempt to coalesce free blocks off the end of the area. The rest of the area (longword 6 to the next to last longword) is initially one big free block from which allocation can take place. Note how the P and T flags (FMEM$V_PREVALLOC and FMEM$V_THISALLOC) are set to prevent free block coalescing off either end of the memory pool area.

Additional memory pool areas may be acquired from LIB$GET_VM during the Debugger's initialization phase. Each such area is initialized to have the same format as the first one except that the free-list list head is absent--only one is needed for the whole memory pool. The initial format of each additional memory pool area is illustrated here:



TEMPORARY MEMORY BLOCKS

"Temporary" memory blocks are blocks which automatically are released at the end of the current command. Such blocks are allocated by a special allocation routine (DBG$GET_TEMPMEM), but they do not have to be explicitly released to the memory pool. Instead one routine is called (DBG$REL_TEMPMEM) which automatically releases all "temporary" blocks accumulated since the last such call. Temporary blocks have the following format:

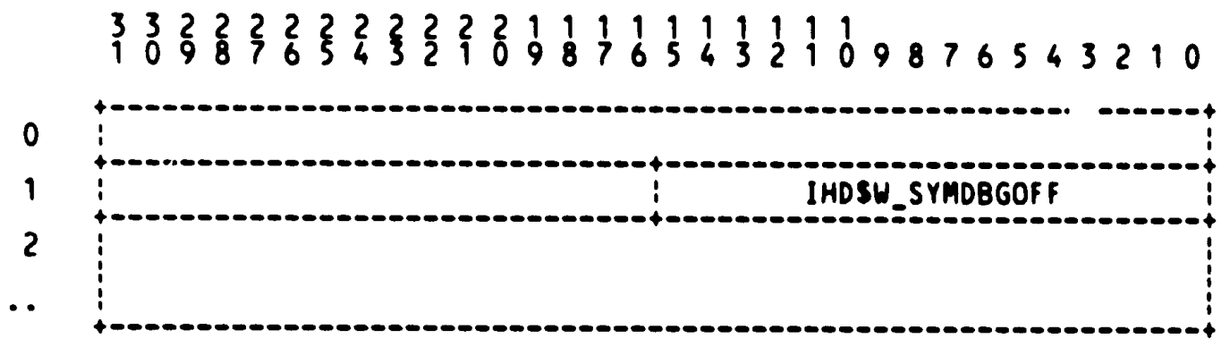


Longword 0 is the same as for any other memory pool block. Longword 1 contains a forward link to the next block on the temporary block list. Longword 2 is similar to longword 0, but the length there is two less and the sentinel value is a different sentinel value (FMEM$K_TEMPSENT).

```
! Define the distinguishing sentinel value for "temporary" memory blocks.

LITERAL
        FMEM$K_TEMPSENT = %X'B4';       ! Sentinel value in longword 2
```

I M A G E F I L E H E A D E R D E F I N I T I O N S

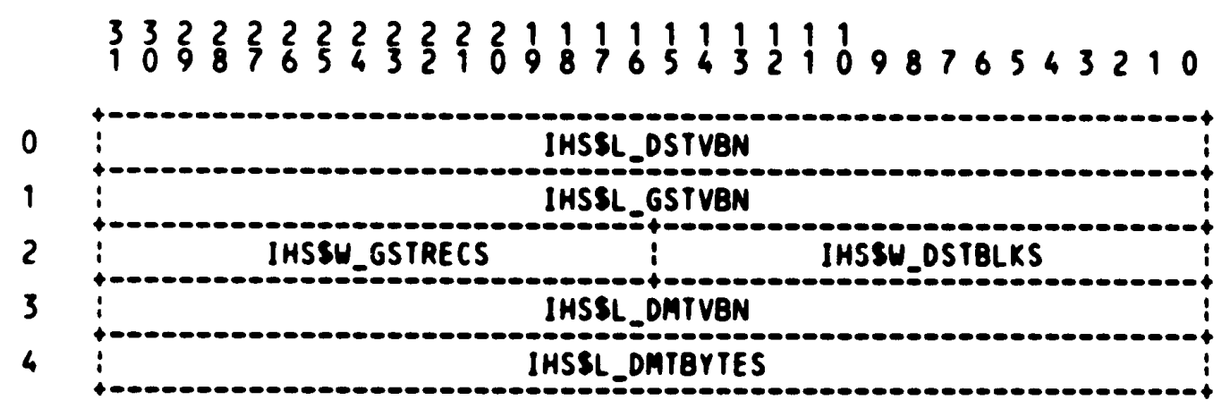
The executable image file header block contains a pointer in a fixed location which points to a small block later in the header which gives the size and location of the Debug Symbol Table (DST) and the Global Symbol Table (GST). The first part of the executable image file header looks as follows:



Here IHD$W_SYMDBGOFF contains the byte offset relative to the start of the header of an Image Header Symbol Table descriptor.

IMAGE HEADER SYMBOL TABLE DESCRIPTOR

The Image Header Symbol Table Descriptor (IHS) pointed to by the IHD$W_SYMDBGOFF field in the header has the following format:



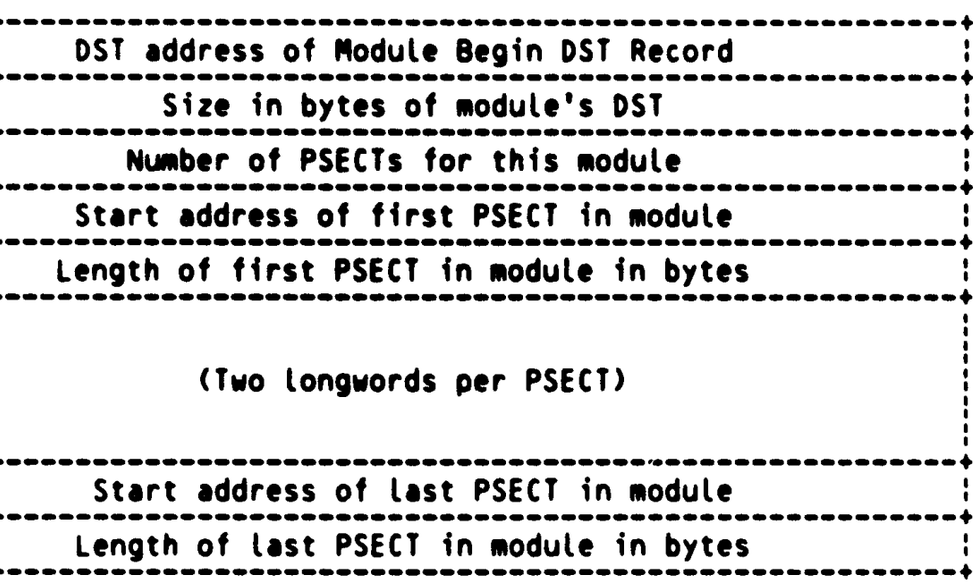
Here IHS$W_DSTBLKS and IHS$L_DSTVBN give the size (in blocks) and loca-

```
!       tion (Virtual Block Number) of the Debug Symbol Table (DST) within the
!       executable image file.  The fields IHS$W_GSTRECS and IHS$L_GSTVBN give
!       the size (in GST records) and start location (Virtual Block Number) of
!       the Global Symbol Table (GST).  Finally, the fields IHS$L_DMTBYTES and
!       IHS$L_DMTVBN give the size (in bytes) and start location (Virtual Block
!       Number) of the Debug Module Table (DMT).  The DMT is described below.
!       These field names are declared by macros in SYS$LIBRARY:LIB.L32.
!
!       The symbol IHD$W_SYMDBGOFF is defined in SYS$LIBRARY:LIB.L32 and is
!       therefore not defined here.  A pointer to the image file header is
!       declared as follows:
!
!                       IHDPTR: REF BLOCK[,BYTE]
!
!       A pointer to the image header symbol descriptor is declared like this:
!
!                       IHSPTR: REF IHS$ENTRY
!
!
! Define the IHS$ENTRY macro.
!
MACRO
        IHS$ENTRY = BLOCK[IHS$K_LENGTH,BYTE] %; ! IHS$K_LENGTH is defined in
                                                !       SYS$LIBRARY:LIB.L32
```

IMAGE FILE DEBUG MODULE TABLE

The Image File Header in an executable image file points to the Image Header Symbol Table descriptor as described above. If bit 5 of field IHD$L_LNKFLAGS in the image header is set, this is a "new" image, i.e. one produced by the VMS V4.0 or later linker, and the IHS$L_DMTVBN and IHS$L_DMTBYTES fields exist in the Image Header Symbol Table descriptor. (If bit 5 is not set, this is an "old" image and those fields do not exist.) If non-zero, IHS$L_DMTVBN gives the Virtual Block Number in the image file of the Debug Module Table (the DMP). IHS$L_DMPBYTES then gives the size of the DMT in bytes. The DMT is only built if the user did a LINK/DEBUG; if he did not, IHS$L_DMTVBN and IHS$L_DMTBYTES are zero.

The Debug Module Table contains one entry per module in the Debug Symbol Table (the DST). This is the format of each such DMT entry:



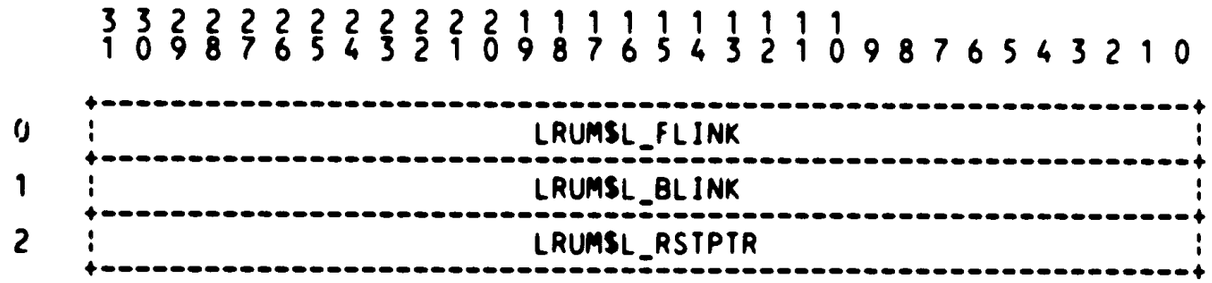
Longword 0 gives the address relative to the start of the DST of the Module Begin DST Record for this module. Longword 1 gives the size of the DST in bytes for the same module. Longword 2 gives the number of PSECTs in the module (i.e., the number of statically allocated program sections), and this is followed by that number of two-longword pairs which give the start address and length (in bytes) of each such PSECT. Since the number of PSECTs cannot exceed 65K, the upper two bytes of longword 2 are available for future expansion.

The DMT is used during DEBUG initialization to initialize the RST and the Program Static Address Table. Using the DMT is much faster than the aternative procedure, namely reading through the entire DST to pick up the needed information. The information in the DMT entry is enough to build a Module RST Entry for each module in the DST and the PSECT information is used to build the Program SAT. The amount of RST symbol table space needed per module is not computable from the DMP information, but is estimated by multiplying the DST size of each module by an appropriate scale factor.

L E A S T R E C E N T L Y U S E D M O D U L E T A B L E

The Least Recently Used Module (LRUM) table keeps track of the Least Recently Used module in the RST. This information is used to select modules to be removed from the RST when the free memory pool is full and new space must be created by releasing old RST entries. The LRUM table is built and maintained by the DBG$MOST_RECENT_MODULE routine in module RSTACCESS and is accessed by routine DBG$GET_MEMORY when memory space must be released by removing the Least Recently Used module. Routine DBG$RST_INIT builds the initial LRUM list head.

The LRUM table consists of a doubly linked list of LRUM entries, with a permanent list head and one entry per module currently in the RST. The address of the list head is given by the global LRUM$LISTHEAD. The format of the individual LRUM entry is show here:



A pointer to a Least Recently Used Module entry is declared as follows:

```
LRUMPTR: REF LRUM$ENTRY
```

```
! Define the fields in the Least Recently Used Module entry.  Also define the
! entry size and the declaration macro.

FIELD LRUM$FLD_DEF =
        SET
        LRUM$L_FLINK    = [ 0, L_ ],    ! The LRUM chain forward link
        LRUM$L_BLINK    = [ 1, L_ ],    ! The LRUM chain backward link
        LRUM$L_RSTPTR   = [ 2, L_ ]     ! A pointer to the Module RST Entry
        TES;

LITERAL
        LRUM$ENTSIZE    = 3;            ! The longword size of the LRUM entr,

MACRO
        LRUM$ENTRY = BLOCK[,LONG] FIELD(LRUM$FLD_DEF) %;        ! Declaration macro
```

L E X I C A L S C A N N E R C H A R A C T E R T A B L E

The Lexical Scanner (routine DBG$LEXICAL_SCANNER) operates on a Character Table which gives lexical information about each ASCII character. This information consists of bits and small integer values which characterize the character from the Lexical Scanner's point of view--they indicate whether a given character can be part of an identifier, a numeric constant, an operator, or whatever. The Character Table is thus a vector of 256 elements, indexed by the ASCII character code, where each element contains the information defined here.

A Lexical Scanner Character Table is declared as follows:

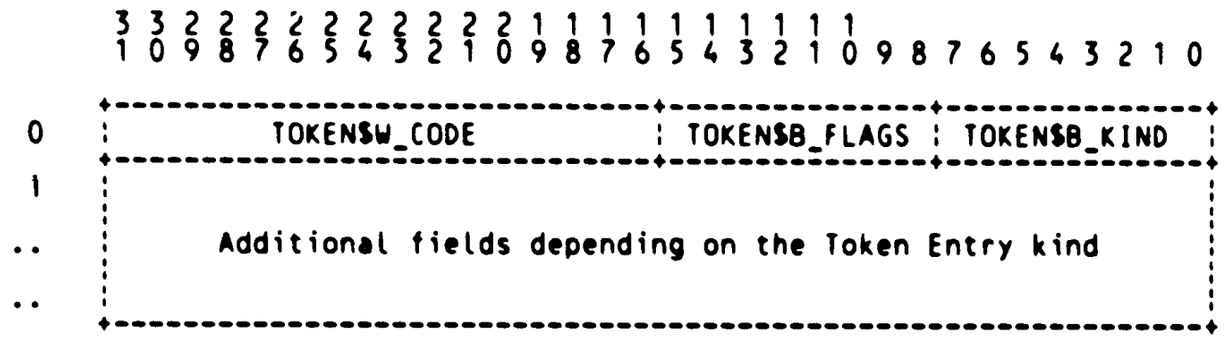
```
CHARTBL: CHRTBL$TABLE
```

```
! Define the fields of the Character Table element.  Also define the
! declaration macro.

FIELD CHRTBL$FLD_DEF =
        SET
        CHRTBL$V_IDENT_START    = [ 0, V_(0) ],  ! Identifier start character
        CHRTBL$V_IDENT_MIDDLE   = [ 0, V_(1) ],  ! Identifier middle character
        CHRTBL$V_IDENT_END      = [ 0, V_(2) ],  ! Identifier end character
        CHRTBL$V_NUMBER_START   = [ 0, V_(3) ],  ! Number start character
        CHRTBL$V_NUMBER_CLASS   = [ 0, V_(4,4)], ! Numeric constant character
                                                 !      class
        CHRTBL$V_ALPHABETIC     = [ 0, V_(8) ],  ! Alphabetic character (A-Z)
        CHRTBL$V_DIGIT          = [ 0, V_(9) ],  ! Numeric digit (0-9)
        CHRTBL$V_STRING_QUOTE   = [ 0, V_(10) ], ! String quote character
        CHRTBL$V_OPCHAR         = [ 0, V_(11) ], ! Operator character
        CHRTBL$V_OPCHAR_PREFIX  = [ 0, V_(12) ], ! Operator prefix character
        CHRTBL$V_OPCHAR_INFIX   = [ 0, V_(13) ], ! Operator infix character
        CHRTBL$V_OPCHAR_POSTFIX = [ 0, V_(14) ], ! Operator postfix character
        CHRTBL$V_ADDRESS_OP     = [ 0, V_(15) ], ! Address Expression operator
        CHRTBL$V_SPECIAL_SYMBOL = [ 0, V_(16) ], ! Special DEBUG symbol (. \ ^)
        CHRTBL$V_SPACE          = [ 0, V_(17) ], ! Space or Tab character
        CHRTBL$V_TERMINATOR     = [ 0, V_(18) ], ! May start Terminator Token
        CHRTBL$V_SPECIAL_CASE   = [ 0, V_(19) ], ! Special case in lex scanner
        CHRTBL$L_WHOLE_ENTRY    = [ 0, L_ ]      ! The whole entry for a char.
        TES;

MACRO
        CHRTBL$TABLE = BLOCKVECTOR[256,1,LONG] FIELD(CHRTBL$FLD_DEF) %;


! Define mask values for all the bits in the Character Table element.  These
! values are used to initialize Character Table elements.
!
LITERAL
        CHRTBL$M_NOTHING        = 0,     ! No bits set
        CHRTBL$M_IDENT_START    = 1^0,   ! Identifier start character
```

```
        CHRTBL$M_IDENT_MIDDLE   = 1^1,  ! Identifier middle character
        CHRTBL$M_IDENT_END      = 1^2,  ! Identifier end character
        CHRTBL$M_IDENT_ANYWHERE = 7^0,  ! Anywhere in an identifier
        CHRTBL$M_NUMBER_START   = 1^3,  ! Number start character
        CHRTBL$M_ALPHABETIC     = 1^8,  ! Alphabetic character (A-Z)
        CHRTBL$M_DIGIT          = 1^9,  ! Numeric digit (0-9)
        CHRTBL$M_STRING_QUOTE   = 1^10, ! String quote character
        CHRTBL$M_OPCHAR         = 1^11, ! Operator character
        CHRTBL$M_OPCHAR_PREFIX  = 1^12, ! Operator prefix character
        CHRTBL$M_OPCHAR_INFIX   = 1^13, ! Operator infix character
        CHRTBL$M_OPCHAR_POSTFIX = 1^14, ! Operator postfix character
        CHRTBL$M_ADDRESS_OP     = 1^15, ! Address Expression operator
        CHRTBL$M_SPECIAL_SYMBOL = 1^16, ! Special DEBUG symbol (. \ ^)
        CHRTBL$M_SPACE          = 1^17, ! Space or tab character
        CHRTBL$M_TERMINATOR     = 1^18, ! May start terminator token
        CHRTBL$M_SPECIAL_CASE   = 1^19; ! Special case in lexical scanner

! Note that the possible values of the CHRTBL$V_NUMBER_CLASS field are defined
! under the description of the Number Scanner State Table.  They are the same
! values as are allowed in the NUMST$B_CHAR_CLASS field and their names all
! have the form NUMST$K_CLASS_xxx.
```

L E X I C A L T O K E N E N T R Y

A Lexical Token Entry is returned by the Lexical Scanner every time it is called. Operator Token Entries are found in predefined tables specific to each language supported by DEBUG while Operand Token Entries are built as needed by the Lexical Scanner (DBG$LEXICAL_SCANNER). The format of a Lexical Token Entry depends on the kind of Token Entry it is, but all Lexical Token Entries have this general format:



A pointer to a Lexical Token Entry is defined as follows:

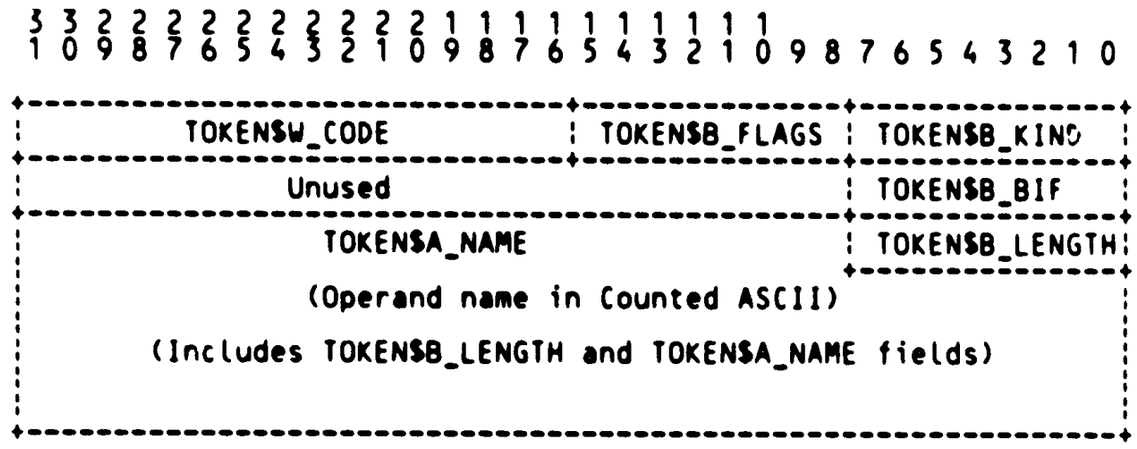
```
TOKENPTR: REF TOKEN$ENTRY
```

```
! Define the fields of the Lexical Token Entry.

FIELD TOKEN$FLD_DEF =
        SET
        TOKEN$B_KIND     = [ 0, B0_ ],    ! Token kind
        TOKEN$B_FLAGS    = [ 0, B1_ ],    ! Token Entry flag bits
        TOKEN$V_PRIMARY  = [ 0, V1_(0) ], ! Token is Primary Symbol operator
                                          !     (used in Operator entries only)
        TOKEN$V_LEXICAL  = [ 0, V1_(1) ], ! Token is a lexical operator (used
                                          !     in Operator entries only)
        TOKEN$V_SGNEXT   = [ 0, V1_(2) ], ! Sign extension S in X<P,S,E> operation
                                          !     (used in Operator Lexical Token
                                          !     for the bit-select operation)
        TOKEN$V_BALANCED_PARENS           ! Token is Terminator Token only if
                         = [ 0, V1_(0) ], !     parentheses are balanced
                                          !     (used in Terminator entries only)
        TOKEN$V_MUST_BE_SINGLE            ! Token is Terminator Token only if this
                         = [ 0, V1_(1) ], !     character is not followed by it-
                                          !     self (: is term but not ::)
                                          !     (used in Terminator entries only)
        TOKEN$V_ARGUMENT_LIST             ! Argument list follows this Ada tick
                         = [ 0, V1_(3) ], !     operator.
        TOKEN$W_CODE     = [ 0, W1_ ],    ! Operand or operator code for token
        TOKEN$B_BIF      = [ 1, B0_ ],    ! Built-in Function Code
```

```
        TOKEN$B_LENGTH  = [ 2, B0_ ],   ! Length of operand name in bytes
        TOKEN$A_NAME    = [ 2, A1_ ],   ! Operand name in ASCII

        TOKEN$B_R_PREC  = [ 1, B0_ ],   ! Operator's Right Precedence
        TOKEN$B_L_PREC  = [ 1, B1_ ],   ! Operator's Left Precedence
        TOKEN$W_SUBCODE = [ 1, W1_ ],   ! Sub-code field
        TOKEN$W_BIT_OFFSET
                        = [ 2, W0_ ],   ! Offset P in X<P,S,E> operation
        TOKEN$W_BIT_LENGTH
                        = [ 2, W1_ ],   ! Length S in X<P,S,E> operation
        TOKEN$B_OPLEN   = [ 3, B0_ ],   ! Length of operator name in bytes
        TOKEN$A_OPNAME  = [ 3, A1_ ]    ! Operator name in ASCII
        TES;

LITERAL
        TOKEN$K_ENTSIZE = 2,            ! Size of fixed part of an Operand
                                        !       Lexical Token Entry
        TOKEN$K_ENTSIZE_OPERATOR = 3;   ! Size of fixed part of an Operator
                                        !       Lexical Token Entry

MACRO
        TOKEN$ENTRY = BLOCK[,LONG] FIELD(TOKEN$FLD_DEF) %;      ! Declaration macro

! Define the valid values of the TOKEN$B_KIND field.  This field indicates
! what kind of token this is.
!
LITERAL
        TOKEN$K_OPERAND         = 1,    ! Operand
        TOKEN$K_PREFIX_OP       = 2,    ! Prefix Operator
        TOKEN$K_INFIX_OP        = 3,    ! Infix Operator
        TOKEN$K_POSTFIX_OP      = 4;    ! Postfix Operator

! Define mask values for the bits in the TOKEN$B_FLAGS field.
!
LITERAL
        TOKEN$M_PRIMARY         = 1^0,  ! Token is Primary Symbol operator
        TOKEN$M_LEXICAL         = 1^1,  ! Token is lexical operator
        TOKEN$M_BALANCED_PARENS = 1^0,  ! Terminator requires balanced parens
        TOKEN$M_MUST_BE_SINGLE  = 1^1;  ! Terminator character must be single
```

OPERAND LEXICAL TOKEN ENTRY

Operands are identifiers, numeric constants, string or character constants, or special constructs such as %LINE nnn which generate identifiers. The Lexical Token Entry for an operand has the following format:



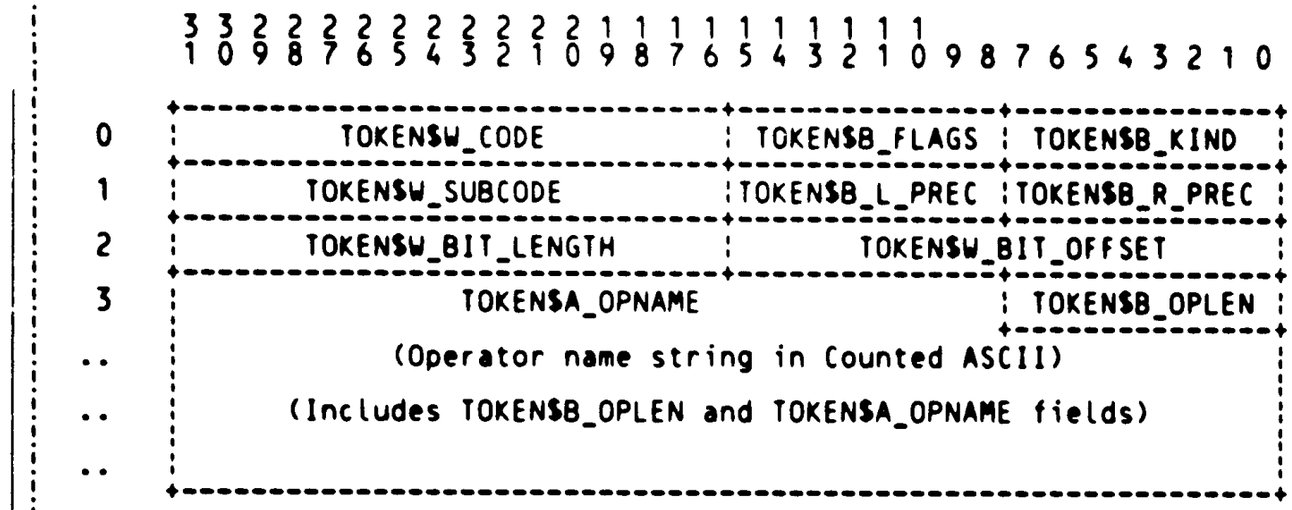
The TOKEN$B_KIND field of an Operand Lexical Token Entry always has the value TOKEN$K_OPERAND.

Define the valid values of the TOKEN$W_CODE field for operands. This field says what kind of operand this Token Entry represents.

```
LITERAL
        TOKEN$K_MIN_OPERAND     = 1,    ! --- Minimum value for TOKEN$W_CODE
        TOKEN$K_IDENTIFIER      = 1,    ! Identifier
        TOKEN$K_STRING          = 2,    ! Character String Constant
        TOKEN$K_CHARACTER       = 3,    ! Single Character Constant
        TOKEN$K_INTEGER         = 4,    ! Integer Constant
        TOKEN$K_HEX_INTEGER     = 5,    ! Hexadecimal Integer Constant
        TOKEN$K_FLOATING        = 6,    ! Floating Point Constant
        TOKEN$K_EXP_E_FLOAT     = 7,    ! Floating Constant with E Exponent
        TOKEN$K_EXP_D_FLOAT     = 8,    ! Floating Constant with D Exponent
        TOKEN$K_EXP_Q_FLOAT     = 9,    ! Floating Constant with Q Exponent
        TOKEN$K_BIN_INTEGER     = 10,   ! Binary Constant
        TOKEN$K_OCT_INTEGER     = 11,   ! Octal Constant
        TOKEN$K_COMPLEMENT      = 12,   ! 1's Complement
        TOKEN$K_BIT_STRING      = 13,   ! Bit String Constant (as in PL/I)
        TOKEN$K_PACK_DECIMAL    = 14,   ! Packed Decimal Constant
        TOKEN$K_EXP_G_FLOAT     = 15,   ! Floating Constant with G Exponent
        TOKEN$K_BUILTIN_FUNCTION= 16,   ! Built-in Function
        TOKEN$K_MAX_OPERAND     = 16;   ! --- Maximum value for TOKEN$W_CODE
```

OPERATOR LEXICAL TOKEN ENTRY

The Lexical Token Entry for an operator contains additional information not needed for operands, namely the left and right precedences of the operator. This is the format of an Operator Lexical Token Entry:



The TOKEN$B_KIND field of an Operator Lexical Token Entry always has one of the three values TOKEN$K_PREFIX_OP, TOKEN$K_INFIX_OP, or TOKEN$K_POSTFIX_OP.

```
! Define some size literals used for allocating memory.

LITERAL
        TOKEN$K_FIXED_SIZE_BYTE = 13,           ! Fixed size (not including
                                                ! the name) of an operator
                                                ! token. Actual size in
                                                ! bytes is this value plus
                                                ! the OPLEN field.
        TOKEN$K_FIXED_SIZE_LONG = 4;            ! Fixed size (not including
                                                ! the name) of an operator
                                                ! token. Actual size in
                                                ! longwords is this value
                                                ! plus OPLEN / 4.

! Define the valid values of the TOKEN$W_CODE field for operators.  This field
! indicates which operation is to be performed.  Not all languages allow all
! operators, but all operators allowed in at least one language are listed in
! this table.  Note that the same operator may be represented differently in
! different languages--thus "=" and ".EQ." are both considered to be the same
```

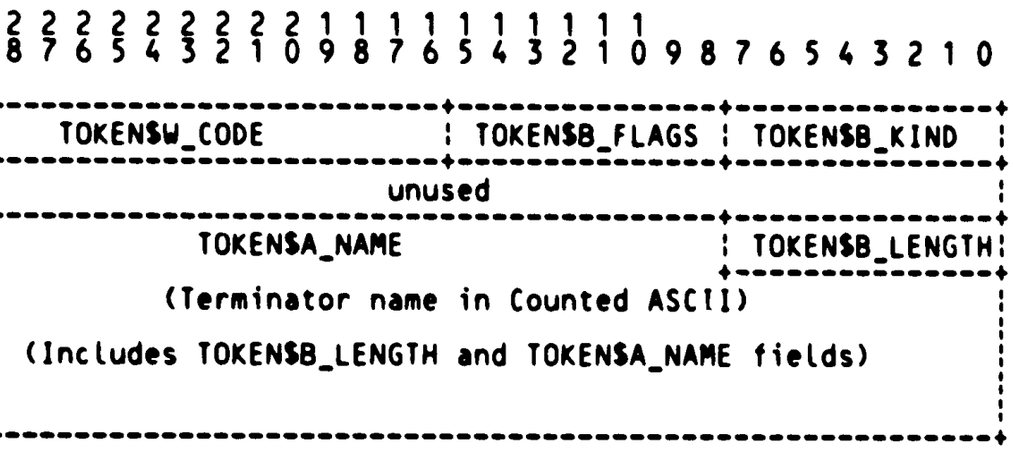
```
! comparison operator.
!
LITERAL
        TOKEN$K_MIN_PRIMARY_OP  = 1,     ! --- Minimum value of TOKEN$W_CODE
                                         ! --- for Primary Symbol operators
        TOKEN$K_PRIMARY_TERM    = 1,     ! Terminator operator for primary
        TOKEN$K_GLOBAL_SLASH    = 2,     ! Prefix backslash for GST scope (\X)
        TOKEN$K_BACKSLASH       = 3,     ! Infix backslash for pathname qualifi-
                                         !     cation (EXAM M\R\X)
        TOKEN$K_SUBSCRIPT       = 4,     ! Subscript open parenthesis operator
        TOKEN$K_DOT             = 5,     ! Infix data qualification operator
                                         !     as in A.B(2).C in PL/I
        TOKEN$K_COBOL_OF        = 6,     ! Infix data qualification operator
                                         !     as in C OF B OF A(2) in COBOL
        TOKEN$K_PASCAL_DEREF    = 7,     ! Postfix dereference operator such as
                                         !     ^ in PASCAL (as in P^.X)
        TOKEN$K_PLI_DEREF       = 8,     ! Infix dereference operator such as
                                         !     -> in PL/I (as in P->X)
        TOKEN$K_INVOCNUM        = 9,     ! Invocation Number operator
        TOKEN$K_BIF_OP          = 10,    ! Built-in Function (parameter list)
        TOKEN$K_ADA_TICK        = 11,    ! The ADA tick operator for symbol
                                         !     attribute qualification.
        TOKEN$K_MAX_PRIMARY_OP  = 11;    ! --- Maximum value for TOKEN$W_CODE
                                         ! --- for Primary Symbol operators


LITERAL
        TOKEN$K_MIN_OPERATOR    = 1,     ! --- Minimum value for TOKEN$W_CODE
                                         ! --- for language expression operators
        TOKEN$K_INITIATOR       = 1,     ! Expression initiator pseudo-operator
        TOKEN$K_TERMINATOR      = 2,     ! Expression terminator pseudo-operator
        TOKEN$K_INDIRECT        = 3,     ! Indirection operator (prefix . or @)
        TOKEN$K_UNARY_PLUS      = 4,     ! Unary plus operator (+A)
        TOKEN$K_UNARY_MINUS     = 5,     ! Unary minus operator (-A)
        TOKEN$K_ADD             = 6,     ! Addition operator (A+B)
        TOKEN$K_SUBTRACT        = 7,     ! Subtraction operator (A-B)
        TOKEN$K_MULTIPLY        = 8,     ! Multiplication operator (A*B)
        TOKEN$K_DIVIDE          = 9,     ! Division operator (A/B)
        TOKEN$K_POWER_OF        = 10,    ! Exponentiation operator (A**B)
        TOKEN$K_OPENPAREN       = 11,    ! Open expression parenthesis "("
        TOKEN$K_CLOSEPAREN      = 12,    ! Close expression parenthesis ")"
        TOKEN$K_EQUAL           = 13,    ! Equal comparison operator
        TOKEN$K_NOT_EQUAL       = 14,    ! Not equal comparison operator
        TOKEN$K_GTR_THAN        = 15,    ! Greater than comparison operator
        TOKEN$K_GTR_THAN_U      = 16,    ! Unsigned Greater than comparison operator
        TOKEN$K_GTR_EQUAL       = 17,    ! Greater than or equal comparison oper.
        TOKEN$K_GTR_EQUAL_U     = 18,    ! Unsigned Greater than or equal to comparison operator
        TOKEN$K_LSS_THAN        = 19,    ! Less than comparison operator
        TOKEN$K_LSS_THAN_U      = 20,    ! Unsigned Less than comparison operator
        TOKEN$K_LSS_EQUAL       = 21,    ! Less than or equal comparison oper.
        TOKEN$K_LSS_EQUAL_U     = 22,    ! Unsigned Less than or equal comparison operator
        TOKEN$K_NOT             = 23,    ! Boolean NOT operator
        TOKEN$K_AND             = 24,    ! Boolean AND operator
        TOKEN$K_OR              = 25,    ! Boolean OR operator
        TOKEN$K_XOR             = 26,    ! Boolean XOR (exclusive OR) operator
        TOKEN$K_EQV             = 27,    ! Boolean EQV (equivalence) operator
```

```
        TOKEN$K_SHORT_AND       = 28,   ! Short-circuited boolean AND operator
        TOKEN$K_SHORT_OR        = 29,   ! Short-circuited boolean OR operator
        TOKEN$K_BIT_NOT         = 30,   ! Bitwise NOT operator
        TOKEN$K_BIT_AND         = 31,   ! Bitwise AND operator
        TOKEN$K_BIT_OR          = 32,   ! Bitwise OR operator
        TOKEN$K_BIT_XOR         = 33,   ! Bitwise XOR operator
        TOKEN$K_BIT_EQV         = 34,   ! Bitwise EQV operator
        TOKEN$K_CONCATENATE     = 35,   ! String concatenation
        TOKEN$K_MODULUS         = 36,   ! Modulus operator
        TOKEN$K_REMAINDER       = 37,   ! Remainder operator
        TOKEN$K_LEFT_SHIFT      = 38,   ! Left shift operator
        TOKEN$K_RIGHT_SHIFT     = 39,   ! Right shift operator
        TOKEN$K_ADDRESS_OF      = 40,   ! Address-of operator (as in C)
        TOKEN$K_SIZEOF          = 41,   ! Size-of operator (as in C)
        TOKEN$K_SET_MEMBER      = 42,   ! Set membership operator
        TOKEN$K_ABSOLUTE        = 43,   ! Absolute value operator
        TOKEN$K_INFIX_NOT       = 44,   ! Lexical operator (as in COBOL's NOT =)
        TOKEN$K_SUCCESSOR       = 45,   ! Successor operator (BIF)
        TOKEN$K_PREDECESSOR     = 46,   ! Predecessor operator (BIF)
        ! Unused                = 47,   ! Unused
        TOKEN$K_INT_DIVIDE      = 48,   ! Lexical operator for integer divide
                                        ! (in PASCAL, A DIV B must be
                                        ! distiguished from A / B)
        TOKEN$K_BITSELECT       = 49,   ! Bit selection in BLISS and address
                                        !     expressions (<pos,size,ext>)
        TOKEN$K_DEPOSIT         = 50,   ! = for Deposit Command ( acting as
                                        !     lang. dependent assignment )
        TOKEN$K_CONVERT         = 51,   ! Used as to indicate subscript
                                        !     conversion operator
        TOKEN$K_RADIX_DEC       = 52,   ! Set current radix to decimal
        TOKEN$K_RADIX_HEX       = 53,   ! Set current radix to hexadecimal
        TOKEN$K_RADIX_OCT       = 54,   ! Set current radix to octal
        TOKEN$K_RADIX_BIN       = 55,   ! Set current radix to binary
        TOKEN$K_IDENTITY        = 56,   ! Identitiy operator
        TOKEN$K_OPENSET         = 57,   ! Set constant open [
        TOKEN$K_BIT_IMP         = 58,   ! Bitwise IMP (implication) operator
        TOKEN$K_PRE_INCR        = 59,   ! Pre-increment (++X)
        TOKEN$K_POST_INCR       = 60,   ! Post-increment (X++)
        TOKEN$K_PRE_DECR        = 61,   ! Pre-decrement (--X)
        TOKEN$K_POST_DECR       = 62,   ! Post-decrement (X--)
        TOKEN$K_PREFIX_GTR      = 63,   ! Prefix > as in COBOL, ie. NOT >
        TOKEN$K_PREFIX_LSS      = 64,   ! Prefix < as in COBOL, ie. NOT <
        TOKEN$K_PREFIX_EQL      = 65,   ! Prefix = as in COBOL, ie. NOT =
        TOKEN$K_NEGCONST        = 66,   ! Negative constant
        TOKEN$K_POSCONST        = 67,   ! Positive constant
        TOKEN$K_MAX_OPERATOR    = 67;   ! --- Maximum value of TOKEN$W_CODE
                                        ! --- for language expression operators

! Define the possible values for the TOKEN$K_SUBCODE field.
! This field is used to store additional information about the operator.
! At present the one use of this field is for the ADA attribute operators.
! For these, the TOKEN$W_CODE field contains "TOKEN$K_ADA_TICK" and the
! subcode field identifies which tick operator is present (""FIRST",
! ""LAST", etc.
! The reason for doing it this way is because there may be a large number
! of tick operators, all of which are treated the same for purposes of
```

```
! operator precedence parsing. Tables thus are smaller if there is only
! one code for "tick" and a subcode for each tick operator.
!
LITERAL
        TOKEN$K_TICK_MIN        = 1,

        TOKEN$K_TICK_CONSTRAINED= 1,
        TOKEN$K_TICK_FIRST      = 2,
        TOKEN$K_TICK_LAST       = 3,
        TOKEN$K_TICK_LENGTH     = 4,
        TOKEN$K_TICK_POS        = 5,
        TOKEN$K_TICK_PRED       = 6,
        TOKEN$K_TICK_SIZE       = 7,
        TOKEN$K_TICK_SUCC       = 8,
        TOKEN$K_TICK_VAL        = 9,

        TOKEN$K_TICK_MAX        = 9;
```

TERMINATOR LEXICAL TOKEN ENTRY

Terminators are lexical tokens which mark the end of an expression. For example, in subscript expressions, "," and ")" mark the ends of the expressions as in "ARR(K+2, I*3-4)". Similarly, in the command SET BREAK X+4 DO(...), the keyword "DO" marks the end of the expression as long as it does not appear within parentheses. Terminator Lexical Token Entries define tokens which act as expression terminators in specified contexts. The Terminator Lexical Token Entry has the following format:



The TOKEN$B_KIND field of a Terminator Lexical Token Entry always has the value zero (0).

The TOKEN$B_FLAGS field can contain the flag TOKEN$V_BALANCED_PARENS which, if set, indicates that this token acts as a terminator only if parentheses are balanced when it is encountered. For example, ")" terminates a subscript expression only the subscript expression's parentheses are balanced so far. Similarly, DO terminates the address expression in SET BREAK expr DO ... only if it is not enclosed in parentheses; otherwise there would be no way to include the identifier "DO" (as allowed in many languages) in the address expression.

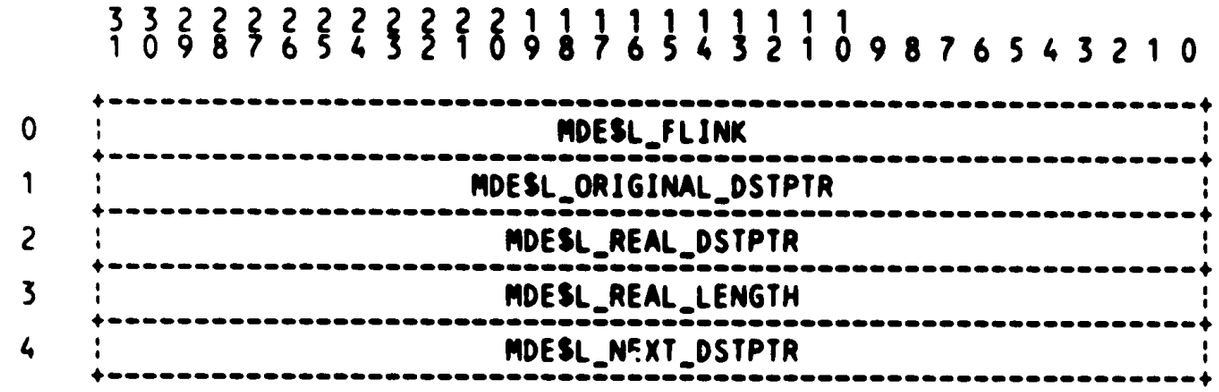
```
! Define the valid values of the TOKEN$W_CODE field for terminators.  This
! field says what kind of terminator this Token Entry represents.

LITERAL
        TOKEN$K_MIN_TERMINATOR  =  0,   ! --- Minimum value for TOKEN$W_CODE
        TOKEN$K_TERM_NONE       =  0,   ! No terminator (just end of line)
        TOKEN$K_TERM_COMMA      =  1,   ! Comma ","
        TOKEN$K_TERM_CLOSE      =  2,   ! Close paren or bracket (")" or "]")
        TOKEN$K_TERM_COLON      =  3,   ! Colon ":"
```

```
        TOKEN$K_TERM_EQUAL      =  4,   ! Equal sign "="
        TOKEN$K_TERM_DO         =  5,   ! DO keyword "DO" (as in SET BREAK)
        TOKEN$K_TERM_THEN       =  6,   ! THEN keyword "THEN" (as in IF - THEN)
        TOKEN$K_TERM_WHEN       =  7,   ! WHEN keyword "WHEN" (as in SET BREAK)
        TOKEN$K_TERM_COMCOL     =  8,   ! Allow both comma "," and colon ":"
        TOKEN$K_TERM_CMWHDO     =  9,   ! Allow comma, WHEN, and DO (SET BREAK)
        TOKEN$K_TERM_GTRTHAN    = 10,   ! Greater-than sign ">" (as in <p,s,e>)
        TOKEN$K_TERM_OPEN       = 11,   ! Open paren "("
        TOKEN$K_TERM_COMPAREN   = 12,   ! "," or ")" in CALL
        TOKEN$K_TERM_TO         = 13,   ! "TO" in FOR i=1 TO n
        TOKEN$K_TERM_BY         = 14,   ! "BY"
        TOKEN$K_TERM_DOT        = 15,   ! Dot "." (as in Set constant, ie. 1..4)
        TOKEN$K_MAX_TERMINATOR  = 15;   ! --- Maximum value for TOKEN$W_CODE
```

```
!                       L I N K E D   L I S T   N O D E
!
!
!       Linked lists using this definition are generated by language specific
!       routines to indicate:
!
!       1. The pages on which a symbol's R-value is contained.
!                       ( DBG$NXXX_GET_PAGES )
!
!       2. Primary Descriptors which represent a range specification.
!                       ( DBG$NXXX_RANGE_VAL )
!
!
!        3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
!        1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
!       +---------------------------------------------------------------+
!   0   !                     DBG$L_LINK_NODE_LINK                      !
!       +---------------------------------------------------------------+
!   1   !                     DBG$L_LINK_NODE_VALUE                     !
!       +---------------------------------------------------------------+
!
!
!       A pointer to a Link Node is declared as follows:
!
!                       LINKNODE: REF DBG$LINK_NODE
!
!***
!
! Define the Link Node fields and declaration macro.

FIELD DBG$LINK_NODE_FIELDS =
        SET
        DBG$L_LINK_NODE_LINK    = [ 0, L_ ],    ! Pointer to next node
        DBG$L_LINK_NODE_VALUE   = [ 1, L_ ]     ! Node value field
        TES;

LITERAL
        DBG$K_LINK_NODE_SIZE = 2;               ! Size of link node in
                                                ! longwords

MACRO
        DBG$LINK_NODE = BLOCK [DBG$K_LINK_NODE_SIZE]
        FIELD (DBG$LINK_NODE_FIELDS) %;
!**-
```

M O V E D D S T E N T R Y D E F I N I T I O N S

To support DST continuation records, every continued DST record is merged with its continuation records into one record and is moved to DEBUG permanent memory. A list of moved DST records is kept to allow access to these moved records. Each entry on that list has the following structure:



As the RST for a module is built, continuation records are merged and the new "moved" DST pointer is placed in the RST entry. Thus, moved DSTs are transparent for those routines that access the RST.

Those routines that obtain DST pointers from the DST itself are:

- o Routines that deal with SEPTYP records
- o Routines that access indirect type specs
- o Routines that access novel length type spes
- o Routines that access the next DST record

These routines must access those pointers through routines that search the structure to make sure that the DST hasn't been moved. The routines currently provided are:

- o DBG$RST_DST_PTR - Return the "real" DST pointer
- o DBG$RST_NEXT_DST - Return the "next" DST pointer

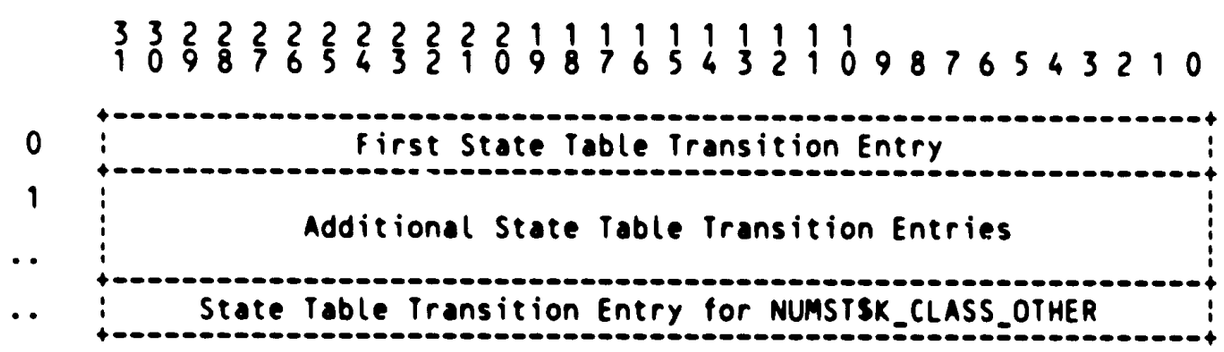
The structure is a singly linked list with its head being DBG$GL_MOVED_DST_LIST_HEAD.

```
!
! Define the fields of the Moved DST Record Entry.  Also define the entry length
! and the declaration macro.
!
FIELD MDE$FIELDS =
        SET
        MDE$L_FLINK             = [ 0, L_ ],    ! Forward link
```

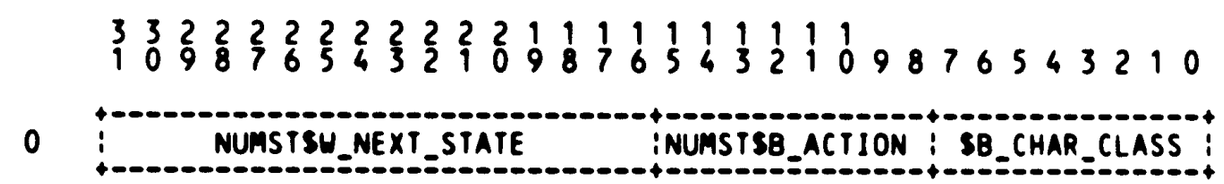
```
        MDE$L_ORIGINAL_DSTPTR   = [ 1, L_ ],    ! The Original DST pointer
        MDE$L_REAL_DSTPTR       = [ 2, L_ ],    ! The new, moved, real DST pointer
        MDE$L_REAL_LENGTH       = [ 3, L_ ],    ! The moved DST's length
        MDE$L_NEXT_DSTPTR       = [ 4, L_ ]     ! The next DST pointer
        TES;

LITERAL
        MDE$K_LENGTH = 5;               ! Length of an entry in longwords

MACRO
        MDE$RECORD = BLOCK[,LONG] FIELD(MDE$FIELDS) %;


! NOTE: MDE$L_NEXT_DSTPTR is a simple address calculation to the next DST record
! with all the continuation records consumed.  At the time the calculation is
! made it is not known if the record will be relocated.  Thus any access to it
! must also check it for being relocated by calling DBG$RST_DST_PTR.
```

N U M B E R S C A N N E R S T A T E T A B L E

The Lexical Number Scanner in routine DBG$LEXICAL_SCANNER operates off a Finite-State Machine state table which defines the allowed formats of numeric constants in the current language. There is thus one Number State Table for each language supported by DEBUG (although some languages can share state tables). Each state in the state table is represented by a list of two or more State Table Transition Entries, each of which specifies a state transition to be taken if the current input character is of a class specified in the Transition Entry. The format of a Number Scanner State Table Entry is as follows:



The individual State Table Transition Entry has the following format:



The whole Number Scanner State Table for a language simply consists of a sequence of concatenated State Table Entries, one entry per state in the Finite-State Machine. The whole State Table is thus a vector of State Table Transition Entries. Element zero of the state table is the beginning of the state table's Start State, and each transition's NUMST$W_NEXT_STATE pointer is a state table index which points to the beginning of the next state's State Table Entry.

A pointer to a Number Scanner State Table is declared as follows:

```
STATE_TABLE: REF NUMST$TABLE
```

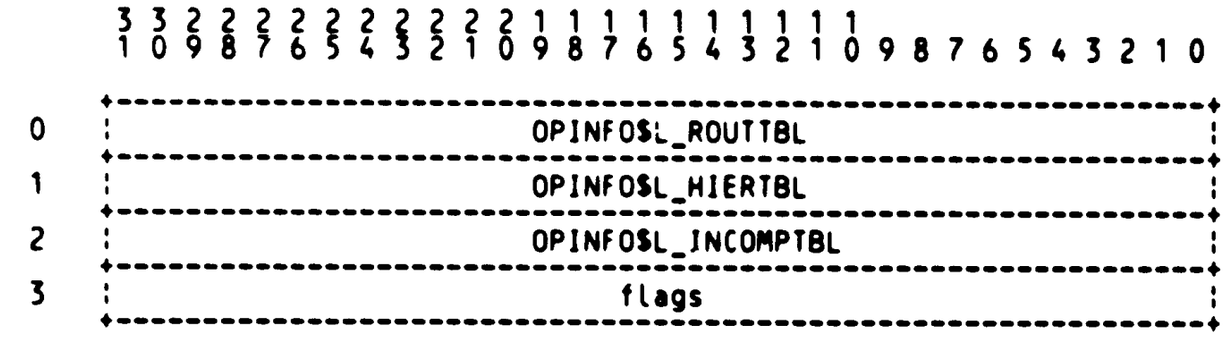
This declares STATE_TABLE to be a block vector which is referenced as .STATE_TABLE[.STATE_INDEX, field-name] where STATE_INDEX is the

```
!       vector index which points to a specific Number Scanner State Table
!       Transition Entry.
!
!
!
! Define the fields in the Number Scanner State Table Transition Entry.
! Also define the declaration macro.
!
FIELD NUMST$FLD_DEF =
        SET
        NUMST$B_CHAR_CLASS = [ 0, B0_ ],! Character class that activates
                                        !       this state transition
        NUMST$B_ACTION     = [ 0, B1_ ],! CASE index for semantic action to be
                                        !       taken during state transition
        NUMST$W_NEXT_STATE = [ 0, W1_ ] ! Index of next state after transition
        TES;

MACRO
        NUMST$TABLE = BLOCKVECTOR[,1,LONG] FIELD(NUMST$FLD_DEF) %;


! Define the allowed values of the character class field NUMST$B_CHAR_CLASS.
! These are the different classes of characters which are significant when
! scanning numeric constants.
!
LITERAL
        NUMST$K_MIN_CLASS       = 0,    ! --- Minimum character class code
        NUMST$K_CLASS_OTHER     = 0,    ! All other characters
        NUMST$K_CLASS_DIGIT     = 1,    ! The decimal digits, "0" - "9"
        NUMST$K_CLASS_HEXDIGIT  = 2,    ! The hex digits "A", "C", and "F"
        NUMST$K_CLASS_DOT       = 3,    ! The decimal point "."
        NUMST$K_CLASS_PLUS      = 4,    ! The plus sign "+" (for exponents)
        NUMST$K_CLASS_MINUS     = 5,    ! The minus sign "-" (for exponents)
        NUMST$K_CLASS_B         = 6,    ! The letter "B" (for binary)
        NUMST$K_CLASS_D         = 7,    ! The letter "D" (for exponents)
        NUMST$K_CLASS_E         = 8,    ! The letter "E" (for exponents)
        NUMST$K_CLASS_Q         = 9,    ! The letter "Q" (for exponents)
        NUMST$K_CLASS_X         = 10,   ! The letter "X" (for hexadecimal)
        NUMST$K_CLASS_UNDERSCORE= 11,   ! "_" is allowed in numbers in ADA
        NUMST$K_CLASS_POUND     = 12,   ! '#' is allowed in numbers in ADA
        NUMST$K_CLASS_G         = 13,   ! The letter "G" (for exponents)
        NUMST$K_MAX_CLASS       = 13;   ! --- Maximum character class code

! Define the valid values for the NUMST$B_ACTION field.  These values are CASE
! indices which select the semantic action routine to be executed for each
! state transition.
!
LITERAL
        NUMST$K_MIN_ACTION      = 1,    ! --- Minimum action index
        NUMST$K_ACT_DO_NOTHING  = 1,    ! No semantic action needed
        NUMST$K_ACT_GO_PAST_DIGIT = 2,  ! Go past digit in integer part
        NUMST$K_ACT_MARK_DEC_PT = 3,    ! Mark that a decimal point was found
        NUMST$K_ACT_GO_PAST_FRAC = 4,   ! Go past digit in fraction part
        NUMST$K_ACT_MARK_E_EXP  = 5,    ! Mark that E-exponent field was found
        NUMST$K_ACT_MARK_D_EXP  = 6,    ! Mark that D-exponent field was found
```

```
        NUMST$K_ACT_MARK_Q_EXP     = 7,  ! Mark that Q-exponent field was found
        NUMST$K_ACT_BACKUP_PTRS    = 8,  ! Backup pointers and return token
        NUMST$K_ACT_GOT_NUMBER     = 9,  ! We got a number--return token for it
        NUMST$K_ACT_NOT_NUMBER     = 10, ! Not a numeric constant
        NUMST$K_ACT_GIVE_ERROR     = 11, ! We should never get here--give error
        NUMST$K_ACT_COB_CKNUM      = 12, ! In COBOL this could be part of a name
        NUMST$K_ACT_COB_CKHEX      = 13, ! In COBOL this could be part of a name
                                         !      if it is not in hex mode
        NUMST$K_ACT_GO_PAST_PACK   = 14, ! Go past digit in Pack decimal
        NUMST$K_ACT_GO_PAST_PACK_FRAC=15,! Go past digit in Pack decimal
        NUMST$K_ACT_GOT_PACK_NUMBER=16,  ! We got a Packed decimal number--
                                         !      return token for it
        NUMST$K_ACT_SAVE_BASE      = 17, ! base#number in ADA
        NUMST$K_ACT_MARK_G_EXP     = 18, ! Mark that G-exponent field was found
        NUMST$K_MAX_ACTION         = 18; ! --- Maximum action index
```

O P E R A T O R I N F O R M A T I O N T A B L E

The Operator Information Table is a blockvector indexed by Operator Code (e.g., TOKEN$K_SUBTRACT) which gives pointers to tables which contain the information necessary to select the type conversions and then the routine invocations which evaluate the operators of a given language.



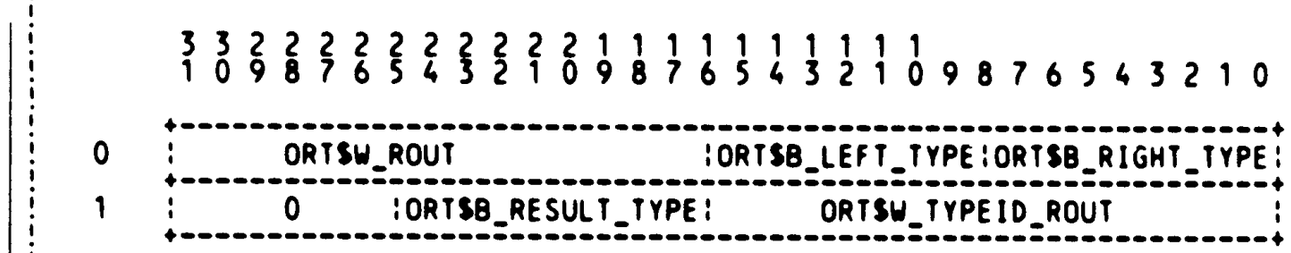
A pointer to an Operator Information Table is declared as follows:

```
OPINFO_TABLE: REF OPINFO$TABLE
```

```
!
! Define the fields of the Operator Information Table entry.  Also define the
! declaration macro.
!
FIELD OPINFO$FLD_DEF =
        SET
        OPINFO$L_ROUTTBL    = [ 0, L_ ], ! Address of Operator Routine Table
        OPINFO$L_HIERTBL    = [ 1, L_ ], ! Address of Type Hierarchy Table
        OPINFO$L_INCOMPTBL  = [ 2, L_ ], ! Address of Type Incompatibility Table
        OPINFO$V_FETCH      = [ 3,V_(0)] ! Flag for whether fetch is done on
                                         !     this operation.
        TES;

MACRO
    OPINFO$TABLE = BLOCKVECTOR[TOKEN$K_MAX_OPERATOR + 1,4,LONG]
        FIELD(OPINFO$FLD_DEF) %;
```

OPERATOR ROUTINE TABLE

The Operator Routine Table for a given operator contains one entry for each routine which can be associated with that operator. Each such entry contains the operand types accepted by that routine (e.g., REAL,REAL for F_Floating multiply), the result type of the operation, a routine index which identifies the routine to be invoked, and an optional typeid check routine index which identifies the typeid check routine to be invoked.



A pointer to an operator routine table entry is declared as follows:

```
ORTPTR : REF ORT$ENTRY
```

```
! Define the fields of the Operator Routine Table entry.  Also define a
! declaration macro.
!
FIELD ORT$FLD_DEF =
        SET
        ORT$A_BASE_ADDR         = [ 0, A_  ],   ! Base address
        ORT$B_RIGHT_TYPE        = [ 0, B0_ ],   ! The right operand Dtype
        ORT$B_LEFT_TYPE         = [ 0, B1_ ],   ! The left operand Dtype
        ORT$W_TYPES             = [ 0, W0_ ],   ! The operand Dtype
        ORT$W_ROUT              = [ 0, W1_ ],   ! The corresponding routine index
        ORT$W_TYPEID_ROUT       = [ 1, W0_ ],   ! The corresponding type id check
                                                !       routine index
        ORT$B_RESULT_TYPE       = [ 1, B2_]     ! Result Dtype
        TES;

MACRO
    ORT$ENTRY = BLOCK[2, LONG] FIELD(ORT$FLD_DEF) %;

MACRO
    ORT$TABLE = BLOCKVECTOR[,2,LONG] FIELD(ORT$FLD_DEF) %;

! This is used to encode "unknown result type" in the dtype field of
! the result value descriptor constructed in DBGEVALOP.
!
LITERAL
        ORT$K_RESULT_UNKNOWN    = 0;
```

```
! Literals that are used in the new style operator evaluation.  These
! values are stored in the ORT$W_ROUT field of the Operator Routine
! Table entry for a given operator operating on a given pair of
! types.  The literal is used as a case index in the DBG$PERFORM_OPERATOR
! routine in order to actually do operations.
!
!****************************************************************************
! Note - the order of the case indices in DBGPERMOP must correspond
! exactly to the ordering of the literals here. One must be very careful
! in changing this list.
!
! This situation is of course a maintenance problem and we should eventually
! figure out a better way to get these literals shared.
!
! In the meantime, the list below must be kept dense and must correspond
! exactly to the case labels in DBGPERMOP.
!****************************************************************************

!***
LITERAL
        ORT$K_MIN_ROUT          = 1,

        ! Add.
        !
        ORT$K_ADD_B_B           = 1,
        ORT$K_ADD_BU_BU         = 2,
        ORT$K_ADD_W_W           = 3,
        ORT$K_ADD_WU_WU         = 4,
        ORT$K_ADD_L_L           = 5,
        ORT$K_ADD_LU_LU         = 6,
        ORT$K_ADD_F_F           = 7,
        ORT$K_ADD_D_D           = 8,
        ORT$K_ADD_G_G           = 9,
        ORT$K_ADD_H_H           = 10,
        ORT$K_ADD_FC_FC         = 11,
        ORT$K_ADD_DC_DC         = 12,
        ORT$K_ADD_GC_GC         = 13,
        ORT$K_ADD_HC_HC         = 14,

        ! Subtract.
        !
        ORT$K_SUB_B_B           = 15,
        ORT$K_SUB_BU_BU         = 16,
        ORT$K_SUB_W_W           = 17,
        ORT$K_SUB_WU_WU         = 18,
        ORT$K_SUB_L_L           = 19,
        ORT$K_SUB_LU_LU         = 20,
        ORT$K_SUB_F_F           = 21,
        ORT$K_SUB_D_D           = 22,
        ORT$K_SUB_G_G           = 23,
        ORT$K_SUB_H_H           = 24,
        ORT$K_SUB_FC_FC         = 25,
        ORT$K_SUB_DC_DC         = 26,
        ORT$K_SUB_GC_GC         = 27,
        ORT$K_SUB_HC_HC         = 28,
```

```
        ! Divide.
        !
        ORT$K_DIV_B_B           = 29,
        ORT$K_DIV_BO_BU         = 30,
        ORT$K_DIV_W_W           = 31,
        ORT$K_DIV_WO_WU         = 32,
        ORT$K_DIV_L_L           = 33,
        ORT$K_DIV_LO_LU         = 34,
        ORT$K_DIV_F_F           = 35,
        ORT$K_DIV_D_D           = 36,
        ORT$K_DIV_G_G           = 37,
        ORT$K_DIV_H_H           = 38,
        ORT$K_DIV_FC_FC         = 39,
        ORT$K_DIV_DC_DC         = 40,
        ORT$K_DIV_GC_GC         = 41,
        ORT$K_DIV_HC_HC         = 42,

        ! Multiply
        !
        ORT$K_MUL_B_B           = 43,
        ORT$K_MUL_BO_BU         = 44,
        ORT$K_MUL_W_W           = 45,
        ORT$K_MUL_WO_WU         = 46,
        ORT$K_MUL_L_L           = 47,
        ORT$K_MUL_LO_LU         = 48,
        ORT$K_MUL_F_F           = 49,
        ORT$K_MUL_D_D           = 50,
        ORT$K_MUL_G_G           = 51,
        ORT$K_MUL_H_H           = 52,
        ORT$K_MUL_FC_FC         = 53,
        ORT$K_MUL_DC_DC         = 54,
        ORT$K_MUL_GC_GC         = 55,
        ORT$K_MUL_HC_HC         = 56,

        ! Remainder.
        !
        ORT$K_MOD_L_L           = 57,
        ORT$K_MOD_LO_LU         = 58,
        ORT$K_REM_L_L           = 59,
        ORT$K_REM_LO_LU         = 60,

        ! Shift operations.
        !
        ORT$K_SHIFT_LEFT_L_L    = 61,
        ORT$K_SHIFT_RT_L_L      = 62,
        ORT$K_SHIFT_RT_LO_LU    = 63,

        ! Exponentiation.
        !
        ORT$K_POWER_W_W         = 64,
        ORT$K_POWER_L_L         = 65,
        ORT$K_POWER_F_L         = 66,
        ORT$K_POWER_D_L         = 67,
        ORT$K_POWER_G_L         = 68,
        ORT$K_POWER_H_L         = 69,
        ORT$K_POWER_FC_L        = 70,
```

```
        ORT$K_POWER_DC_L        = 71,
        ORT$K_POWER_GC_L        = 72,
        ORT$K_POWER_F_F         = 73,
        ORT$K_POWER_D_F         = 74,
        ORT$K_POWER_F_D         = 75,
        ORT$K_POWER_D_D         = 76,
        ORT$K_POWER_G_G         = 77,
        ORT$K_POWER_H_H         = 78,
        ORT$K_POWER_FC_FC       = 79,
        ORT$K_POWER_DC_DC       = 80,
        ORT$K_POWER_GC_GC       = 81,

        ! String operations.
        !
        ORT$K_CONCAT_T_T        = 82,
        ORT$K_EQL_VT_VT         = 83,
        ORT$K_EQL_T_T           = 84,
        ORT$K_GEQ_VT_VT         = 85,
        ORT$K_GEQ_T_T           = 86,
        ORT$K_GTR_VT_VT         = 87,
        ORT$K_GTR_T_T           = 88,
        ORT$K_LEQ_VT_VT         = 89,
        ORT$K_LEQ_T_T           = 90,
        ORT$K_LSS_VT_VT         = 91,
        ORT$K_LSS_T_T           = 92,
        ORT$K_NEQ_VT_VT         = 93,
        ORT$K_NEQ_T_T           = 94,

        ! Relationals.
        !
        ORT$K_EQL_B_B           = 95,
        ORT$K_EQL_W_W           = 96,
        ORT$K_EQL_L_L           = 97,
        ORT$K_EQL_F_F           = 98,
        ORT$K_EQL_D_D           = 99,
        ORT$K_EQL_G_G           = 100,
        ORT$K_EQL_H_H           = 101,
        ORT$K_EQL_FC_FC         = 102,
        ORT$K_EQL_DC_DC         = 103,
        ORT$K_EQL_GC_GC         = 104,
        ORT$K_EQL_HC_HC         = 105,

        ORT$K_NEQ_B_B           = 106,
        ORT$K_NEQ_W_W           = 107,
        ORT$K_NEQ_L_L           = 108,
        ORT$K_NEQ_F_F           = 109,
        ORT$K_NEQ_D_D           = 110,
        ORT$K_NEQ_G_G           = 111,
        ORT$K_NEQ_H_H           = 112,
        ORT$K_NEQ_FC_FC         = 113,
        ORT$K_NEQ_DC_DC         = 114,
        ORT$K_NEQ_GC_GC         = 115,
        ORT$K_NEQ_HC_HC         = 116,

        ORT$K_GEQ_B_B           = 117,
        ORT$K_GEQ_W_W           = 118,
```

```
        ORT$K_GEQ_L_L           = 119,
        ORT$K_GEQ_LO_LU         = 120,
        ORT$K_GEQ_F_F           = 121,
        ORT$K_GEQ_D_D           = 122,
        ORT$K_GEQ_G_G           = 123,
        ORT$K_GEQ_H_H           = 124,

        ORT$K_GTR_B_B           = 125,
        ORT$K_GTR_W_W           = 126,
        ORT$K_GTR_L_L           = 127,
        ORT$K_GTR_LO_LU         = 128,
        ORT$K_GTR_F_F           = 129,
        ORT$K_GTR_D_D           = 130,
        ORT$K_GTR_G_G           = 131,
        ORT$K_GTR_H_H           = 132,

        ORT$K_LEQ_B_B           = 133,
        ORT$K_LEQ_W_W           = 134,
        ORT$K_LEQ_L_L           = 135,
        ORT$K_LEQ_LO_LU         = 136,
        ORT$K_LEQ_F_F           = 137,
        ORT$K_LEQ_D_D           = 138,
        ORT$K_LEQ_G_G           = 139,
        ORT$K_LEQ_H_H           = 140,

        ORT$K_LSS_B_B           = 141,
        ORT$K_LSS_W_W           = 142,
        ORT$K_LSS_L_L           = 143,
        ORT$K_LSS_LO_LU         = 144,
        ORT$K_LSS_F_F           = 145,
        ORT$K_LSS_D_D           = 146,
        ORT$K_LSS_G_G           = 147,
        ORT$K_LSS_H_H           = 148,

        ! Bitwise logical operations.
        !
        ORT$K_BIT_AND_B_B       = 149,
        ORT$K_BIT_AND_W_W       = 150,
        ORT$K_BIT_AND_L_L       = 151,
        ORT$K_BIT_EQV_B_B       = 152,
        ORT$K_BIT_EQV_W_W       = 153,
        ORT$K_BIT_EQV_L_L       = 154,
        ORT$K_BIT_NOT_B         = 155,
        ORT$K_BIT_NOT_W         = 156,
        ORT$K_BIT_NOT_L         = 157,
        ORT$K_BIT_OR_B_B        = 158,
        ORT$K_BIT_OR_W_W        = 159,
        ORT$K_BIT_OR_L_L        = 160,
        ORT$K_BIT_XOR_B_B       = 161,
        ORT$K_BIT_XOR_W_W       = 162,
        ORT$K_BIT_XOR_L_L       = 163,

        ! Logical operations.
        !
        ORT$K_AND_B_B           = 164,
        ORT$K_AND_W_W           = 165,
```

```
        ORT$K_AND_L_L           = 166,
        ORT$K_AND_D_D           = 167,
        ORT$K_NOT_B             = 168,
        ORT$K_NOT_W             = 169,
        ORT$K_NOT_L             = 170,
        ORT$K_NOT_D             = 171,
        ORT$K_OR_B_B            = 172,
        ORT$K_OR_W_W            = 173,
        ORT$K_OR_L_L            = 174,
        ORT$K_OR_D_D            = 175,
        ORT$K_XOR_L_L           = 176,

        ! Unary plus and minus.
        !
        ORT$K_UNARY_MINUS_B     = 177,
        ORT$K_UNARY_MINUS_W     = 178,
        ORT$K_UNARY_MINUS_L     = 179,
        ORT$K_UNARY_MINUS_LU    = 180,
        ORT$K_UNARY_MINUS_F     = 181,
        ORT$K_UNARY_MINUS_D     = 182,
        ORT$K_UNARY_MINUS_G     = 183,
        ORT$K_UNARY_MINUS_H     = 184,
        ORT$K_UNARY_MINUS_FC    = 185,
        ORT$K_UNARY_MINUS_DC    = 186,
        ORT$K_UNARY_MINUS_GC    = 187,
        ORT$K_UNARY_MINUS_HC    = 188,
        ORT$K_UNARY_PLUS_B      = 189,
        ORT$K_UNARY_PLUS_W      = 190,
        ORT$K_UNARY_PLUS_L      = 191,
        ORT$K_UNARY_PLUS_F      = 192,
        ORT$K_UNARY_PLUS_D      = 193,
        ORT$K_UNARY_PLUS_G      = 194,
        ORT$K_UNARY_PLUS_H      = 195,
        ORT$K_UNARY_PLUS_FC     = 196,
        ORT$K_UNARY_PLUS_DC     = 197,
        ORT$K_UNARY_PLUS_GC     = 198,
        ORT$K_UNARY_PLUS_HC     = 199,

        ! Absolute Value.
        !
        ORT$K_ABS_B             = 200,
        ORT$K_ABS_W             = 201,
        ORT$K_ABS_L             = 202,
        ORT$K_ABS_F             = 203,
        ORT$K_ABS_D             = 204,
        ORT$K_ABS_G             = 205,
        ORT$K_ABS_H             = 206,

        ! Indirection.
        !
        ORT$K_INDIRECT_LU       = 207,
        ORT$K_INDIRECT_TPTR     = 208,

        ! BLISS bit-select.
        !
        ORT$K_BITSELECT         = 209,
```

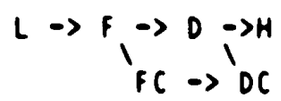
```
        ! Set operations.
        !
        ORT$K_DIFFERENCE_SET_SET = 210,
        ORT$K_EQL_SET_SET        = 211,
        ORT$K_GEQ_SET_SET        = 212,
        ORT$K_IN_SET_SET         = 213,
        ORT$K_INTERSECT_SET_SET  = 214,
        ORT$K_LEQ_SET_SET        = 215,
        ORT$K_NEQ_SET_SET        = 216,
        ORT$K_UNION_SET_SET      = 217,

        ! C operations:
        ! ADDRESS_OF (&), SIZEOF
        ! Also, addition or subtraction involving typed pointers
        ! is special-cased in C because there is scaling to be done.
        !
        ORT$K_ADDRESS_L          = 218,
        ORT$K_SIZEOF_L           = 219,
        ORT$K_ADD_TPTR_L         = 220,
        ORT$K_SUB_TPTR_L         = 221,
        ORT$K_SUB_TPTR_TPTR      = 222,

        ! BASIC implication (A imp B)
        !
        ORT$K_BIT_IMP_B_B        = 223,
        ORT$K_BIT_IMP_W_W        = 224,
        ORT$K_BIT_IMP_L_L        = 225,

        ! C pre- and post- increment, decrement (++X  X++  --X  X--)
        !
        ORT$K_PRE_INCR_L         = 226,
        ORT$K_PRE_INCR_LU        = 227,
        ORT$K_PRE_INCR_D         = 228,
        ORT$K_PRE_INCR_TPTR      = 229,
        ORT$K_POST_INCR_L        = 230,
        ORT$K_POST_INCR_LU       = 231,
        ORT$K_POST_INCR_D        = 232,
        ORT$K_POST_INCR_TPTR     = 233,
        ORT$K_PRE_DECR_L         = 234,
        ORT$K_PRE_DECR_LU        = 235,
        ORT$K_PRE_DECR_D         = 236,
        ORT$K_PRE_DECR_TPTR      = 237,
        ORT$K_POST_DECR_L        = 238,
        ORT$K_POST_DECR_LU       = 239,
        ORT$K_POST_DECR_D        = 240,
        ORT$K_POST_DECR_TPTR     = 241,

        ! Packed Decimal Operations.
        !
        ORT$K_ADD_P_P            = 242,
        ORT$K_SUB_P_P            = 243,
        ORT$K_MUL_P_P            = 244,
        ORT$K_DIV_P_P            = 245,
        ORT$K_UNARY_PLUS_P       = 246,
        ORT$K_UNARY_MINUS_P      = 247,
```

```
        ORT$K_EQL_P_P           = 248,
        ORT$K_NEQ_P_P           = 249,
        ORT$K_GTR_P_P           = 250,
        ORT$K_GEQ_P_P           = 251,
        ORT$K_LSS_P_P           = 252,
        ORT$K_LEQ_P_P           = 253,

        ! PL/I Bit-String Operations.
        !
        ORT$K_CONCAT_TF_TF      = 254,
        ORT$K_EQL_TF_TF         = 255,
        ORT$K_NEQ_TF_TF         = 256,
        ORT$K_GTR_TF_TF         = 257,
        ORT$K_GEQ_TF_TF         = 258,
        ORT$K_LSS_TF_TF         = 259,
        ORT$K_LEQ_TF_TF         = 260,
        ORT$K_BIT_NOT_TF        = 261,
        ORT$K_BIT_AND_TF        = 262,
        ORT$K_BIT_OR_TF         = 263,

        ORT$K_EQL_RFA_RFA       = 264,
        ORT$K_NEQ_RFA_RFA       = 265,

        ORT$K_UNARY_PLUS_Q      = 266,
        ORT$K_UNARY_MINUS_Q     = 267,
        ORT$K_UNARY_PLUS_O      = 268,
        ORT$K_UNARY_MINUS_O     = 269,

        ! Built-in Functions
        !
        ORT$K_SUCC_ENUM         = 270,
        ORT$K_PRED_ENUM         = 271,

        ! Fixed binary operations.
        !
        ORT$K_UNARY_PLUS_FIXED  = 272,
        ORT$K_UNARY_MINUS_FIXED = 273,
        ORT$K_ABS_FIXED         = 274,
        ORT$K_ADD_FIXED_FIXED   = 275,
        ORT$K_SUB_FIXED_FIXED   = 276,
        ORT$K_MUL_FIXED_FIXED   = 277,
        ORT$K_DIV_FIXED_FIXED   = 278,
        ORT$K_EQL_FIXED_FIXED   = 279,
        ORT$K_NEQ_FIXED_FIXED   = 280,
        ORT$K_LSS_FIXED_FIXED   = 281,
        ORT$K_GTR_FIXED_FIXED   = 282,
        ORT$K_LEQ_FIXED_FIXED   = 283,
        ORT$K_GEQ_FIXED_FIXED   = 284,
        ORT$K_MAX_ROUT          = 284;  !*** ALSO CHANGE the corresponding
                                        !*** definition in DBGPERMOP.MAR if
                                        !*** you change this.

! Literals that are defined for type id check in DBG$PERFORM_TYPEID_CHECK.
!
LITERAL
```

```
        ORT$K_TYPEID_MIN_ROUT    = 1,
        ORT$K_TYPEID_ENUM_ENUM   = 1,
        ORT$K_TYPEID_SET_SET     = 2,
        ORT$K_TYPEID_TPTR_TPTR   = 3,
        ORT$K_TYPEID_SUBRNG_SUBRNG = 4,
        ORT$K_TYPEID_MAX_ROUT    = 4;
!**-
```

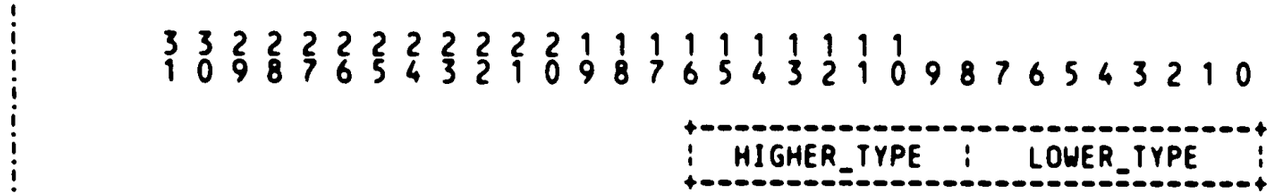
TYPE GRAPH

The type conversions that are done for a given operator in a given language are specified by a set of directed acyclic graphs. For example, part of the graph that specifies the implicit type conversions to be done for FORTRAN addition might look like:



Each graph is represented by listing its edges. Each edge is just a pair of Type IDs.

Below we define the data structure used to declare the edges of these graphs.



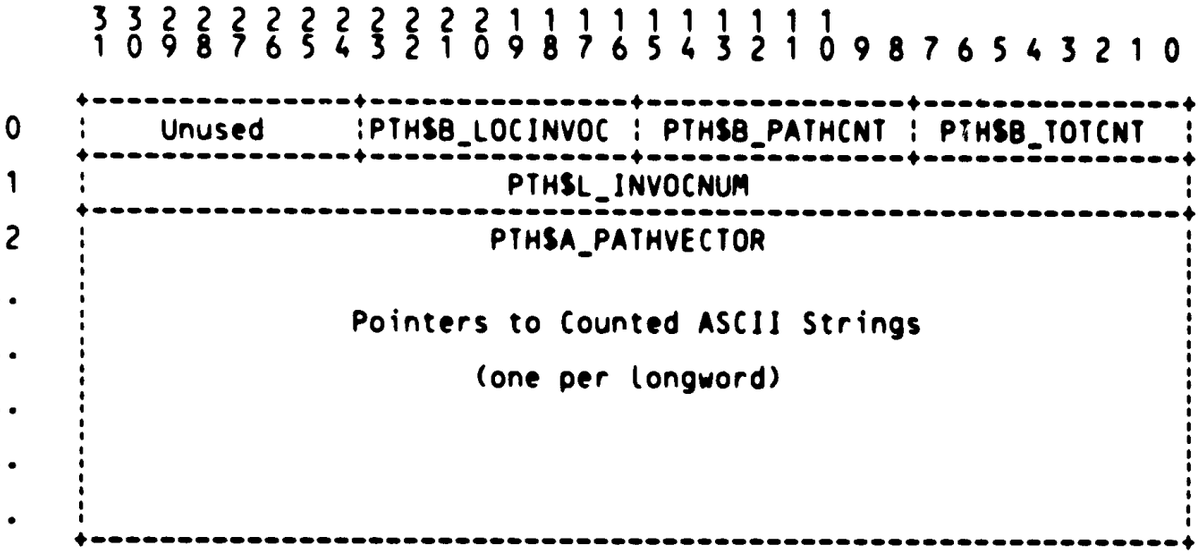

```
! Define the fields in an entry in an edge of a Type Graph.
!
FIELD TYPE_GRAPH$FLD_DEF =
        SET
        TYPE_GRAPH$B_LOWER_TYPE  = [0,B0_],
        TYPE_GRAPH$B_HIGHER_TYPE = [0,B1_],
        TYPE_GRAPH$W_BOTH_TYPES  = [0,W0_]
        TES;

! Define the macro which is used to declare an entry in a Type Graph
!
MACRO
    TYPE_GRAPH$ENTRY = BLOCK [1,WORD] FIELD (TYPE_GRAPH$FLD_DEF) %;
```

```
!
!       T Y P E   C O N V E R S I O N   I N F O R M A T I O N   T A B L E
!
!       The Type Conversion Information Table is a vector of longwords used
!       by a language which gives pointers to tables which contain the
!       information necessary to select the type conversions and then
!       the routine invocations which does the type conversion.
!
!        3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
!        1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
!       +---------------------------------------------------------------+
!  0    !                       CVTINFO$L_MAPTBL                        !
!       +---------------------------------------------------------------+
!  1    !                       CVTINFO$L_CVTTBL                        !
!       +---------------------------------------------------------------+
!  2    !                       flags                                   !
!       +---------------------------------------------------------------+
!
!       A pointer to an Type Conversion Information Table is declared as
!       follows:
!
!                       CVTINFO_TABLE: REF CVTINFO$TABLE
!
! Define the fields of the Type Conversion Information Table entry.  Also
! define the declaration macro.
!
FIELD CVTINFO$FLD_DEF =
        SET
        CVTINFO$L_MAPTBL    = [ 0, L_ ], ! Address of Type Map Table
        CVTINFO$L_CVTTBL    = [ 1, L_ ], ! Address of Type Conversion Table
        CVTINFO$V_ROUND     = [ 2, V_(0)]! Flag to indicate conversion result
                                         !       is rounded
        TES;

MACRO
    CVTINFO$TABLE = BLOCK[3,LONG] FIELD (CVTINFO$FLD_DEF) %;
```

```
!       TYPE CONVERT ENTRY


!       A Type Convert Entry contains a type conversion
!       routine index which identifies the routine to be invoked, from data
!       type, and to data type.

!       3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
!       1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
!       +---------------------------------------------------------------+
!       !       TYPE_CVT$W_ROUT         !  HIGHER_TYPE  !   LOWER_TYPE   !
!       +---------------------------------------------------------------+


!       A pointer to an Type Map Entry is declared as follows:

!                       CVTPTR : REF TYPE_CVT$ENTRY

! Define the fields of the Type Map Table entry.  Also define an entry
! declaration macro.

FIELD
        TYPE_CVT$FLD_DEF =
            SET
            TYPE_CVT$B_LOWER_TYPE  = [0, B0_],
            TYPE_CVT$B_HIGHER_TYPE = [0, B1_],
            TYPE_CVT$W_MAP_PAIR    = [0, W0_],
            TYPE_CVT$W_ROUT        = [0, W1_]
            TES;

MACRO
        TYPE_CVT$ENTRY = BLOCK[1, LONG] FIELD (TYPE_CVT$FLD_DEF) %;


! Literals that are used in the new style operator evaluation.
! The literal is used as a case index in the DBG$LANGUAGE_TYPE_CONV
! routine in order to actually do the conversion.
!
!***
LITERAL
        CVT$K_MIN_ROUT          = 1,
        CVT$K_PLI_CVT           = 1,
        CVT$K_COB_PICT          = 2,
        CVT$K_MAX_ROUT          = 2;
!**-
```

P A T H N A M E D E S C R I P T O R S

The pathname descriptor is used internally in the Debugger to represent a pathname, including data qualification. Thus "MOD\ROUT\X" is a pathname in this sense, and so is "MOD\ROUT\A.B.C". The format of a pathname descriptor is shown here:



The Pathname Descriptor contains two counts fields which give the total number of names in the pathname (five for M\R\A.B.C) and the total number of names before the first dot (three for M\R\A.B.C). It also contains an invocation number (which is zero if none was explicitly specified) and an index into PTH$A_PATHVECTOR which indicates which pathname component was qualified by the invocation number (1 for A 2\X and 4 for M\R1\R2\R3 5\X). This index is zero if no invocation number was given. The rest of the descriptor consists of longwords containing pointers to the individual names in the pathname, each represented by a Counted ASCII String.

A pointer to a Pathname Descriptor is declared as follows:

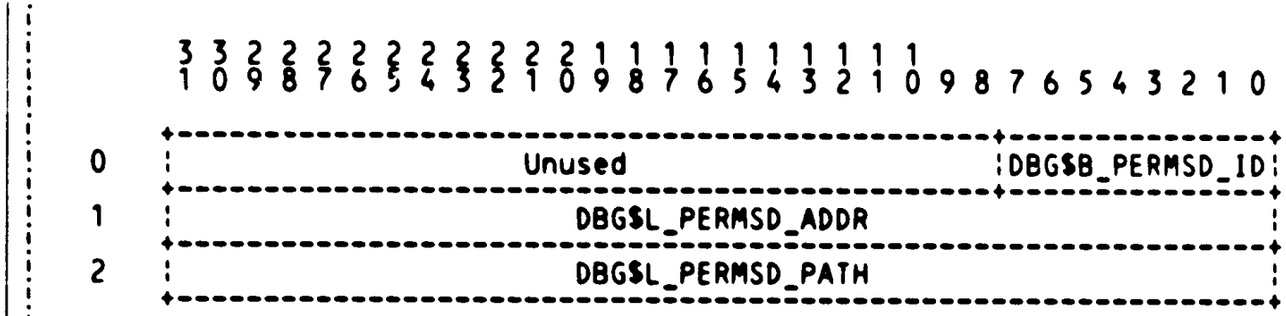
```
PATHPTR: REF PTH$PATHNAME
```

```
!***

! Define the Pathname Descriptor fields.

FIELD PTH$FLD_DEF =
        SET
        PTH$B_TOTCNT    = [ 0, B0_ ],   ! Total number of names in pathname
```

```
                                        !       including all data qualification
        PTH$B_PATHCNT    = [ 0, B1_ ],  ! Numuer of names in pathname proper
                                        !       before any data qualification
        PTH$B_LOCINVOC   = [ 0, B2_ ],  ! The location in the vector below of
                                        !       the name with an invocation num-
                                        !       ber (first name is 1) or zero
        ! ---            = [ 0, B3_ ],  ! Unused
        PTH$L_INVOCNUM   = [ 1, L_ ],   ! The invocation number or zero
        PTH$A_PATHVECTOR= [ 2, A_ ]     ! The start of the pathname vector--has
                                        !       one name pointer (pointing to a
                                        !       Counted ASCII name) per longword.
        TES;

! Define the maximum allowed size of a pathname, i.e. the maximum number of
! individual names, including data components.  Also define the size of the
! fixed part of the descriptor and the declaration macro.
!
LITERAL
        DBG$K_MAX_PATHNAME = 50,        ! Maximum size of a pathname
        DBG$K_PATHDESCSIZE = 2,         ! Size of fixed part of Pathname
                                        !       Descriptor
        DBG$K_PATHNAME_SIZE = DBG$K_MAX_PATHNAME + DBG$K_PATHDESCSIZE;

MACRO
        PTH$PATHNAME = BLOCK[DBG$K_PATHNAME_SIZE] FIELD(PTH$FLD_DEF) %;
!***-
```

P E R M A N E N T S Y M B O L D E S C R I P T O R S

The Permanent Symbol Descriptor is used to describe Debugger "permanent symbols" in Value Descriptors, Primary Descriptors, and address expressions. Permanent symbols name the VAX registers; "R0" is thus a permanent symbol. Permanent Symbol Descriptors are only used for languages COBOL and PL/I. A Permanent Symbol Descriptor has this format:



A pointer to a Permanent Symbol Descriptor is declared as follows:

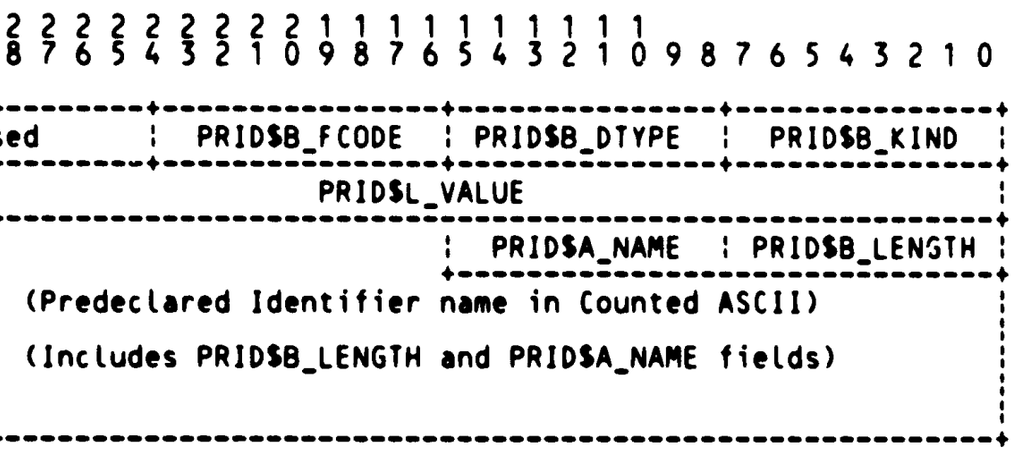
```
PSDPTR: REF DBG$PERMSD
```

```
!***
! Define the Permanent Symbol Descriptor fields and declaration macro.
!
FIELD DBG$PERMSD_FIELDS =
        SET
        DBG$B_PERMSD_ID   = [ 0, B0_ ], ! Symbol ID (register number)
        DBG$L_PERMSD_ADDR = [ 1, L_ ],  ! Address of the register contents
        DBG$L_PERMSD_PATH = [ 2, L_ ]   ! Address of symbol's pathname descr.
        TES;

LITERAL
        DBG$K_PERMSD_SIZE = 3;          ! Size of Permanent Symbol Descriptor
                                        !      in longwords

MACRO
        DBG$PERMSD = BLOCK[DBG$K_PERMSD_SIZE] FIELD(DBG$PERMSD_FIELDS) %;

! These are the possible values of the DBG$B_PERMSD_ID field.
!
LITERAL
        DBG$K_R0        = 200,          ! Register R0
        DBG$K_R1        = 201,          ! Register R1
        DBG$K_R2        = 202,          ! Register R2
        DBG$K_R3        = 203,          ! Register R3
        DBG$K_R4        = 204,          ! Register R4
        DBG$K_R5        = 205,          ! Register R5
```

```
        DBG$K_R6        = 206,          ! Register R6
        DBG$K_R7        = 207,          ! Register R7
        DBG$K_R8        = 208,          ! Register R8
        DBG$K_R9        = 209,          ! Register R9
        DBG$K_R10       = 210,          ! Register R10
        DBG$K_R11       = 211,          ! Register R11
        DBG$K_AP        = 212,          ! Argument Pointer (AP)
        DBG$K_FP        = 213,          ! Frame Pointer (FP)
        DBG$K_SP        = 214,          ! Stack Pointer (SP)
        DBG$K_PC        = 215,          ! Program Counter (PC)
        DBG$K_PSL       = 216;          ! Processor Status Longword (PSL)
!**-
```

P R E D E C L A R E D I D E N T I F I E R E N T R Y

Language reserves Predeclared Identifier as names of functions, procedures, types, values, etc. Predeclared Identifier Entry is used by DEBUG to describe one of these predeclared identifiers. An identifer is looked up in this table, if an entry is found, a value descriptor is created for this entry. For example, TRUE, FALSE in PASCAL, .TRUE., .FALSE., in FORTRAN. The Predeclared Identifier Entry has the following format:



The PRID$B_KIND field defines the kind of Predefined Identifier. The PRID$B_KIND field of an Predefined Identifier Constant Entry always has the value of PRID$K_CONSTANT. It has the following format.

A pointer to a Predefined Identifier Value Entry is defined as follows:

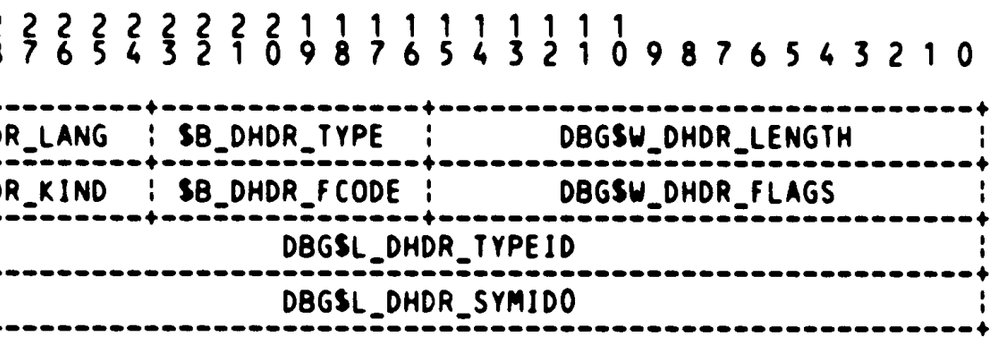
```
PRIDPTR: REF PRID$ENTRY
```

```
! Define the fields of the Predefined Identifier Entry.

FIELD PRID$FLD_DEF =
        SET
        PRID$B_KIND     = [ 0, B0_],    ! Predefined Identifier Kind
        PRID$B_DTYPE    = [ 0, B1_],    ! Data Type of the Predefined ID constant
        PRID$B_FCODE    = [ 0, B2_],    ! FCODE of the data type
        PRID$L_VALUE    = [ 1, L_],     ! Data value of the Predefined ID constant
        PRID$B_LENGTH   = [ 2, B0_],    ! Length of the Predefined ID name in butes
        PRID$A_NAME     = [ 2, A1_]     ! Predefined ID name in ASCII
        TES;

LITERAL
        PRID$K_ENTSIZE = 2;             ! Size of fixed part is 2 longwords

MACRO
```

```
        PRID$ENTRY = BLOCK[,LONG] FIELD(PRID$FLD_DEF) %;
! Define the valid values of the PRID$B_KIND field.
!
LITERAL
        PRID$K_CONSTANT = 1;            ! Predeclared ID constant
```

P R I M A R Y A N D V A L U E D E S C R I P T O R H E A D E R

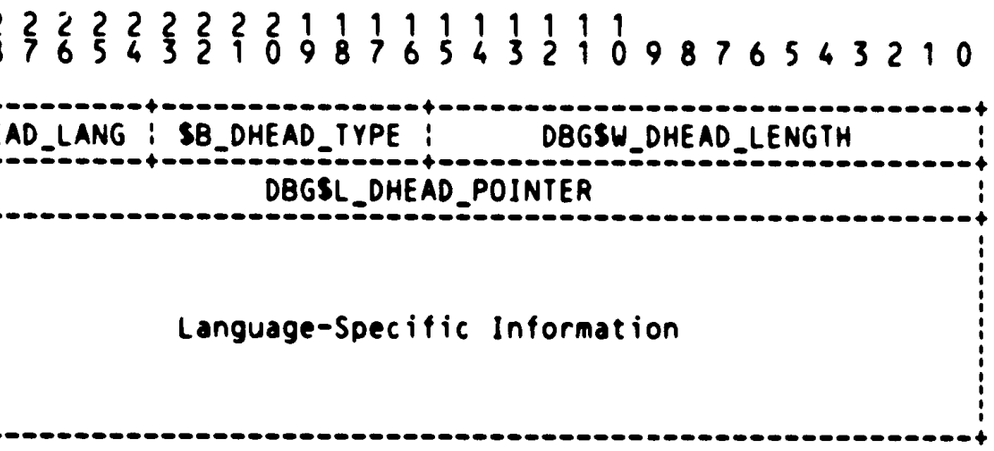
Primary Descriptors and Value Descriptors are descriptors built by DEBUG to describe Primary Symbols and language values, respectively. The language-independent versions of these descriptors all have a common header section which has the following format:




```
! Define the Primary or Value Descriptor Header Block fields.

FIELD DBG$DHDR_FIELDS =
        SET
        DBG$B_DHDR_LANG      = [ 0, B3_ ],       ! Language Code
        DBG$B_DHDR_TYPE      = [ 0, B2_ ],       ! Type Code (Primary/Value/...)
        DBG$W_DHDR_LENGTH    = [ 0, W0_ ],       ! Length of descriptor in bytes
        DBG$B_DHDR_KIND      = [ 1, B3_ ],       ! RST kind of entity
        DBG$B_DHDR_FCODE     = [ 1, B2_ ],       ! FCODE of data item
        DBG$W_DHDR_FLAGS     = [ 1, W0_ ],       ! Flags word
          DBG$V_DHDR_AGGR      = [ 1, V0_(0)],   ! Set if data item is an aggregate
          DBG$V_DHDR_SUBREF    = [ 1, V0_(1)],   ! Set if partial data reference
          DBG$V_DHDR_BITREF    = [ 1, V0_(2)],   ! Set if bit-string sub-reference
          DBG$V_DHDR_SGNEXT    = [ 1, V0_(3)],   ! Set if value should be sign-extended
          DBG$V_DHDR_BLIBLK    = [ 1, V0_(4)],   ! Set if data item is a BLISS block
          DBG$V_DHDR_UNCVT     = [ 1, V0_(5)],   ! Unconverted constant flag
          DBG$V_DHDR_LITERAL   = [ 1, V0_(6)],   ! Flag for literal value
          DBG$V_DHDR_OVERRIDE  = [ 1, V0_(7)],   ! Set if override was present
          DBG$V_DHDR_TMPREF    = [ 1, V1_(0)],   ! Set if temporary SUBREF
          DBG$V_DHDR_FORMAT    = [ 1, V1_(4,4)], ! Printing Format Code
        DBG$L_DHDR_TYPEID    = [ 2, L_  ],       ! TYPEID of data item
        DBG$L_DHDR_SYMIDO    = [ 3, L_  ]        ! Context Symid (root data item)
        TES;
```

PRIMARY AND VALUE DESCRIPTOR HEADER FOR COBOL AND PL/I

The Primary Descriptor and the Value Descriptor for COBOL and PL/I are descriptor blocks built by the language-specific routines for those two languages. As such their contents are unknown to the Debugger kernel (and are not described here), but they all must have the common header described here. A Primary Descriptor or Value Descriptor for COBOL or PL/I thus has this format:



A pointer to a descriptor header block is declared as follows:

```
DESCPTR: REF DBG$DHEAD
```

```
!***
! Define the Value or Primary Descriptor Header Block fields and the declaration
! macro.

FIELD DBG$DHEAD_FIELDS =
        SET
        DBG$W_DHEAD_LENGTH = [ 0, W0_ ],! Length of descriptor in bytes
                                        !       including this header
        DBG$B_DHEAD_TYPE   = [ 0, B2_ ],! The type of this descriptor
        DBG$B_DHEAD_LANG   = [ 0, B3_ ],! Language of the descriptor
        DBG$L_DHEAD_POINTER= [ 1, L_ ]  ! Unused at present
        TES;

LITERAL
        DBG$K_DHEAD_SIZE = 2;           ! Size of descriptor header in longwords

MACRO
        DBG$DHEAD = BLOCK[DBG$K_DHEAD_SIZE] FIELD(DBG$DHEAD_FIELDS) %;
```

```
! The possible values of the DBG$B_DHEAD_LANG field are the language codes
! defined in the section entitled "Miscellaneous Literals" above.  The possible
! value of the DBG$B_DHEAD_TYPE field, which are defined in the same section,
! are the following:
!
!       DBG$K_PRIMARY_DESC              ! This is a Primary Descriptor
!       DBG$K_VALUE_DESC                ! This is a non-volatile Value Descriptor
!       DBG$K_V_VALUE_DESC              ! This is a volatile Value Descriptor
!--
```

P R I M A R Y D E S C R I P T O R D E F I N I T I O N S

Primary Descriptors describe Primary Symbols parsed in DEBUG commands. A Primary Symbol may be a simple name, such as "ABC", or a more complex name including pathname qualification, data qualification, subscripting, and dereferencing, such as "MOD\ROUT\A.B[2,3]^.C(4).D".

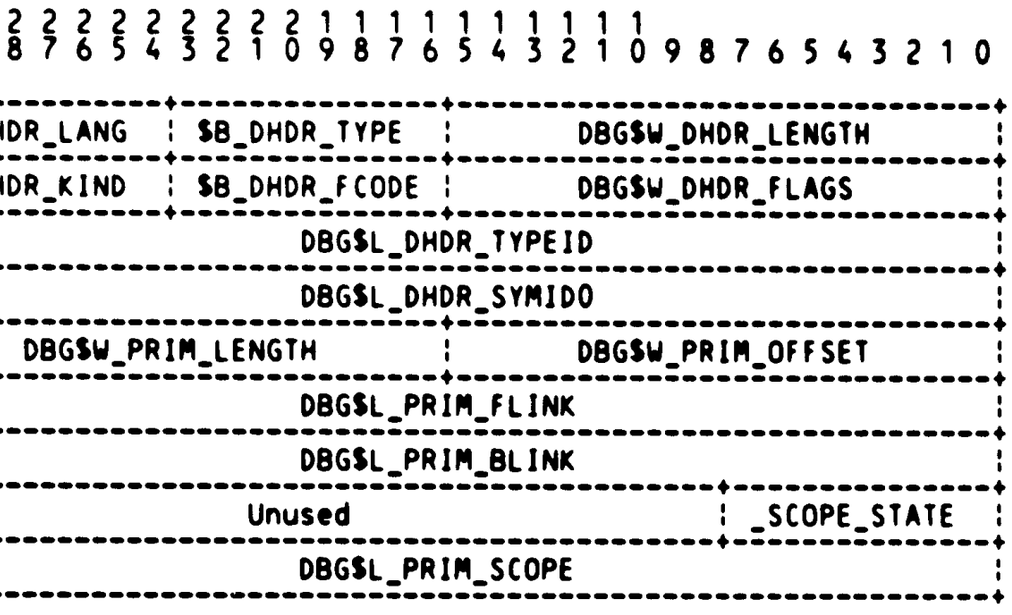
The Primary Descriptor described here is the Primary Descriptor built by the language-independent parser in module DBGPARSER.

A Primary Descriptor consists of a Root Node followed by a doubly linked list of Sub-Nodes which describe the individual components of the symbol described by the Primary Descriptor. Each Sub-Node thus describes an instance of data qualification, subscripting, derefencing, or the like. The last Sub-Node describes the final object named by the Primary Symbol.

Consider the Primary Symbol "A\B\C.D[2]^.E" for example. The Primary Descriptor Root Node would point to a Sub-Node for A\B\C which requires a Record Sub-Node. It points to another Sub-Node for component D of record A\B\C. This component is an array and requires an Array Sub-Node. The next node is the Sub-Node for the array element A\B\C.D[2]. This element is of a pointer type. The next Sub-Node is a node for the pointed-to object which happens to be a record and therefore calls for a Record Sub-Node. Finally, the last Sub-Node on the chain is for component E of that record. The type of Sub-Node required depends on the data type of E, but would be a Normal Sub-Node for all data types other than arrays and records.

PRIMARY DESCRIPTOR ROOT NODE

A Primary Descriptor consists of a Root Node followed by a doubly linked list of Sub-Nodes which describe the individual components of the symbol described by the Primary Descriptor. The format of the Primary Descriptor Root Node is as follows:



A pointer to a Primary Descriptor Root Node is declared as follows:

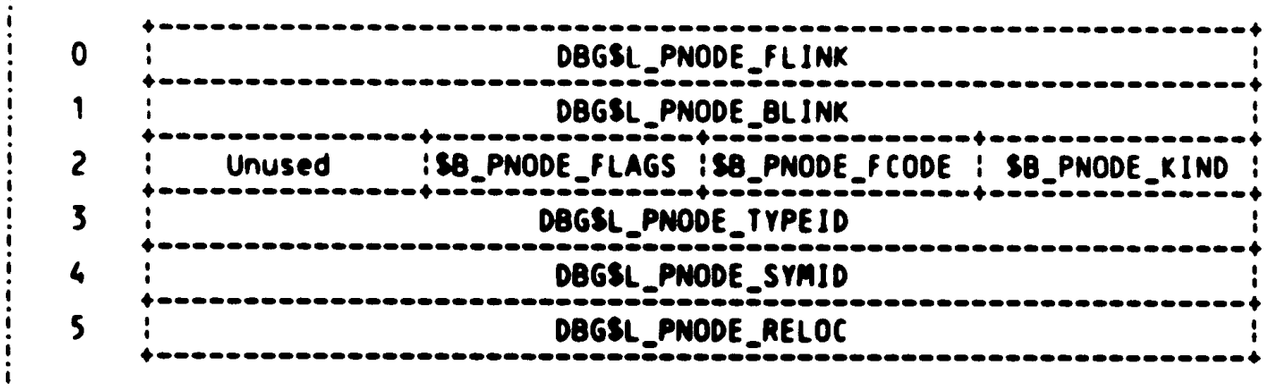
```
PRIMPTR: REF DBG$PRIMARY
```

```
! Define the fields of the language-independent Primary Descriptor Root Node.

FIELD DBG$PRIM_FIELDS =
    SET
    DBG$W_PRIM_OFFSET      = [ 4, SW0_ ],! Offset within data item
    DBG$W_PRIM_LENGTH      = [ 4, W1_ ], ! Length of partial reference
    DBG$A_PRIM_FLINK       = [ 5, A_ ],  ! Address of Sub-Node list head in root
    DBG$L_PRIM_FLINK       = [ 5, L_ ],  ! Forward link to first Sub-Node
    DBG$L_PRIM_BLINK       = [ 6, L_ ],  ! Backward link to last Sub-Node
    DBG$B_PRIM_SCOPE_STATE= [ 7, B0_ ], ! Scope state (see next field)
    DBG$L_PRIM_SCOPE       = [ 8, L_ ]   ! Scope where symbol was looked up
    TES;

! Define the declaration macro.  Note that the Primary Descriptor also has the
! Value Descriptor fields defined.  This is to simplify the declarations in
```

```
! those routines which can accept either a Primary Descriptor or a Value
! Descriptor as input.
!
MACRO
    DBG$PRIMARY = BLOCK [,LONG]
                  FIELD(DBG$DHDR_FIELDS, DBG$PRIM_FIELDS, DBG$VALUE_FIELDS) %;

! Define the size in longwords of the Primary Descriptor Root Node.
!
LITERAL
        DBG$K_PRIMARY_SIZE      = 9;    ! Size of Root Node in longwords
```

PRIMARY DESCRIPTOR NORMAL SUB-NODE

Primary Descriptor Sub-Nodes are attached to the Primary Descriptor Root Node via a doubly linked list. The first Sub-Node represents the first data item in the Primary Symbol (for example, A/B in "A/B.C[2].D") and subsequent nodes represent subsequent items (such as .C, .C[2], and .D). All Primary Descriptor Sub-Nodes include a "common core" of information, with other fields added for certain specific sub-nodes. The "Normal Sub-Node" described here is used for all "normal" symbols, meaning all symbols for which no specific Primary Descriptor Sub-Node has been defined below. This Sub-Node has only the "common core" fields and no type-specific fields:



A pointer to a Primary Descriptor Sub-Node is declared as follows:

```
PNODEPTR: REF DBG$PRIM_NODE
```

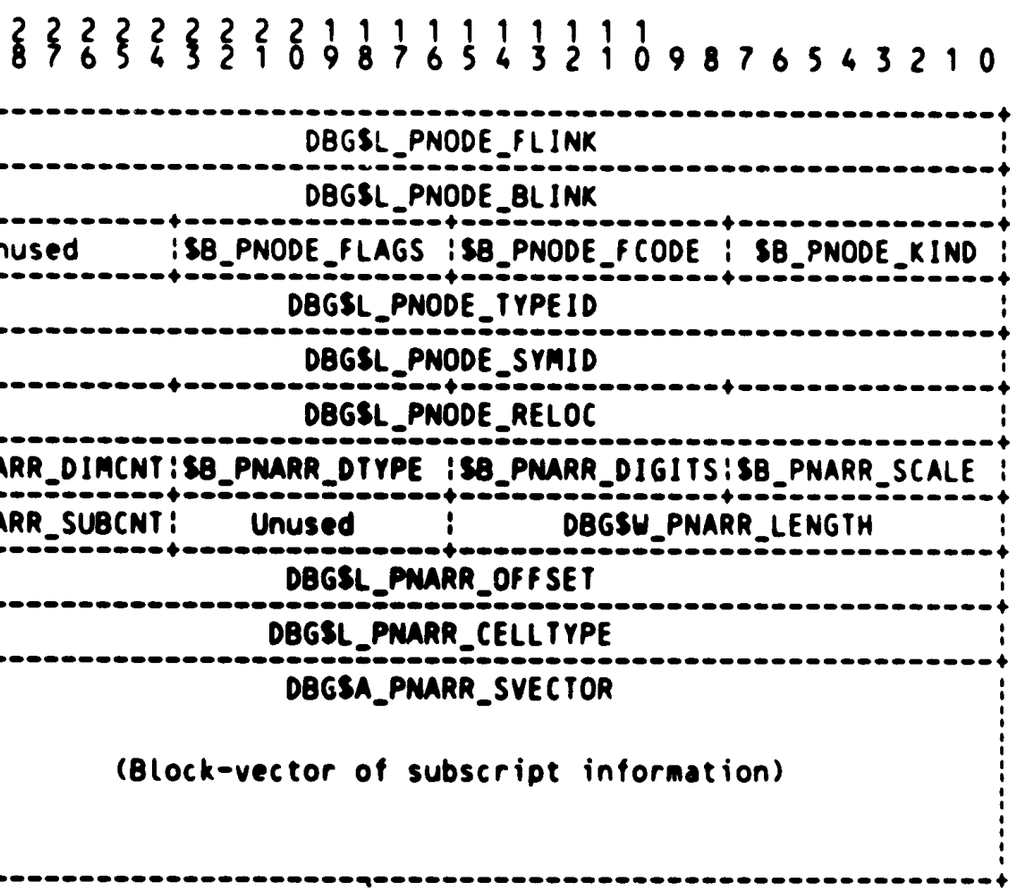
The following are the meanings of the fields in the Normal Sub-Node.

DBG$B_PNODE_KIND	-- May be any RST kind, although it usually is RST$K_DATA or RST$K_TYPCOMP.
DBG$B_PNODE_FCODE	-- Can be any FCODE other than those for which separate Primary Descriptor Sub-Nodes are described below.
DBG$L_PNODE_TYPEID	-- The Type ID for the object's type.
DBG$L_PNODE_SYMID	-- The SYMID for the object. This field may be zero, as it is for an array element or a pointed-to element.
DBG$L_PNODE_RELOC	-- A 'relocation constant' for the address. This is normally zero, but can contain a user address (in the case of anonymous references) or a byte offset (for strings)

```
!
! Define the size of the Primary Descriptor Normal Sub-Node.
!
LITERAL
        DBG$K_PRIM_SIZE_NORMAL  = 6;     ! Size of Normal Sub-Node in longwords

! Define the fields of the Primary Descriptor Sub-Node.  Also define the
! declaration macro.
!
FIELD DBG$PNODE_FIELDS =
        SET
        DBG$L_PNODE_FLINK  = [ 0, L_ ], ! Link to following primary sub-node
        DBG$L_PNODE_BLINK  = [ 1, L_ ], ! Link to preceding primary sub-node
        DBG$B_PNODE_KIND   = [ 2, B0_ ],! The RST "kind" of this component
        DBG$B_PNODE_FCODE  = [ 2, B1_ ],! FCODE  for this component
        DBG$B_PNODE_FLAGS  = [ 2, B2_ ],! Flags byte
        DBG$V_PNODE_EVAL   = [ 2, V2_(0) ], ! Evaluate subscripting, dot
                                            !   qualification, or dereferencing
                                            !   operation on this component
        DBG$V_PNARR_COLUMN = [ 2, V2_(1) ], ! Set if array in 'column major' order
        DBG$V_PNARR_BITREF = [ 2, V2_(2) ], ! Set if this is a BIT array
        DBG$V_PNARR_RANGE  = [ 2, V2_(3) ], ! Set if array has subscript ranges
        DBG$V_PNVAR_VALID  = [ 2, V2_(4) ], ! Set if Tag has been validated
        DBG$V_PNODE_IGNORE   = [ 2, V2_(5)],! Says to ignore SYMID in address
                                            !   computation.
        DBG$L_PNODE_TYPEID = [ 3, L_ ],     ! TYPEID for this component
        DBG$L_PNODE_SYMID  = [ 4, L_ ],     ! Generic SYMID for this component
        DBG$L_PNODE_RELOC  = [ 5, L_ ],     ! Address relocation factor

        DBG$W_PNREC_INDEX  = [ 6, W0_ ],! Record Component Index
        DBG$W_PNREC_NCOMPS = [ 6, W1_ ],! Number of components in record

        DBG$W_PNVAR_INDEX  = [ 6, W0_ ],! Variant Component Index
        DBG$W_PNVAR_NCOMPS = [ 6, W1_ ],! Number of components in variant
        DBG$L_PNVAR_TAGID  = [ 7, L_ ],! SYMID of tag variable (or 0)
        DBG$L_PNVAR_COMPLST= [ 8, L_ ], ! Pointer to list of record components
                                        !       in this record variant
        DBG$L_PNVAR_DSTPTR = [ 9, L_ ], ! Pointer to Variant Value DST Record
                                        !       for this record variant

        DBG$B_PNARR_SCALE  = [ 6, B0_ ],! Decimal scale factor for array element
        DBG$B_PNARR_DIGITS = [ 6, B1_ ],! Decimal digits for array element or 0
        DBG$B_PNARR_DTYPE  = [ 6, B2_ ],! DTYPE of an array element
        DBG$B_PNARR_DIMCNT = [ 6, B3_ ],! Number of array dimensions
        DBG$W_PNARR_LENGTH = [ 7, W0_ ],! Length of each array element
        DBG$B_PNARR_SUBCNT = [ 7, B3_ ],! Number of actual subscripts supplied
        DBG$L_PNARR_OFFSET = [ 8, L_ ], ! Offset to element [0,0,...,0] (bit or
                                        !       byte depending on BITREF flag)
        DBG$L_PNARR_CELLTYPE=[ 9, L_ ], ! Type ID of array element data type
        DBG$A_PNARR_SVECTOR= [10, A_ ]  ! BLOCKVECTOR of subscripts
        TES;

MACRO
        DBG$PRIM_NODE = BLOCK [,LONG] FIELD(DBG$PNODE_FIELDS)%;
```

PRIMARY DESCRIPTOR ARRAY SUB-NODE

This Primary Descriptor Sub-Node is used for array types (i.e., for Primary Symbol components whose FCODE is Array). The node contains all information needed later about the array type itself, including the dimension count, all subscript types and bounds, etc. This is the format:



The following are the meanings of the fields in the Array Type Sub-Node.

```
DBG$B_PNODE_KIND     -- Must be RST$K_DATA or RST$K_TYPCOMP.
DBG$B_PNODE_FCODE    -- Must be RST$K_TYPE_ARRAY.
DBG$L_PNODE_TYPEID   -- The Type ID for the array type.
DBG$L_PNODE_SYMID    -- The SYMID for the array (or zero).
DBG$B_PNARR_SCALE    -- The element scale factor or zero.
DBG$B_PNARR_DIGITS   -- The element digit count or zero.
DBG$B_PNARR_DTYPE    -- The element VAX standard type code (if
                        applicable) or zero.
```

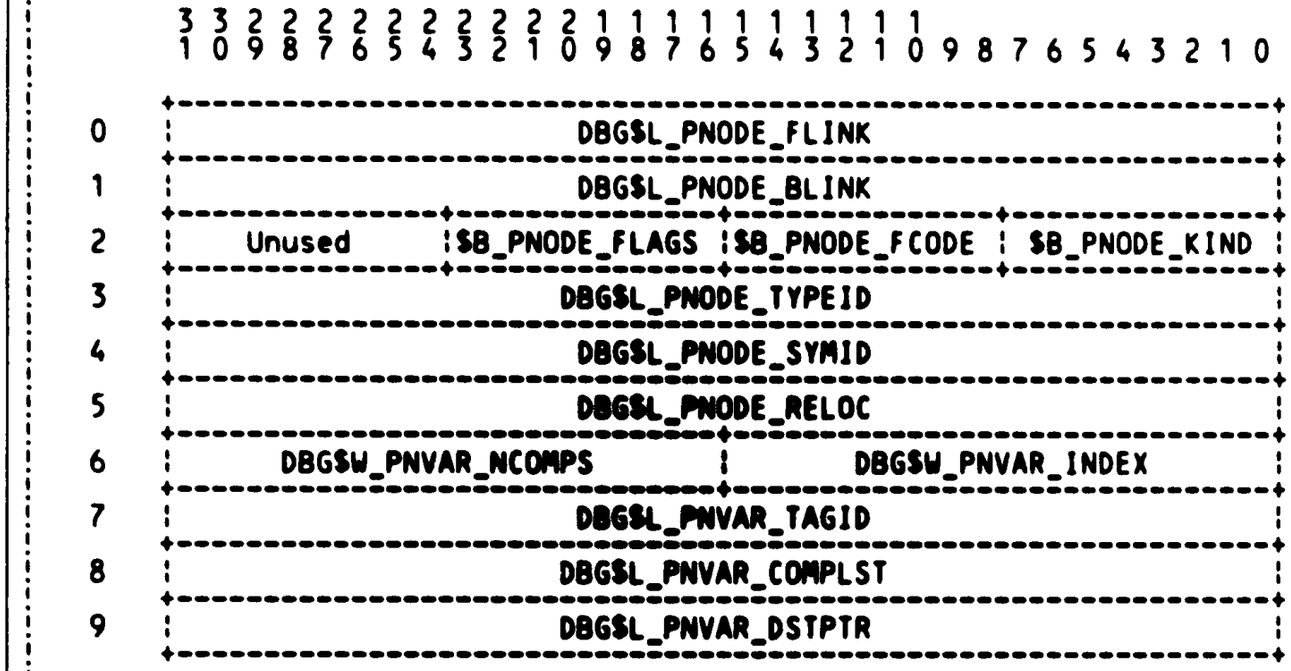
```
!       DBG$B_PNARR_DIMCNT    -- The number of array dimensions.
!       DBG$B_PNARR_SUBCNT    -- The number of actual subscripts supplied.
!                                In some languages this may be less than
!                                the number of array dimensions.
!       DBG$B_PNARR_LENGTH    -- The length of each array element.  This
!                                length is in bytes except for DTYPEs
!                                V (bits) and P (packed decimal).
!       DBG$L_PNARR_OFFSET    -- The offset to element ARR[0,0,...,0]
!                                from the start of the array.
!       DBG$L_PNARR_CELLTYPE -- The Type ID of the array element type.
!       DBG$A_PNARR_SVECTOR   -- Start of the subscript block-vector.
!
!
!       The block-vector of subscript information consists of a vector whose
!       elements are blocks of the following format.  There is once such block
!       for each dimension in the array.
!
!        3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
!        1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
!       +---------------------------------------------------------------+
!   0   !                       DBG$L_PNSUB_SVALUE                      !
!       +---------------------------------------------------------------+
!   1   !                       DBG$L_PNSUB_STRIDE                      !
!       +---------------------------------------------------------------+
!   2   !                       DBG$L_PNSUB_LBOUND                      !
!       +---------------------------------------------------------------+
!   3   !                       DBG$L_PNSUB_UBOUND                      !
!       +---------------------------------------------------------------+
!   4   !                       DBG$L_PNSUB_TYPEID                      !
!       +---------------------------------------------------------------+
!
!
! Define the fields in the subscript information block-vector.
!
FIELD DBG$PNSUB_FIELDS =
        SET
        DBG$L_PNSUB_SVALUE = [ 0, L_ ],  ! Subscript Value
        DBG$L_PNSUB_STRIDE = [ 1, L_ ],  ! Subscript Stride (bit or byte
                                         !      depending on BITREF flag)
        DBG$L_PNSUB_LBOUND = [ 2, L_ ],  ! Subscript Lower Bound
        DBG$L_PNSUB_UBOUND = [ 3, L_ ],  ! Subscript Lower Bound
        DBG$L_PNSUB_TYPEID = [ 4, L_ ]   ! Subscript type's Type ID.  If the
                                         !      subscript type is longword
                                         !      integer, this field is zero.
        TES;

MACRO
        DBG$PRIM_NODE_SUBS = BLOCKVECTOR[,5,LONG] FIELD(DBG$PNSUB_FIELDS) %;

! Define the size of the Primary Descriptor Array Sub-Node.
!
```

```
LITERAL
        DBG$K_PRIM_SIZE_ARRAY   = 10,   ! Size of fixed part in longwords
        DBG$K_PRIM_SIZE_SUBS    = 5;    ! Size of variable part per subscript,
                                        !      also in longwo ds
```

```
!       PRIMARY DESCRIPTOR RECORD SUB-NODE
!
!
!       This Primary Descriptor Sub-Node is used for record types (i.e., for
!       Primary Symbol components whose FCODE is Record).  The only extra
!       field in this sub-node is the record (or "structure") component
!       index which indicates which record component is selected from the
!       defined components of this record type.
!
!        3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
!        1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
!       +---------------------------------------------------------------+
!   0   !                       DBG$L_PNODE_FLINK                       !
!       +---------------------------------------------------------------+
!   1   !                       DBG$L_PNODE_BLINK                       !
!       +---------------+---------------+---------------+---------------+
!   2   !    Unused     !$B_PNODE_FLAGS !$B_PNODE_FCODE ! $B_PNODE_KIND !
!       +---------------+---------------+---------------+---------------+
!   3   !                     DBG$L_PNODE_TYPEID                        !
!       +---------------------------------------------------------------+
!   4   !                       DBG$L_PNODE_SYMID                       !
!       +-------------------------------+-------------------------------+
!   5   !                       DBG$L_PNODE_RELOC                       !
!       +---------------------------------------------------------------+
!   6   !      DBG$W_PNREC_NCOMPS       !       DBG$W_PNREC_INDEX       !
!       +-------------------------------+-------------------------------+
!
!
!  The following are the meanings of the fields in the Record Type Sub-Node.
!
!       DBG$B_PNODE_KIND      -- Must be RST$K_DATA or RST$K_TYPCOMP.
!       DBG$B_PNODE_FCODE     -- Must be RST$K_TYPE_RECORD.
!       DBG$L_PNODE_TYPEID    -- The Type ID for the record type.
!       DBG$L_PNODE_SYMID     -- The SYMID for the record (or zero).
!       DBG$W_PNREC_INDEX     -- The index of the selected record component
!                                in the set of record components for this
!                                record type.  This is meaningful only if
!                                the DBG$V_PNODE_EVAL is set--otherwise
!                                this field is always initialized to 1.
!       DBG$W_PNREC_NCOMPS    -- The number of components in this record
!
!  Define the size of the Primary Descriptor Record Sub-Node.
!
LITERAL
        DBG$K_PRIM_SIZE_RECORD  = 7;    ! Size of Record Sub-Node in longwords
```

PRIMARY DESCRIPTOR VARIANT SUB-NODE

This Primary Descriptor Sub-Node is used for variant types (i.e. for Primary Symbol components whose FCODE is variant). The extra fields in this sub-node indicate which is the currently selected component of the variant set (much as a record component is shown in the record sub-node), the number of components, and the SYMID of the associated TAG variable (or zero if no TAG variable) and other information.



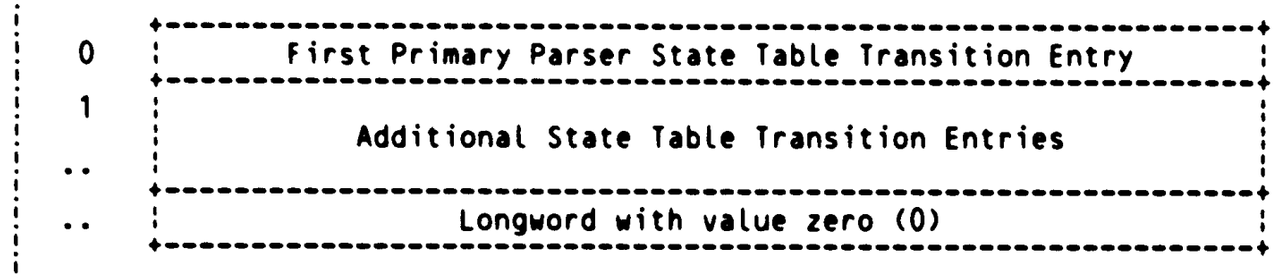
The following are the meanings of the fields in the VARIANT Type Sub-Node.

```
DBG$B_PNODE_KIND     -- Must be RST$K_VARIANT.
DBG$B_PNODE_FCODE    -- Must be RST$K_TYPE_VARIANT.
DBG$L_PNODE_TYPEID   -- The Type ID for the variant type.
DBG$L_PNODE_SYMID    -- Always Zero.
DBG$W_PNVAR_INDEX    -- The index of the selected component from
                        the set of components for this variant.
                        This field is always initialized to 1.
DBG$W_PNVAR_NCOMPS   -- The number of components in this variant.
DBG$L_PNVAR_TAGID    -- The SYMID of the TAG variable (or zero)
DBG$L_PNVAR_COMPLST  -- Pointer to the list of record components
                        in this record variant.
DBG$L_PNVAR_DSTPTR   -- Pointer to the Variant Value DST Record
                        for this record variant.
```

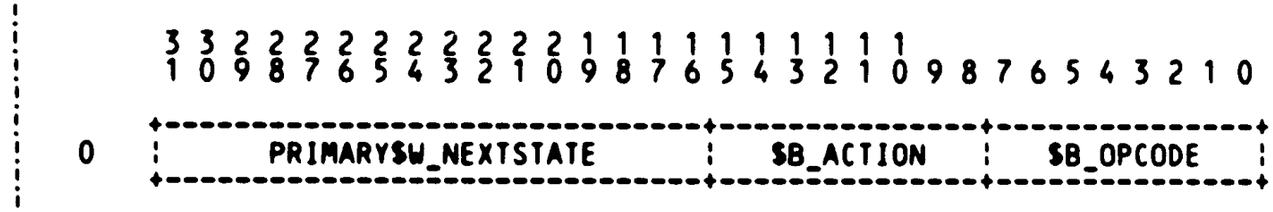
```
!
! Define the size of the Primary Descriptor Variant Sub-Node.
!
LITERAL
        DBG$K_PRIM_SIZE_VARIANT = 10;    ! Size of Variant Sub-Node in longwords
```

P R I M A R Y P A R S E R S T A T E T A B L E

The Primary Parser (DBG$PRIMARY_PARSER) operates off a Finite-State Machine (FSM) state table which defines the allowed formats of Primary Symbols in the current language. A Primary Symbol consists of a symbol name which may include pathname qualification, data qualification, subscripting, dereferencing, and possibly other qualification depending on the language. In PASCAL, for example, M\R\A.B[2,3]^.C[4].D is a valid Primary Symbol. The operators in a Primary Symbol may include "\", ".", "[]", "^", and so on. Exactly which operators are allowed and in which order they are allowed is language-dependent, and this is what is specified by the Primary Parser State Table. Each state in the state table is represented by zero or more Primary Parser State Table Transition Entries, each of which specifies a transition to be taken if the current operator is the one specified in the transition entry. The format of a Primary Parser State Table Entry is as follows:



The individual Primary Parser State Table Transition Entry has this format:



The whole Primary Parser State Table for a language simply consists of a sequence of concatenated State Table Entries, with one entry per state in the Finite-State Machine. The whole state table is thus a vector of transition entries and is indexed as such. Element zero of the state table is the beginning of the state table's Start State, and each transition's next state pointer (PRIMARY$W_NEXT_STATE) is a state table index which points to the beginning of the next state's state table entry.

A pointer to a Primary Parser State Table is declared as follows:

```
!
!                       PRIMARY_TABLE: REF PRIMARY$TABLE
!
!       This declares PRIMARY_TABLE to be a block vector which is referenced
!       as .PRIMARY_TABLE[.STATE_INDEX, field-name] where STATE_INDEX is the
!       vector index pointing to the current Primary Parser State Table
!       Transition Entry.
!
!
! Define the fields in the Primary parser State Table Transition Entry.  Also
! define the declaration macro.
!
FIELD PRIMARY$FLD_DEF =
        SET
        PRIMARY$B_OPCODE = [ 0, B0_ ],  ! Operator code which activates this
                                        !     state transition
        PRIMARY$B_ACTION = [ 0, B1_ ],  ! CASE index for semantic action to be
                                        !     taken during state transition
        PRIMARY$W_NEXTSTATE=[ 0 , W1_ ] ! Index of next state after transition
        TES;

MACRO
        PRIMARY$TABLE = BLOCKVECTOR[,1,LONG] FIELD(PRIMARY$FLD_DEF) %;


! Define the allowed values of the PRIMARY$B_ACTION field.  These values are
! CASE indeces which select the semantic action routine to be executed for
! each state transition.
!
LITERAL
        PRIMARY$K_MIN_ACTION            =  1,   ! --- Minimum action index
        PRIMARY$K_ACT_START_GBL           =  1, ! Global scope backslash found
        PRIMARY$K_ACT_GBL_TERM            =  2, ! Terminator after global symbol
        PRIMARY$K_ACT_START_SLASH         =  3, ! Backslash after first operand
        PRIMARY$K_ACT_START_DOT           =  4, ! Dot qualification after first operand
        PRIMARY$K_ACT_START_DOT_PLI       =  5, ! '' in PL/I
        PRIMARY$K_ACT_START_DOT_COB       =  6, ! '' in COBOL
        PRIMARY$K_ACT_START_SUBSCR        =  7, ! Subscripting after first operand
        PRIMARY$K_ACT_START_SUBSCR_BLI    =  8, ! Bliss subscript
        PRIMARY$K_ACT_START_SUBSCR_PLI    =  9, ! PL/I subscript
        PRIMARY$K_ACT_START_DEREF         = 10, ! PASCAL dereference ^
        PRIMARY$K_ACT_START_DEREF_PLI     = 11, ! PL/I "->" operator
        PRIMARY$K_ACT_START_TERM          = 12, ! Terminator after first operand
        PRIMARY$K_ACT_START_BIF_CALL      = 13, ! Built-in function call
        PRIMARY$K_ACT_START_TICK          = 48, ! Ada tick operator

        PRIMARY$K_ACT_SLASH_SLASH         = 14, ! Backslash after previous backslash
        PRIMARY$K_ACT_SLASH_INVOCNUM      = 15, ! Invocation Number in pathname
        PRIMARY$K_ACT_SLASH_DOT           = 16, ! Dot qualification after backslash
        PRIMARY$K_ACT_SLASH_DOT_PLI       = 17, ! '' in PL/I
        PRIMARY$K_ACT_SLASH_SUBSCR        = 18, ! Subscripting after backslash
        PRIMARY$K_ACT_SLASH_SUBSCR_PLI    = 19, ! '' in PL/I
        PRIMARY$K_ACT_SLASH_SUBSCR_BLI    = 20, ! Bliss subscript after backslash
        PRIMARY$K_ACT_SLASH_DEREF         = 21, ! PASCAL ^ after backslash
        PRIMARY$K_ACT_SLASH_DEREF_PLI     = 22, ! PL/I "->" operator after "/"
        PRIMARY$K_ACT_SLASH_TERM          = 23, ! Terminator operator after backslash
```

```
        PRIMARY$K_ACT_SLASH_TICK            = 49, ! Ada tick operator

        PRIMARY$K_ACT_DOT_SLASH_COB         = 24, ! A of B\C in COBOL
        PRIMARY$K_ACT_DOT_DOT               = 25, ! Dot qualification after previous dot
        PRIMARY$K_ACT_DOT_DOT_PLI           = 26, ! " in PL/I
        PRIMARY$K_ACT_DOT_DOT_COB           = 27, ! " in COBOL
        PRIMARY$K_ACT_DOT_SUBSCR            = 28, ! Subscripting after dot qualification
        PRIMARY$K_ACT_DOT_SUBSCR_PLI        = 29, ! " in PL/I
        PRIMARY$K_ACT_DOT_SUBSCR_COB        = 30, ! " in COBOLL
        PRIMARY$K_ACT_DOT_DEREF             = 31, ! PASCAL ^ after dot
        PRIMARY$K_ACT_DOT_DEREF_PLI         = 32, ! PL/I "->" operator after "."
        PRIMARY$K_ACT_DOT_TERM              = 33, ! Terminator operator after dot qualif.
        PRIMARY$K_ACT_DOT_TERM_PLI          = 34, ! " in PL/I
        PRIMARY$K_ACT_DOT_TERM_COB          = 35, ! " in COBOL
        PRIMARY$K_ACT_DOT_TICK              = 50, ! Ada tick operator

        PRIMARY$K_ACT_SUBSCR_DOT            = 36, ! Dot qualification after subscripting
        PRIMARY$K_ACT_SUBSCR_DOT_PLI        = 37, ! " in PL/I
        PRIMARY$K_ACT_SUBSCR_SUBSCR         = 38, ! Subscripting after subscripting
        PRIMARY$K_ACT_SUBSCR_SUBSCR_PLI     = 39, ! " in PL/I
        PRIMARY$K_ACT_SUBSCR_DEREF          = 40, ! PASCAL ^ after []
        PRIMARY$K_ACT_SUBSCR_DEREF_PLI      = 41, ! PLI "->" operator after subscript
        PRIMARY$K_ACT_SUBSCR_TERM           = 42, ! Terminator operator after subscripting
        PRIMARY$K_ACT_SUBSCR_TERM_PLI       = 43, ! " in PL/I

        PRIMARY$K_ACT_DEREF_DOT             = 44, ! Dot after dereferencing
        PRIMARY$K_ACT_DEREF_SUBSCR          = 45, ! Subscripting after dereferencing
        PRIMARY$K_ACT_DEREF_DEREF           = 46, ! Dereferencing after dereferencing
        PRIMARY$K_ACT_DEREF_TERM            = 47, ! Terminator after dereferencin

        PRIMARY$K_MAX_ACTION              = 50;   ! --- Maximum action index
```

```
!                P R I N T   I N F O R M A T I O N   T A B L E
!
!       The Print Information Table is a blockvector indexed by FCODE
!       which gives pointers to tables and print routine indexes necessary
!       to print primary symbols for a given language.
!
!       3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
!       1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
!       +---------------------------------------------------------------+
!   0   !                   PRTINFO$L_ROUT_INDEX                        !
!       +---------------------------------------------------------------+
!   1   !                   PRTINFO$L_CHARTBL                           !
!       +---------------------------------------------------------------+
!
!       A pointer to an Print Information Table is declared as follows:
!
!               PRINT_INFO_TABLE: REF PRTINFO$TABLE
!
!
! Define the fields of the Print Information Table entry.  Also define the
! declaration macro.

FIELD PRTINFO$FLD_DEF =
        SET
        PRTINFO$L_ROUT_INDEX    = [ 0, L_ ],
        PRTINFO$L_CHARTBL       = [ 1, L_ ]
        TES;

MACRO
    PRTINFO$TABLE = BLOCKVECTOR[RST$K_TYPE_MAXIMUM + 1, 2, LONG]
        FIELD(PRTINFO$FLD_DEF) %;

! Define valid PRTINFO$L_ROUT_INDEX literals.
!
LITERAL
        PRT$K_MIN_ROUT          = 0,
        PRT$K_OTHER             = 0,
        PRT$K_ARRAY             = 1,
        PRT$K_POINTER           = 2,
        PRT$K_RECORD            = 3,
        PRT$K_RECORD_COB        = 4,
        PRT$K_POINTER_C         = 5,
        PRT$K_RECORD_C_PLI      = 6,
        PRT$K_VARIANT           = 7,
        PRT$K_MAX_ROUT          = 7;

! Define valid Print Character Table Indexes.
!
LITERAL
```

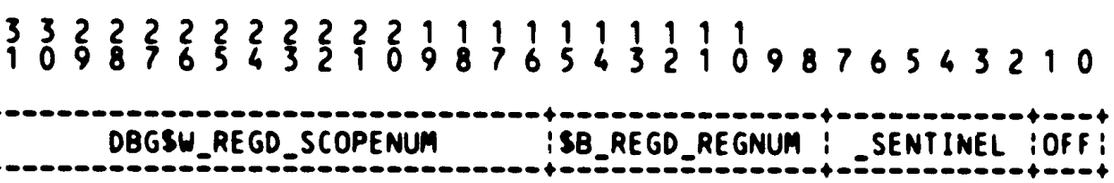
```
        PRT$K_BEGIN_CHAR        = 0,
        PRT$K_SEPARATOR_CHAR    = 1,
        PRT$K_END_CHAR          = 2,
        PRT$K_RECPTR_CHAR       = 3,
        PRT$K_MAX_PRTCHAR       = 4;
```

R E G I S T E R D E S C R I P T O R S

A Register Descriptor is used to describe the absolute address of a register or of a byte address within a register. It thus contains a register number, a byte offset within that register (0, 1, 2, and 3 are the only possible values), and a scope number which indicates to which call frame in the call stack this register belongs. The descriptor also contains a "sentinel" value which is used for validity checking and for ensuring that a Register Descriptor always has a non-zero value.

Register Descriptors are built by routine DBG$STA_ADDRESS_TO_REGDESCR and are later used for various purposes when registers are examined. They are also required by routine DBG$STA_REGISTER_NAME which generates the register name for printing.

A Register Descriptor fits in a single longword. This is the format:



A Register Descriptor is declared as follows:

```
REGDESCR: DBG$REGDESCR
```

```
! Define the fields of the Register Descriptor.  Also define the declaration
! macro.

FIELD DBG$REGD_FLD_DEF =
        SET
        DBG$V_REGD_OFFSET    = [ 0, V_(0,2) ],    ! Byte offset from start of register
        DBG$V_REGD_SENTINEL = [ 0, V_(2,6) ],    ! Sentinel value--must be %X'20'
        DBG$B_REGD_REGNUM    = [ 0, BT_ ],        ! The actual register number
        DBG$W_REGD_SCOPENUM = [ 0, W1_ ]         ! The register's scope number
        TES;

MACRO
        DBG$REGDESCR = BLOCK[1,LONG] FIELD(DBG$REGD_FLD_DEF) %;
```

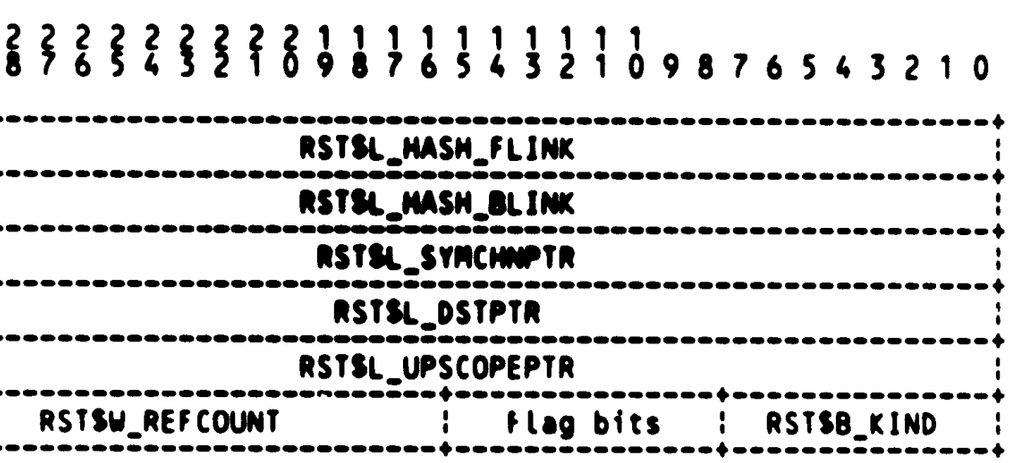
R U N - T I M E S Y M B O L T A B L E D E F I N I T I O N S

This section contains all definitions related to the structure and contents of the Run-Time Symbol Table (RST) through which the Debugger accesses the Debug Symbol Table (DST) in the user's executable image. The RST itself is built and accessed entirely through the Symbol-Table Access Routines in modules RSTCNTRL and RSTACCESS.

The RST is the structure through which the DST is accessed, one module at a time. This means that for each "active" module in the user's program, there is an RST data structure which describes that module and all lexical entities and data items in it. This structure, which is scattered through the Debugger's free memory, contains one entry for each lexical entity and data item in the module's DST. Each RST entry contains a pointer to the corresponding DST entry so all the DST entry information can be accessed rapidly. In addition, the RST itself is hashed by symbol name so that a symbol's RST and DST entries can be located rapidly given the symbol name.

RST ENTRY COMMON CORE

All entries in the Run-Time Symbol Table have the same information in their first parts. This RST entry "common core" is described here:



The hash chain forward and backward links link together all RST entries for symbols whose names have the same hash value. The hash chain is doubly linked so that RST entries can be removed from the hash chains when the whole RST for a module is removed to free up its space. Each hash chain has a permanent list head; RST entries in the RST Hash Table therefore never have zero hash links.

Some RST entries are not in the RST Hash Table, however. Such entries are "temporary" in the sense that they are created when needed and are thrown away when no longer referenced. Examples include many Data Type RST Entries and RST entries with invocation numbers. Such RST entries are put on the Temporary RST Entry List. This is a singly linked list which uses the RST$L_HASH_FLINK field for the links; RST$L_HASH_BLINK is left zero in this case.

All RST entries for a given module are linked together through the RST$L_SYMCHNPTR pointer in a singly linked list. The list starts at the Module RST Entry and is terminated by a zero forward link. This list is traversed when the Module's whole RST is released to make room for another module.

The RST$L_DSTPTR pointer gives the address of the symbol's DST entry. The RST$L_UPSCOPEPTR pointer generally points to the RST entry of the symbol's "containing" entity. This usually means the containing lexical entity, but for data record components it means the containing record.

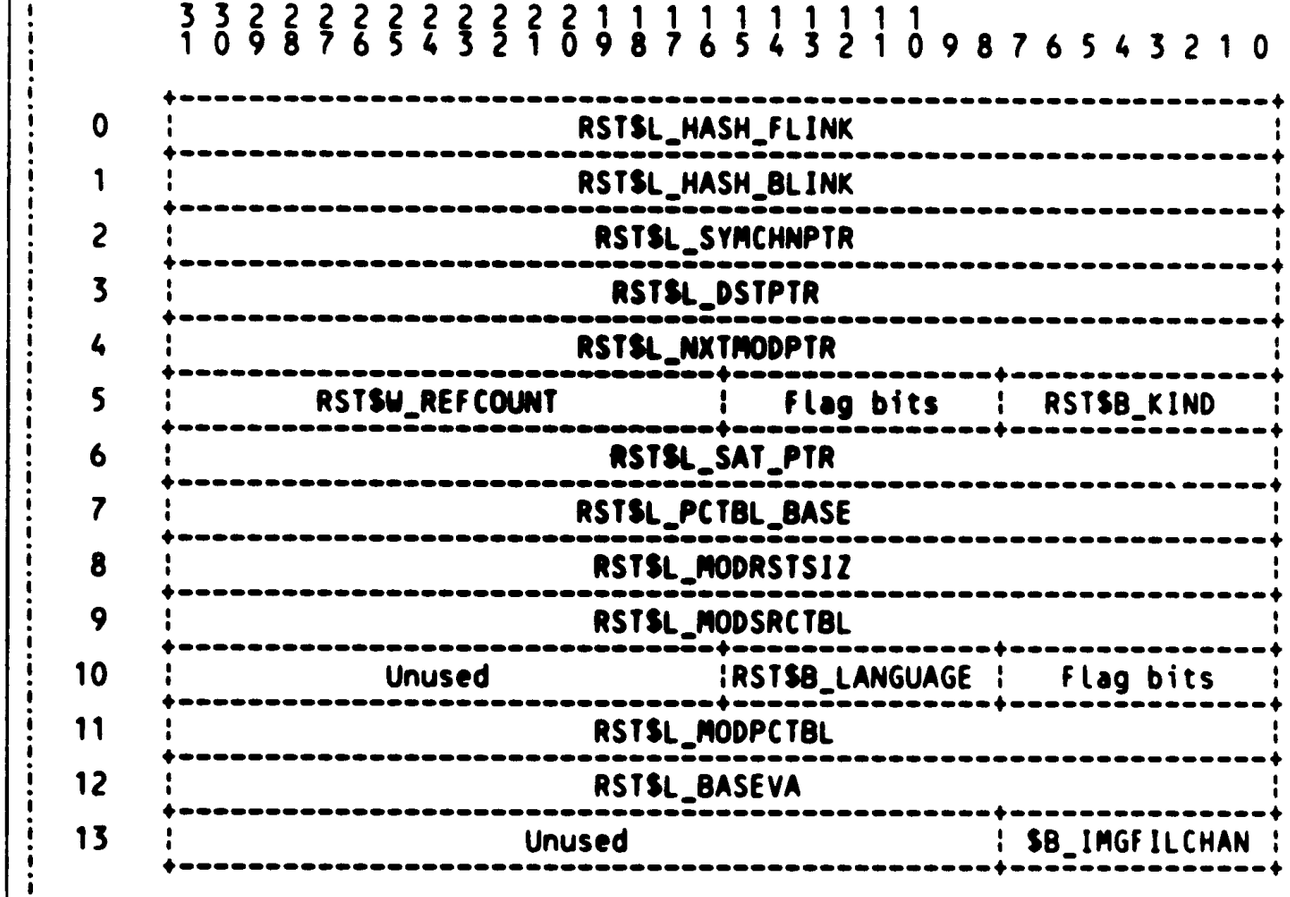
The RST$B_KIND field identifies what kind of symbol and what kind of RST entry this is. The possible values are listed at the end of this section. The RST$W_REFCOUNT field contains a reference count which speci-

```
!       fies how many references (pointers) to this RST entry have been passed
!       to the rest of the Debugger.  When this count is non-zero, this RST en-
!       try cannot be released to free storage.  The reference count is thus
!       used to prevent a "dangling pointer" problem.
!
!       A pointer to a Run-Time Symbol Table Entry is declared as follows:
!
!                       RSTPTR: REF RST$ENTRY
!
!
! This declaration macro collects together all the individual FIELD sets
! declared below for the various kinds of RST entries.
!
MACRO
        RST$ENTRY = BLOCK[,LONG] FIELD(RST$FLD_CORE,RST$FLD_MOD,RST$FLD_LEX,RST$FLD_DATA) %;


! These are the RST entry common core field definitions.
!
FIELD RST$FLD_CORE =
        SET
        RST$L_HASH_FLINK = [ 0, L_ ],   ! Symbol name hash chain forward link
        RST$L_HASH_BLINK = [ 1, L_ ],   ! Symbol name hash chain backward link
        RST$L_SYMCHNPTR  = [ 2, L_ ],   ! Pointer to the next RST entry belong-
                                        !       to the same module as this one
        RST$L_DSTPTR     = [ 3, L_ ],   ! Pointer to this symbol's DST entry
        RST$L_UPSCOPEPTR = [ 4, L_ ],   ! Pointer to the RST entry one level up
                                        !       in scope from this one
        RST$B_KIND       = [ 5, B0_ ],  ! The kind of RST entry this is
        RST$V_GLOBAL     = [ 5, V1_(0)],! Flag set if this is a global symbol
        RST$V_NONZLENGTH = [ 5, V1_(1)],! Flag set to TRUE if this symbol has a
                                        !       non-zero length
        RST$V_INVOCNUM   = [ 5, V1_(2)],! Flag set if symbol has invocation num.
        RST$V_SET_TYPEPTR= [ 5, V1_(3)],! Flag set to indicate that the Data RST
                                        !       Entry RST$L_TYPEPTR field should
                                        !       be set during RST build.
        RST$V_MARKBIT    = [ 5, V1_(4)],! Mark bit available for temporary use
                                        !       (such as stopping recursion)
        RST$V_COBOLGBL   = [ 5, V1_(5)],! Flag bit set to indicate this symbol
                                        !       has the COBOL "global" attribute
        RST$V_REGISTER   = [ 5, V1_(6)],! Flag set if SYMID represents a register
        RST$W_REFCOUNT   = [ 5, W1_ ]   ! Reference count for storage management
        TES;


!***
!
! The following are the possible values for the RST$B_KIND field.  This field
! specifies what kind of RST entry this is; it thus indicates both what kind of
! symbol the RST entry represents and what the format of the RST entry is after
! the "common core" part.
!
LITERAL
        RST$K_INVALID   = 0,            ! Invalid code--cannot occur in RST en-
                                        !       try but can be returned by some
                                        !       Symbol Table Access routines
```

```
        RST$K_NOTUNIQUE = 9,            ! Symbol not unique--cannot occur in RST
                                        !      entry but can be returned by the
                                        !      DBG$STA_GETSYMBOL routine.
        RST$K_MODULE    = 1,            ! Module RST entry
        RST$K_ROUTINE   = 2,            ! Routine RST entry
        RST$K_BLOCK     = 3,            ! Block RST entry
        RST$K_ENTRY     = 8,            ! Entry Point RST entry
        RST$K_LABEL     = 4,            ! Label RST entry
        RST$K_LINE      = 5,            ! Line number RST entry
        RST$K_DATA      = 6,            ! Data item RST entry
        RST$K_TYPCOMP   = 10,           ! Type component RST entry
        RST$K_TYPE      = 7,            ! Data type RST entry
        RST$K_VARIANT   = 11,           ! Record variant set RST entry
        RST$K_INVOCNUM  = 12,           ! Invocation number RST Entry
        RST$K_OVERLOAD  = 13,           ! Overloaded Symbol RST Entry
        RST$K_KIND_MINIMUM = RST$K_INVALID,     ! Minimum possible kind value
        RST$K_KIND_MAXIMUM = RST$K_OVERLOAD;    ! Maximum possible kind value
!**-
```

RST ENTRY FOR A MODULE

A module is represented by a Module RST Entry at all times, whether the rest of the module is in the RST or not. This Module Entry then points to a linked list of RST entries for the remaining symbols in the module via the RST$L_SYMCHNPTR pointer; the end of this list is indicated by a zero pointer. The RST$L_UPSCOPEPTR pointer is not used as an up-scope pointer in a Module Entry--there is nothing up-scope from a module. Instead RST$L_NXTMODPTR, which occupies the same location in the RST entry as RST$L_UPSCOPEPTR, is used to link all Module RST Entries in the whole program into a singly linked list. This list is scanned during RST initialization and when processing the SHOW MODULES command.



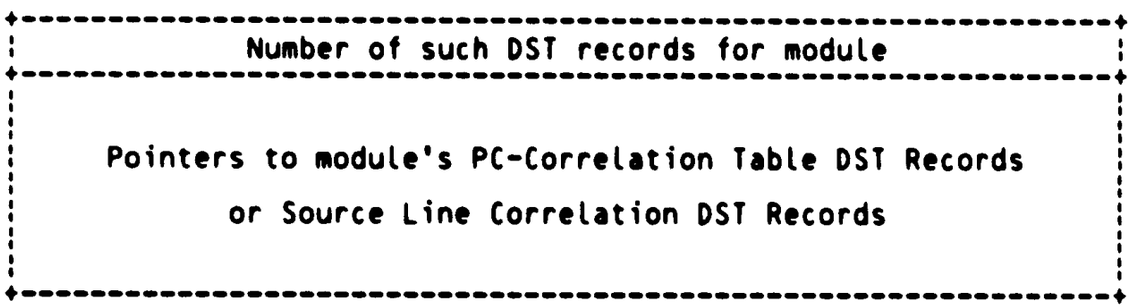
The RST$L_MODPCTBL field points to a table of pointers to the PC-Correlation Table DST records for this module (used for PC to line number correlation and vice versa). The first cell of this table (namely MOD_PC_TBL[0]) gives the number of PC-Correlation Table DST records in the module's DST, and the remaining cells (i.e., MOD_PC_TBL[1] through MOD_PC_TBL[.MOD_PC_TBL[0]]) contain pointers to those DST records. Each

cell is one longword. If there are no PC-Correlation Table DST records for a given module, the RST$L_MODPCTBL field is zero.

PC-Correlation Table DST records need not have any special order or nesting with respect to the DST records for routines and lexical blocks and are therefore viewed as belonging to the module as a whole. The PC-Correlation Tables for a module are scanned by searching them in the order of their pointers in the RST$L_MODPCTBL table. The addresses given by the PC Correlation Table DST records are relative to the value of the RST$L_PCTBL_BASE field, which contains the address of the lowest address routine in the module.

The RST$L_MODSRCTBL field points to a table of pointers to the Source Line Correlation DST Records for this module. The first cell of this table (i.e., MODSRCTBL[0]) contains the number of Source Line Correlation DST Records in this module's DST and the remaining cells (i.e., MODSCRTBL[1] through MODSRCTBL[.MODSRCTBL[0]]) contain pointers to those DST records. Each cell in the table is a longword. If there are no such DST records at all, the RST$L_MODSRCTBL field is zero.

The memory blocks pointed to by RST$L_MODPCTBL and RST$L_MODSCRTBL both have the following format:



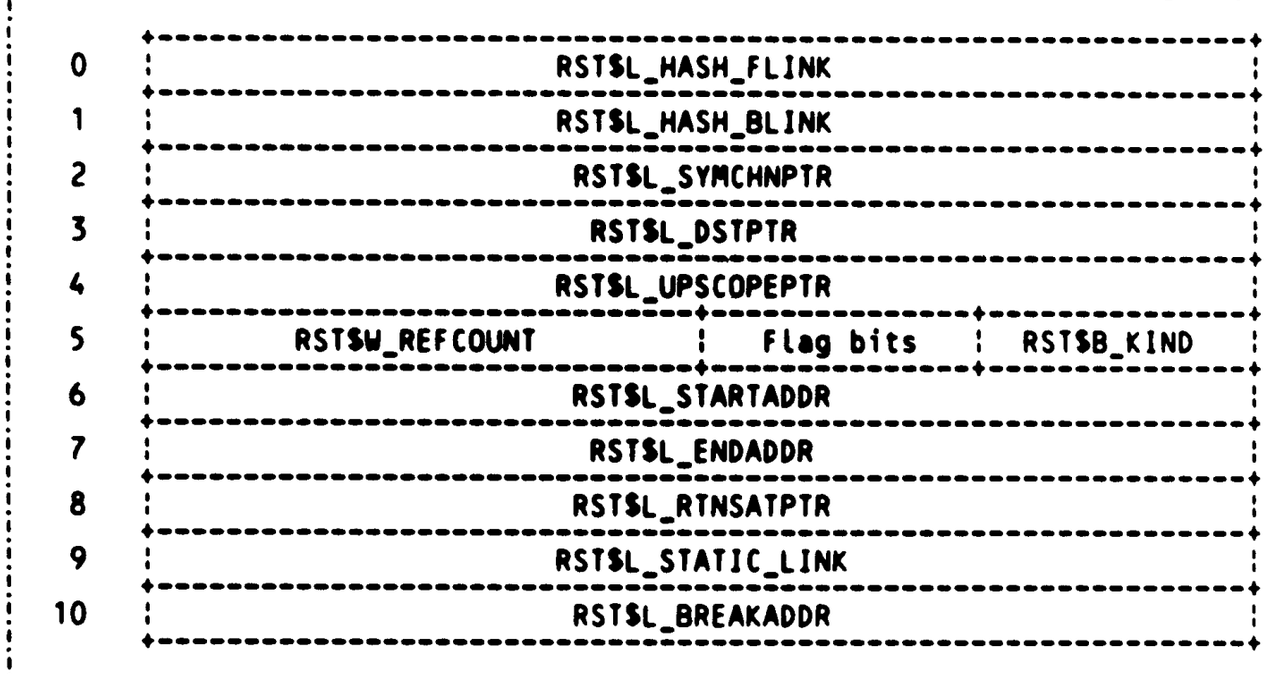
When registers are examined using a numbered scope which does not correspond to any routine in the RST, a dummy Module RST Entry is created to represent that scope. In such an RST entry, the RST$V_MODNUMSCP bit is set to distinguish it from other Module RST Entries. In addition, the scope number (the number of call frames from the top of the call stack) of the desired scope is stored in the RST$L_MODSCPNUM field. This field overlays the RST$L_MODRSTSIZ field, which is not needed in this case. The name of such a "module" is just the scope number in decimal ASCII. These special Module RST Entries are built by the GET_REGISTER_SYMID routine and are recognized by the DBG$STA_SETCONTEXT routine, both in RSTACCESS.

This set of field declarations includes all fields outside the RST common core

```
! used in the Module RST Entry.
!
FIELD RST$FLD_MOD =
        SET
        RST$L_NXTMODPTR = [ 4, L_ ],    ! Pointer to the next Module RST Entry
        RST$L_SAT_PTR   = [ 6, L_ ],    ! Pointer to this module's Static
                                        !       Address Table (SAT)
        RST$L_PCTBL_BASE= [ 7, L_ ],    ! The base address for the offsets given
                                        !       by the PC Correlation Tables
        RST$L_MODRSTSIZ = [ 8, L_ ],    ! The total size in bytes of the RST
                                        !       for this module--used only by the
                                        !       SHOW MODULE command.
        RST$L_MODSCPNUM = [ 8, L_ ],    ! The scope number if the MODNUMSCP bit
                                        !       described below is set
        RST$L_MODSRCTBL = [ 9, L_ ],    ! Pointer to table of pointers to the
                                        !       module's Source Line Correlation
                                        !       DST Records (or zero)
        RST$V_MODSET    = [ 10, V_(0) ],! Flag indicating this module is SET
        RST$V_MOD_IN_RST= [ 10, V_(1) ],! Flag indicating module is in the RST
        RST$V_ANONMOD   = [ 10, V_(2) ],! Flag indicating this is the "anonymous
                                        !       module" containing all unclaimed
                                        !       global symbols.
        RST$V_MODNUMSCP = [ 10, V_(3) ],! Flag indicating that this is a special
                                        !       RST entry for a numbered scope
        RST$V_SHARE_IMAGE=[ 10, V_(4) ],! Flag indicating that this is a shared
                                        !       image
        RST$V_OLDPLIFLAG= [ 10, V_(5) ],! Flag indicating this is a PL/I module
                                        !       compiled with an old compiler
                                        !       which generates incorrect VAX
                                        !       standard array descriptors
        RST$B_LANGUAGE  = [ 10, B1_ ],  ! The language of this module
        RST$L_MODPCTBL  = [ 11, L_ ],   ! Pointer to a table of pointers to the
                                        !       module's PC-Correlation Table
                                        !       DST records (or zero).
        RST$L_BASEVA    = [ 12, L_ ],   ! The base address of a shared image
        RST$B_IMGFILCHAN= [ 13, B_ ]    ! The executable shared image file
                                        !       channel number
        TES;

! Define a symbolic name for the size of the Module RST Entry in longwords.
!
LITERAL
        RST$K_MODENTSIZ = 12,           ! Size of Module RST Entry
        RST$K_SHARED_MODENTSIZ = 14;    ! Size of Shared Image Module RST
                                        !       Entry
```

RST ENTRY FOR A ROUTINE

The RST entry for a routine has the format shown here. Each Routine RST Entry has an up-scope pointer (RST$L_UPSCOPEPTR) which points to the RST entry for the containing lexical entity. That can be a module, a lexical block, or another routine. The start and end addresses of the routine's code are also given in the RST entry. There is also a Static Address Table (SAT) pointer which points to the SAT entry for this routine, and there is a Static Link pointer which points to the routine's Static Link DST record (or is zero). The routine breakpoint address is also kept in the RST Entry--this address may be set from a Prolog DST Record. The Routine RST Entry is always built when the RST for a module is built.



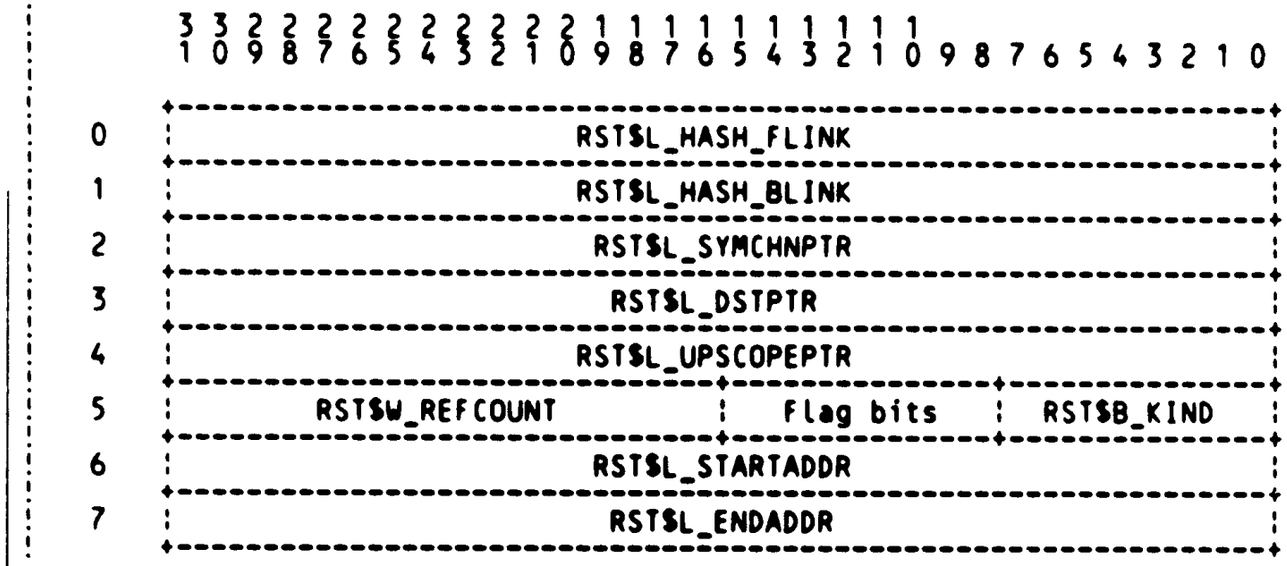
This set of field declarations includes the fields outside the RST common core used in the Routine, Lexical Block, Entry Point, Label, and Line Number RST Entries. The individual illustrations show which RST entries use which fields.

```
FIELD RST$FLD_LEX =
        SET
        RST$L_STARTADDR = [ 6, L_ ],    ! Lexical entity start address
        RST$L_ENDADDR   = [ 7, L_ ],    ! Lexical entity end address
        RST$L_RTNSATPTR = [ 8, L_ ],    ! Address of Routine's SAT entry
        RST$L_STATIC_LINK = [ 9, L_ ],  ! Address of Static Link DST record or 0
```

```
        RST$L_BREAKADDR = [ 10, L_ ]    ! Routine breakpoint address (start
                                        !      address, start address + 2, or
                                        !      address from Prolog DST Record)
        TES;

! Define symbolic names for the lengths of the lexical entity RST entries.
! All lengths are expressed in longwords.
!
LITERAL
        RST$K_ROUTENTSIZ = 11,          ! Size of Routine RST Entry
        RST$K_LEXENTSIZ  = 8,           ! Size of Lexical Block RST Entry
        RST$K_EPTENTSIZ  = 11,          ! Size of Entry Point RST Entry
        RST$K_LBLENTSIZ  = 7,           ! Size of Instruction Label RST Entry
        RST$K_LINENTSIZ  = 8;           ! Size of Line Number RST Entry
```

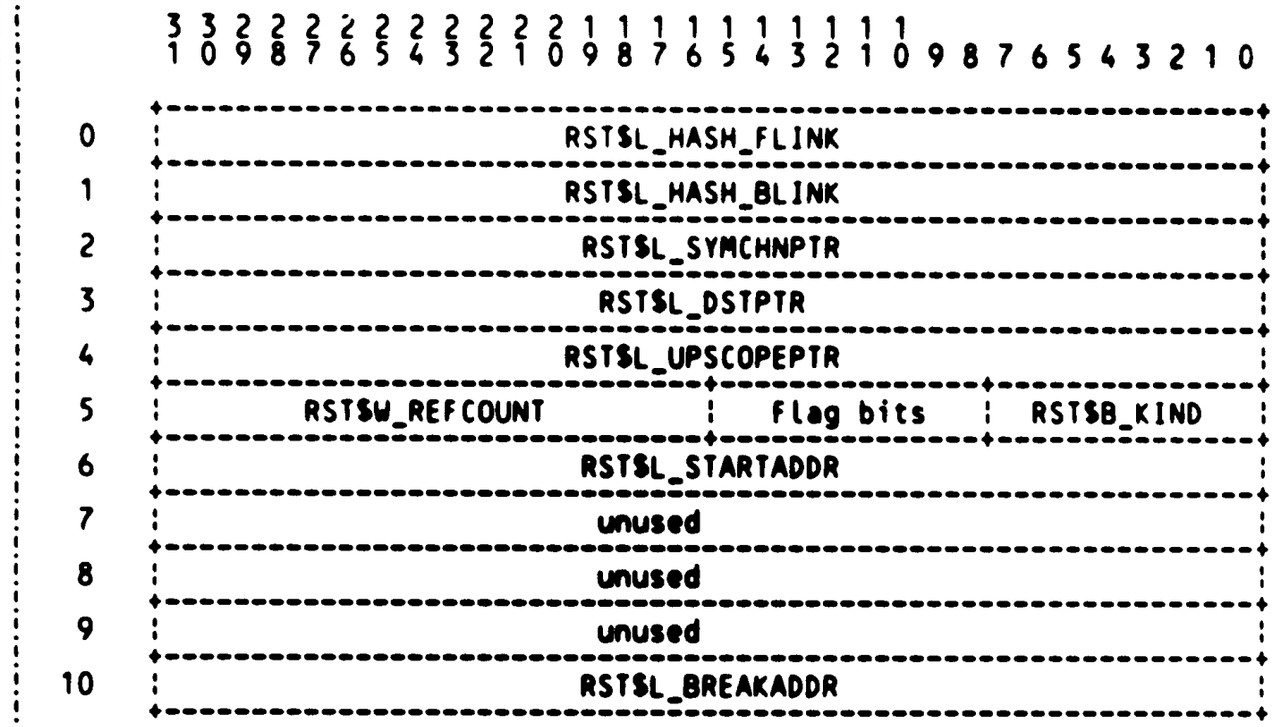
RST ENTRY FOR A LEXICAL BLOCK

The RST entry for a lexical block has the same format as that for a routine. Again the up-scope pointer points to the RST entry of the containing lexical entity, and the start and end addresses of the block's code are given. The Lexical Block RST Entry is always built when the RST for the whole module is built.



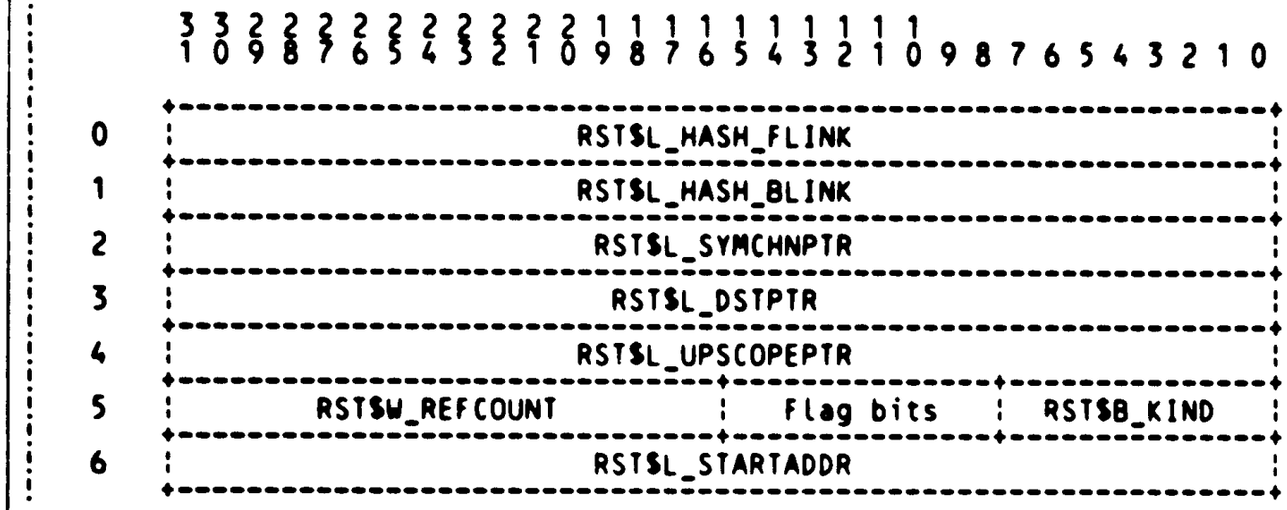
RST ENTRY FOR AN ENTRY POINT

The RST entry for an entry point (i.e., an alternate point through which a routine can be called) is shown here. The up-scope pointer points to the RST entry of the containing lexical entity. An entry point has no independent extent and therefore its RST entry has only a start address. The Entry Point RST Entry is always built when the whole module's RST is built.



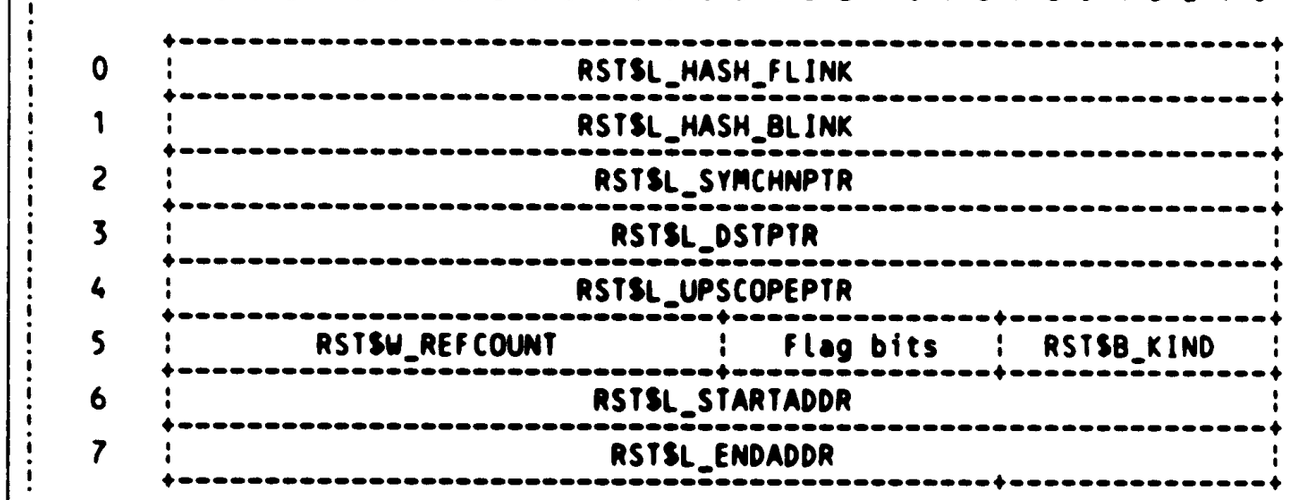
RST ENTRY FOR AN INSTRUCTION LABEL

The RST entry for a label (which labels a point in the user's code) is shown here. The up-scope pointer points to the RST entry of the containing lexical entity. Since a label does not have extent, there is only a start address in the RST entry; no end address is given. The Label RST Entry is always built when the whole module's RST is built.



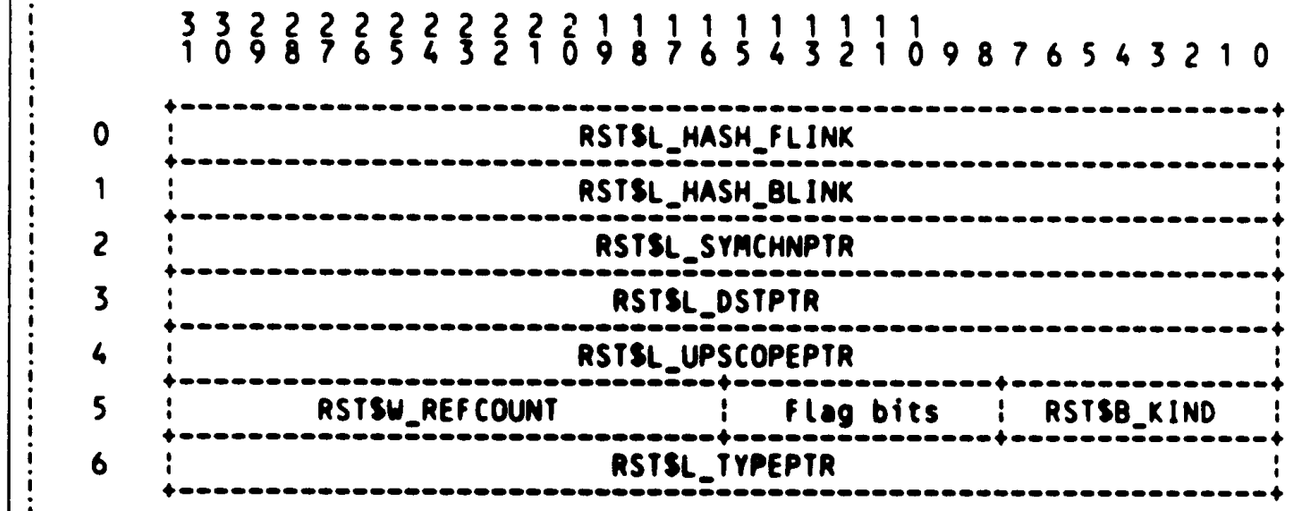
RST ENTRY FOR A LINE NUMBER

The RST entry for a line number has the format shown here. The up-scope pointer points to the RST entry of the containing lexical entity (always a routine or a block). Both the start and the end address are given. Line Number RST Entries are not built when the whole module's RST is built, however; there are too many lines in a program to make this practical. Instead each Line Number RST Entry is built when the corresponding line number is referenced through the Symbol Table Access routines. A dummy Label DST Record for the Line Number is built in the same memory block as the Line Number RST Entry. The RST entry points to this DST record and the DST record's name field contains the line number as a counted ASCII string (e.g. "%LINE 25.2").



RST ENTRY FOR A DATA SYMBOL

The RST entry for a data symbol has the format shown here. The up-scope pointer points to the RST entry for the containing lexical entity (i.e., module, routine, or lexical block) or, if this is a data component, to the RST entry for the containing data object. The Data Symbol RST Entry also has a Type Pointer (RST$L_TYPEPTR). For simple data types (those defined by one-byte type codes or standard descriptors) this pointer is zero, but for more complex data types (records, variants, and enumeration types) this pointer points to the RST entry for that data type. If the type pointer is zero, the type can be found directly in the data symbol's DST entry.



This set of field declarations includes the fields outside the RST common core used in the Data Symbol, Data Type Component, Data Type, Variant Set, and Invocation Number RST Entries. The individual illustrations show which RST entries use which fields.

```
FIELD RST$FLD_DATA =
        SET
        RST$L_TYPEPTR    = [ 6, L_ ],    ! A pointer to the Data Type RST Entry
                                         !      which gives the data type of this
                                         !      data object.  For simple types,
                                         !      this pointer is zero.
        RST$B_FCODE      = [ 6, B0_ ],   ! The "Format Code" of a Data Type RST
                                         !      Entry--indicates nature of type:
                                         !      array, atomic, record, etc.
        RST$W_TYPREFCNT  = [ 6, W1_ ],   ! The number of Data RST Entries that
                                         !      reference this Type RST Entry
        RST$L_TYPREFTBL  = [ 7, L_ ],    ! Pointer to Type Reference Table
        RST$L_BITSIZE    = [ 8, L_ ],    ! The length in bits of data items of
                                         !      this data type
```

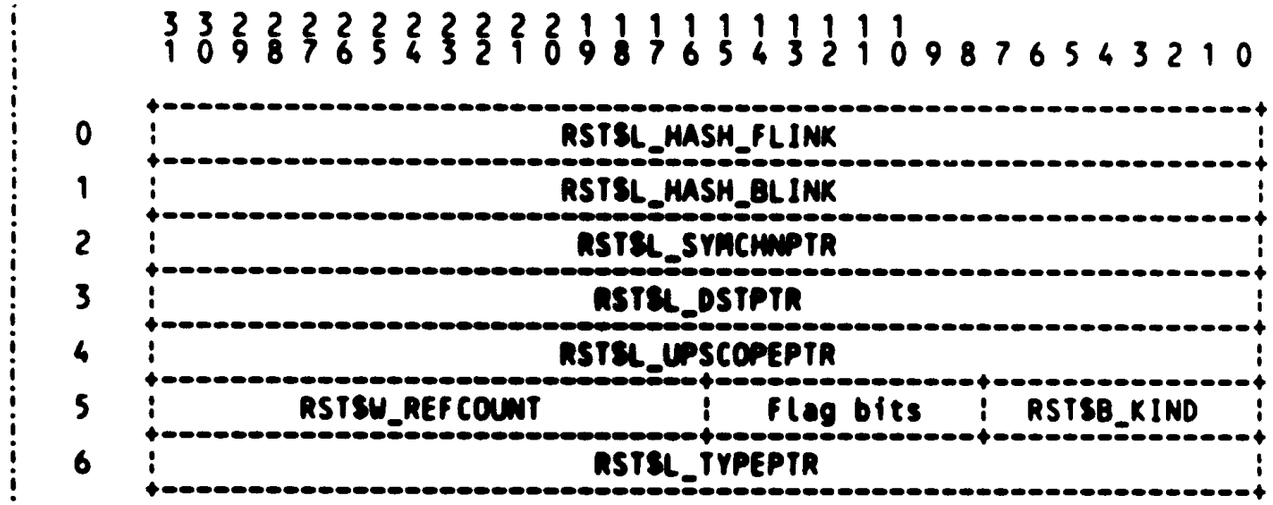
```
        RST$L_DST_TYP_REC_PTR =         ! A pointer to the DST record containing
                        [ 9, L_ ],      !      the embedded type-spec described
                                        !      by the DEBUG built DST record
                                        !      pointed to by RST$L_DSTPTR.
        RST$L_TYPCOMPCNT= [ 10, L_ ],   ! The number of record components if
                                        !      this is a data record type
        RST$A_TYPCOMPLST= [ 11, A_ ],   ! The start of a table of record compon-
                                        !      ent RST pointers
        RST$L_VARSETCNT = [ 2, L_ ],    ! The number of distinct variants in
                                        !      this record variant set
        RST$L_VARTAGPTR = [ 4, L_ ],    ! A pointer to the RST entry of the tag
                                        !      variable for this variant set
        RST$A_VARSETTBL = [ 6, A_ ],    ! The start of a table of variant set
                                        !      members, giving the tag value and
                                        !      component list for each variant.
        RST$L_INVOCNUM  = [ 6, L_ ]     ! The desired invocation number
        TES;

! Declare symbolic names for the lengths of the various data, type, and
! invocation number RST entries.  All lengths are in longwords.
!
LITERAL
        RST$K_DATENTSIZ = 7,            ! Size of the Data Symbol RST Entry and
                                        !      the Type Component RST Entry
        RST$K_OLENTSIZ = 6,             ! Size of the Overloaded Symbol RST Entry
        RST$K_TYPENTSIZ = 11,           ! Size of the Data Type RST Entry
        RST$K_VARENTSIZ = 6,            ! Size of the Variant Set RST Entry
        RST$K_INVENTSIZ = 7;            ! Size of Invocation Number RST Entry
```

RST ENTRY FOR A DATA TYPE COMPONENT

The RST entry for a data type component has the format shown here. A "component" in this sense is a record component, one of a set of variants, or one of the enumeration literals of an enumeration type. The interpretation of a data type component is thus dependent on the "type kind" of the containing data type.

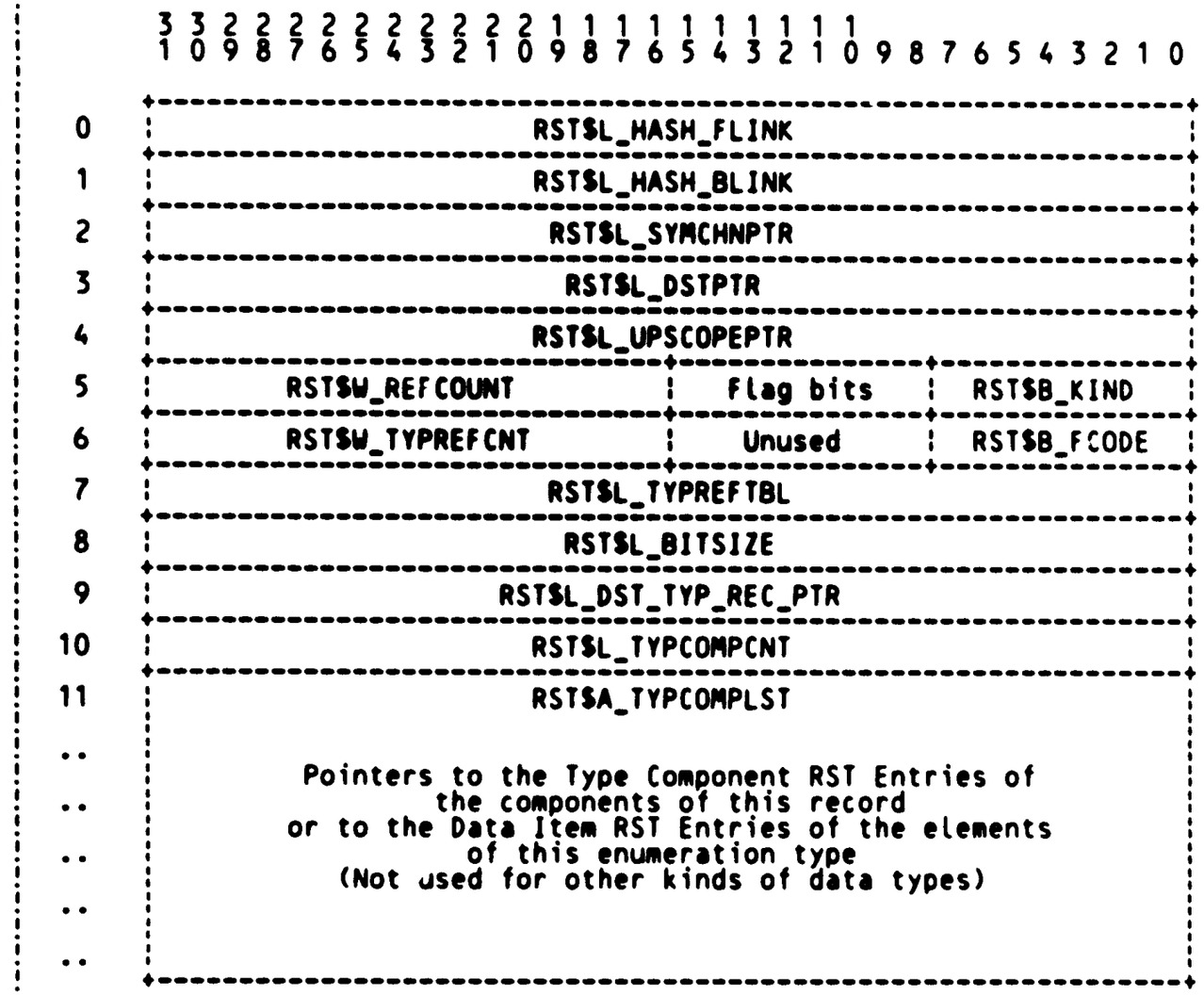
The up-scope pointer always points to the RST entry for the data type to which this is a component. Data Type Component RST Entries are always built when the RST for the whole module is built.

Like the Data Item RST Entry, the this RST entry also has a type pointer which specifies the data type of the component. This is zero for simple types or a pointer to a Data Type RST Entry for a more complex data type (such as records).



RST ENTRY FOR A DATA TYPE

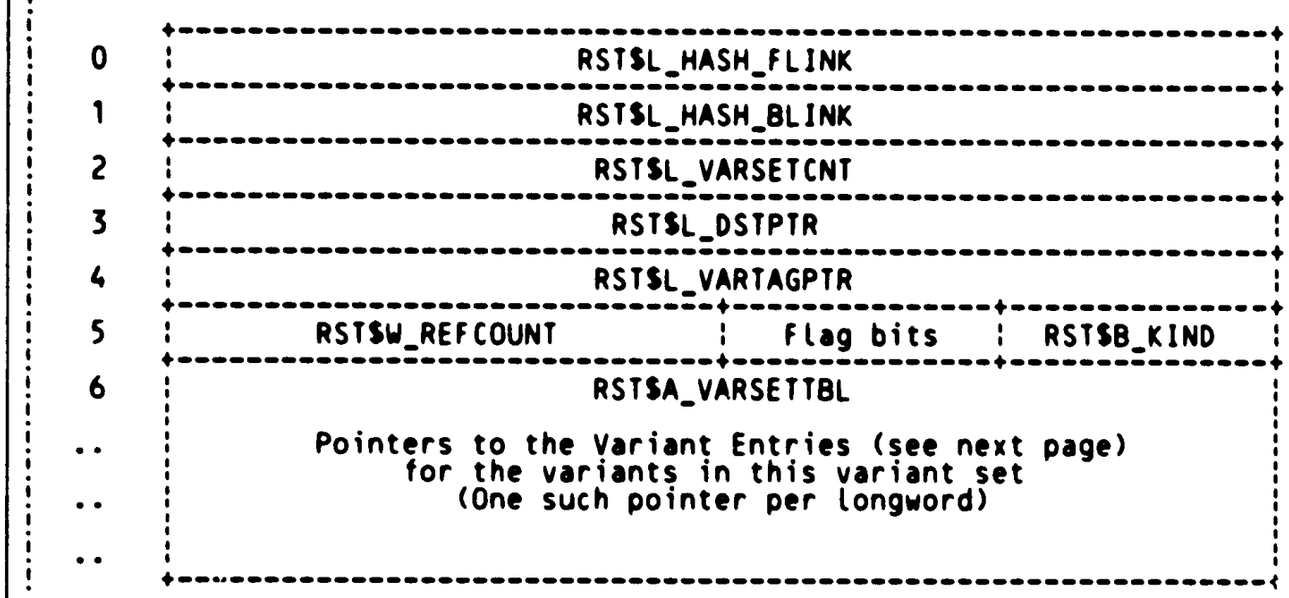
The RST entry for a data type has the format shown here. Not all data types have RST entries--those described by one-byte type codes or standard descriptors do not. Data Type RST Entries are used for all record, variant, and enumeration type data types, however. The up-scope pointer always points to the RST entry for the lexical entity (module, routine, or lexical block) in which the data type was declared. A data type need not have a name, however--it can be anonymous as it is in a PL/I or COBOL record declaration. Data Type RST Entries for records, variants, and enumeration types are built when the RST as a whole is built. All other Data Type RST Entries are built only when needed.



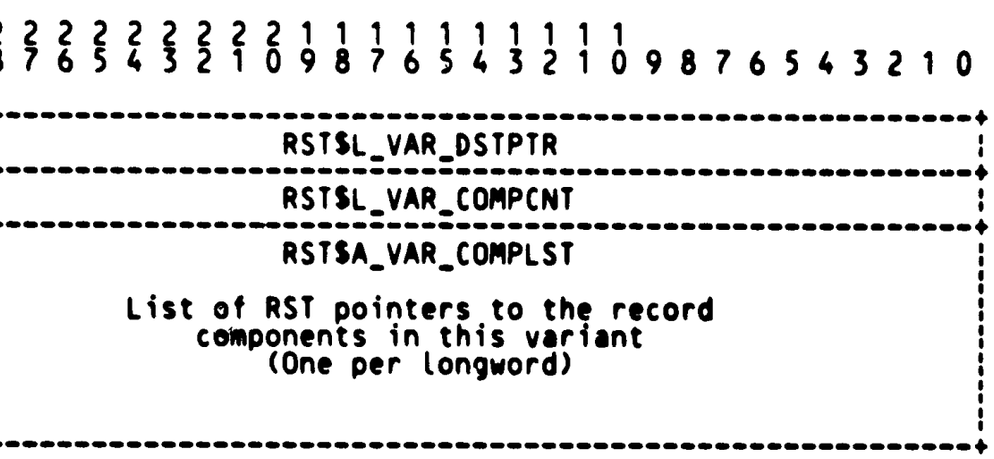
```
!***
!
! These are the possible values for the RST$B_FCODE field.
!
LITERAL
        RST$K_TYPE_MINIMUM = 1,          ! --- Minimum possible FCODE value
        RST$K_TYPE_ARRAY          = 1,   ! Array type
        RST$K_TYPE_ATOMIC         = 2,   ! Atomic VAX standard type
        RST$K_TYPE_DESCR          = 3,   ! VAX standard descriptor type
        RST$K_TYPE_ENUM           = 4,   ! Enumeration type
        RST$K_TYPE_PICT           = 5,   ! Picture type (as in Cobol and PL/I)
        RST$K_TYPE_TPTR           = 6,   ! Typed pointer type
        RST$K_TYPE_RECORD         = 7,   ! Record data type
        RST$K_TYPE_SET            = 8,   ! Set type
        RST$K_TYPE_SUBRNG         = 9,   ! Subrange data type
        !                         = 10,  ! Unused--available for future use
        !                         = 11,  ! Unused--available for future use
        RST$K_TYPE_COBHACK        = 12,  ! Cobol Hack data item
        RST$K_TYPE_BLIDATA        = 13,  ! Bliss data item
        RST$K_TYPE_BLIFLD         = 14,  ! Bliss field
        RST$K_TYPE_FILE           = 15,  ! File data type (as in Pascal)
        RST$K_TYPE_PTR            = 16,  ! Untyped pointer data type
        RST$K_TYPE_AREA           = 17,  ! Area type
        RST$K_TYPE_OFFSET         = 18,  ! Offset type
        RST$K_TYPE_VARIANT        = 19,  ! Variant Set (as in Pascal and ADA)
        RST$K_TYPE_RFA            = 20,  ! Record file address type
        RST$K_TYPE_SELF_REL_LAB   = 21,  ! Self relative label
        RST$K_TYPE_TASK           = 22,  ! Task type (as in ADA)
        RST$K_TYPE_MAXIMUM = 22;         ! --- Maximum possible FCODE value
!**-
```

RST ENTRY FOR A RECORD VARIANT SET

The RST entry for a record variant set has the format shown here. A record variant set is the set of variants of some data record which are distinguished by the same tag variable. This RST entry does not appear free-standing--it is always built inside the memory block allocated for the containing record's Data Type RST Entry. Its hash links, symbol chain pointer, and up-scope pointers are thus not used. Its DST pointer points to the variant set's Variant-Set Begin DST record. The entry has extra fields which point to the tag variable's RST entry and give the list of individual record variants within the set.



This is the format of a Variant Entry as pointed to by the pointers in the Variant-Set RST Entry. This entry gives the list of record components which comprise this particular record variant. It also gives a pointer to the Variant-Value DST record for this variant.



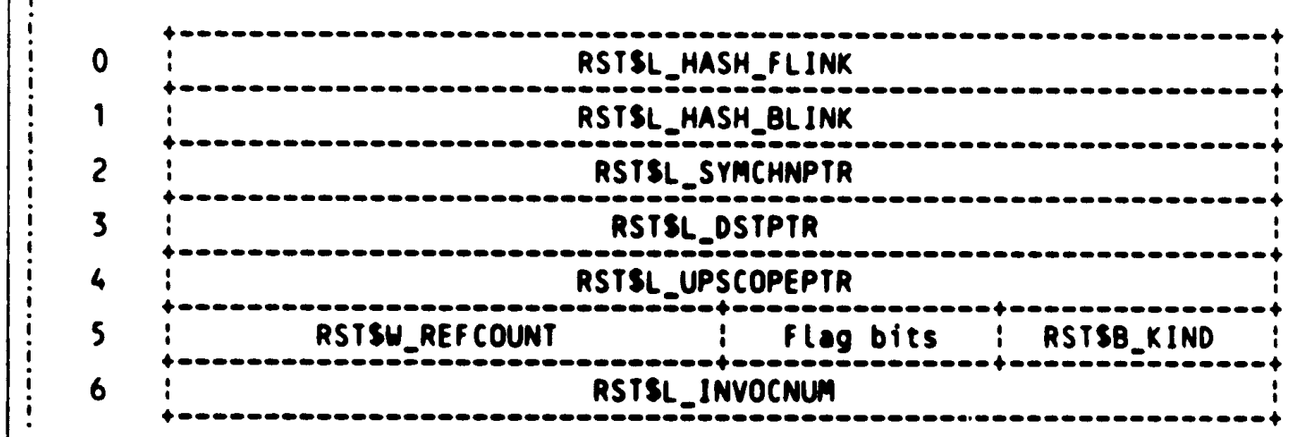
Declarations for the fields in the Variant Entry. Also a declaration macro.

```
FIELD RST$FLD_VARENT =
        SET
        RST$L_VAR_DSTPTR  = [ 0, L_ ],  ! Pointer to Variant-Value DST record
        RST$L_VAR_COMPCNT = [ 1, L_ ],  ! Number of components in this variant
        RST$A_VAR_COMPLST = [ 2, A_ ]   ! Start of component RST pointer list
        TES;

MACRO
        RST$VAR_ENTRY = BLOCK[,LONG] FIELD(RST$FLD_VARENT) %; ! Declaration macro
```

```
!       These definitions have something to do with variant record tag
!       variables.  Unfortunately, no comments were put in to say exactly
!       what they are used for.
!
!***
!
! Define something or another.
!
FIELD RST$TAG_BLOCK_FIELDS =
        SET
        RST$L_TAG_NUMVALS    = [ 0, L_ ],! Number of values to follow (1 or 2)
        RST$L_TAG_LOWBOUND   = [ 1, L_ ],! Lower bound value
        RST$L_TAG_HIGHBOUND  = [ 2, L_ ] ! Upper bound value
        TES;

LITERAL
        RST$K_TAG_BLOCK_SIZE = 3;       ! The size of the tag variable block

MACRO
        RST$TAG_LIST = BLOCKVECTOR[,RST$K_TAG_BLOCK_SIZE]
                                        FIELD(RST$TAG_BLOCK_FIELDS) %;
!**-
```

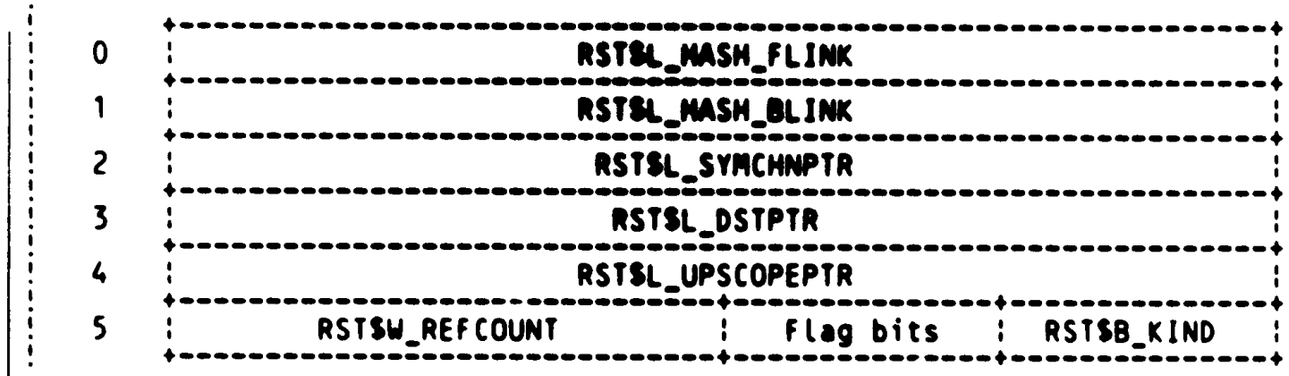
RST ENTRY FOR AN INVOCATION NUMBER

The RST entry for an invocation number has the format shown here. This RST entry always follows the RST entry to which the invocation number belongs on the module's Symbol Chain. If a Data Item RST Entry, for example, has an associated invocation number, then its RST$V_INVOCNUM bit is set and its RST$L_SYMCHNPTR field points to an Invocation Number RST Entry. The Invocation Number RST Entry then specifies the desired invocation number (which is assumed to apply to the innermost routine in the scope of the symbol's declaration). The RST$L_UPSCOPEPTR field in the Invocation Number RST Entry points to the original RST entry of the object (usually data item) without an invocation number. The format of the Invocation Number RST Entry is as follows:



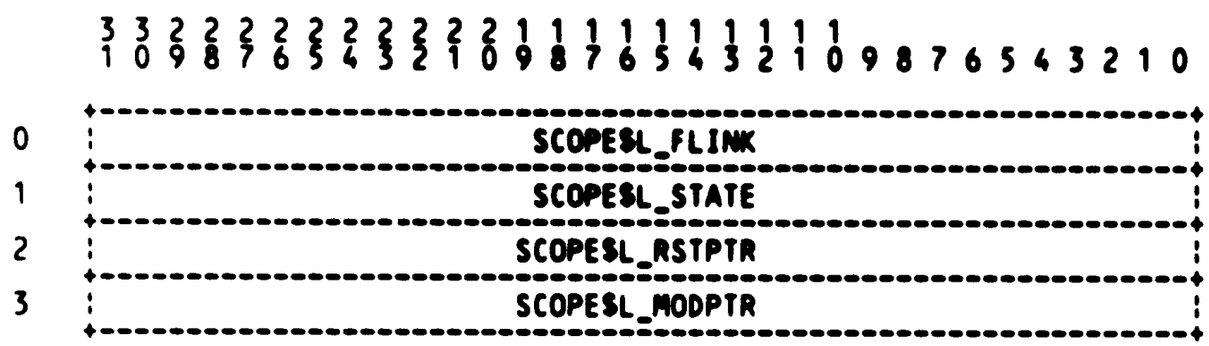
RST ENTRY FOR AN OVERLOADED SYMBOL

The RST entry for an Overloaded Symbol has the format shown below. An Overloaded Symbol RST Entry is created for each Overloaded Symbol DST record (DST$K_OVERLOAD). The RST entry is hashed normally so that DBG$STA_GETSYMBOL can find it.



S C O P E L I S T D E F I N I T I O N S

The scope list maintains the list of scopes declared with the SET SCOPE command. It is a singly linked list pointed to by the global variable SCOPE$LIST in module RSTCNTRL. Each scope entry on the list has this format:



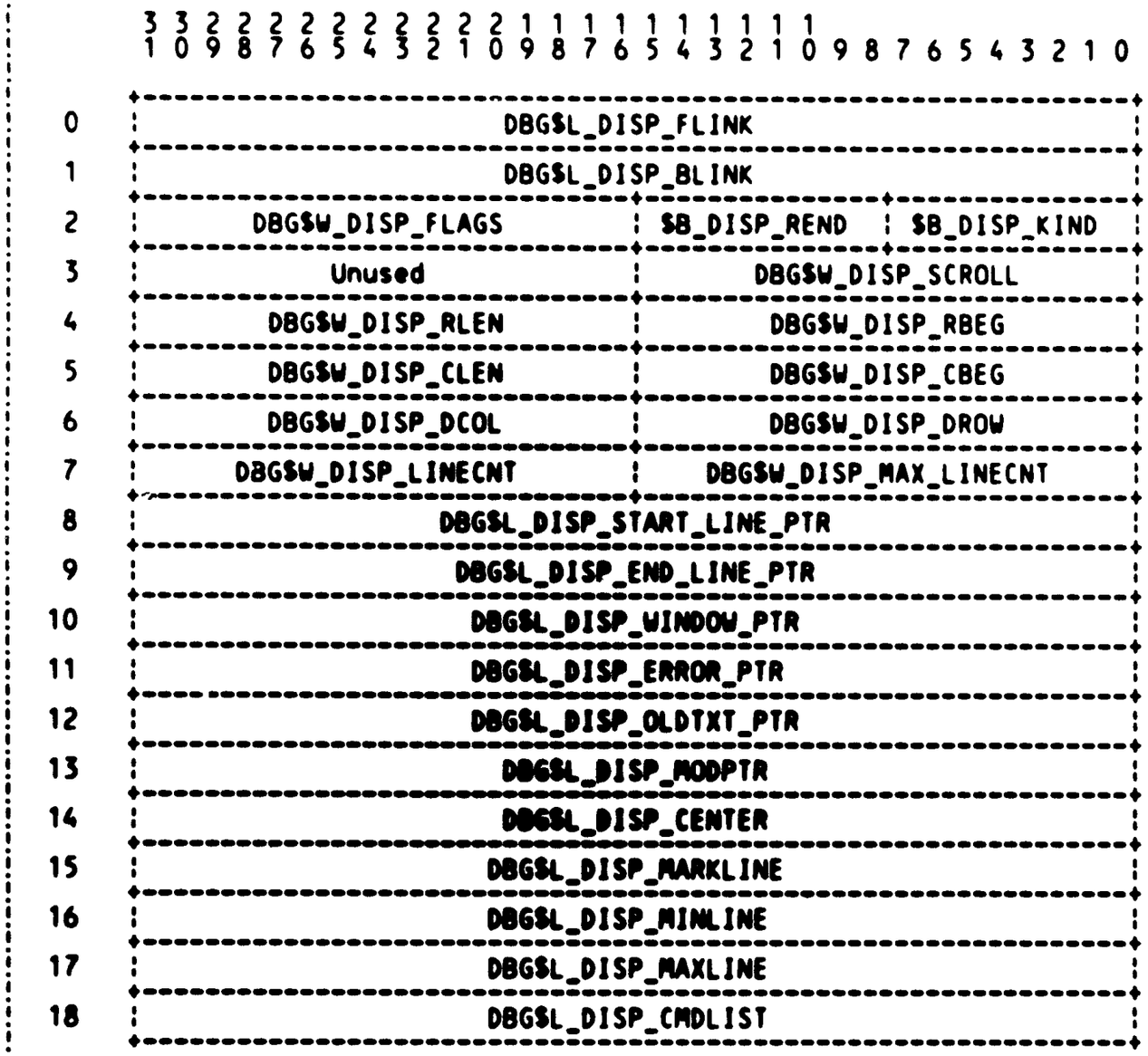
A pointer to a scope list entry is declared as follows:

```
SCOPEPTR: REF SCOPE$ENTRY
```

```
! Declare the scope entry fields and the definition macro.

FIELD SCOPE$FLD_DEF =
        SET
        SCOPE$L_FLINK    = [ 0, L_ ],     ! Scope list forward link or zero
        SCOPE$L_STATE    = [ 1, L_ ],     ! The scope "state" or kind
        SCOPE$L_RSTPTR   = [ 2, L_ ],     ! Pointer to scope's own RST entry
        SCOPE$L_MODPTR   = [ 3, L_ ]      ! Pointer to scope's Module RST Entry
        TES;

LITERAL
        SCOPE$K_ENTSIZE = 4;              ! Size of scope entry in longwords

MACRO
        SCOPE$ENTRY = BLOCK[SCOPE$K_ENTSIZE] FIELD(SCOPE$FLD_DEF) %;


! These are the possible values of the SCOPE$L_STATE field.  Each of these val-
! ues indicates what kind of scope is to be searched to match a given pathname.

LITERAL
        SCOPE$K_NORMAL  = 1,              ! Normal named scope
        SCOPE$K_NUMBERED= 2,              ! Numbered scope (PC in CALL stack)
        SCOPE$K_GLOBAL  = 3,              ! Global Symbol Table
        SCOPE$K_SETMODS = 4;              ! All SET modules
```

T H E S C R E E N D I S P L A Y E N T R Y

The Screen Display Entry contains all information needed to represent and output a screen display on the terminal screen. All Display Entries are linked together by forward and backward links. Each Display Entry contains all information about where the corresponding display is placed on the screen, what attributes it has, and what the contents (the text) of the display is. The text is represented by pointers to the Screen Display Line Entries for the text lines that make up the display.



19	DBG$A_DISP_NAME
..	The display name in Counted ASCII
..	
..	

All Screen Display Entries are linked together on a doubly linked list for which the OWN block DBG$SCR_DISPLAY_LIST is the list head. Also, the DBG$L_DISP_START_LINE_PTR and DBG$L_DISP_END_LINE_PTR fields of the Display Entry constitute the list head for the doubly linked list of Screen Display Line Entries associated with this display.

A pointer to a Display Entry is declared as follows:

```
DISP_PTR: REF DBG$DISP_ENTRY
```

Define the fields of the Screen Display Entry.

```
FIELD DBG$DISP_FLD =
        SET
        DBG$L_DISP_FLINK   = [ 0, L_ ],  ! Screen display list forward link
        DBG$L_DISP_BLINK   = [ 1, L_ ],  ! Screen display list backward link
        DBG$B_DISP_KIND    = [ 2, B0_ ], ! Screen display kind
        DBG$B_DISP_REND    = [ 2, B1_ ], ! Screen display rendition bits
        DBG$V_DISP_REND_BLD = [ 2, V1_(0) ],    ! Bolding rendition bit
        DBG$V_DISP_REND_RV  = [ 2, V1_(1) ],    ! Reverse-video rendition bit
        DBG$V_DISP_REND_BLK = [ 2, V1_(2) ],    ! Blinking rendition bit
        DBG$V_DISP_REND_UND = [ 2, V1_(3) ],    ! Underline rendition bit
        DBG$W_DISP_FLAGS   = [ 2, W1_ ], ! Screen display flag bits
        DBG$V_DISP_REMOVE    = [ 2, V2_(0) ],   ! Remove display from pasteboard
        DBG$V_DISP_INVSCR    = [ 2, V2_(1) ],   ! Invalidate DISP_SCROLL field
        DBG$V_DISP_ATTOP     = [ 2, V2_(2) ],   ! Display at top of source file
        DBG$V_DISP_ATBOT     = [ 2, V2_(3) ],   ! Display at bottom of source file
        DBG$V_DISP_MARKFLG   = [ 2, V2_(4) ],   ! Mark changed lines in display
        DBG$V_DISP_NEWDISP   = [ 2, V2_(5) ],   ! New display--suppress marking
        DBG$W_DISP_SCROLL  = [ 3, SW0_ ],! Screen display scrolling count
        !                  = [ 3, W1_ ], ! Unused (available for future use)
        DBG$W_DISP_RBEG    = [ 4, W0_ ], ! Screen window beginning row location
        DBG$W_DISP_RLEN    = [ 4, W1_ ], ! Screen window row length (height)
        DBG$W_DISP_CBEG    = [ 5, W0_ ], ! Screen window beginning column location
        DBG$W_DISP_CLEN    = [ 5, W1_ ], ! Screen window column length (width)
        DBG$W_DISP_DROW    = [ 6, W0_ ], ! Screen window display row location
        DBG$W_DISP_DCOL    = [ 6, W1_ ], ! Screen window display column location
        DBG$W_DISP_MAX_LINECNT           ! Screen display's maximum line count
                           = [ 7, W0_ ], !      (maximum lines saved in memory)
        DBG$W_DISP_LINECNT= [ 7, W1_ ],  ! Screen display's actual line count
        DBG$L_DISP_START_LINE_PTR        ! Pointer to first line of display text
                           = [ 8, L_ ],  !      (points to display line entry)
```

```
        DBG$L_DISP_END_LINE_PTR         ! Pointer to last line of display text
                        =[ 9, L_ ],     !      (points to display line entry)
        DBG$L_DISP_WINDOW_PTR           ! Pointer to first line in screen window
                        = [ 10, L_ ],   !      (points to a Display Line Entry)
        DBG$L_DISP_ERROR_PTR            ! Pointer to list of error message
                        = [ 11, L_ ],   !      Display Line Entries for display
        DBG$L_DISP_OLDTXT_PTR           ! Pointer to list of old Display Line
                        = [ 12, L_ ],   !      Entries--used to mark changes
        DBG$L_DISP_MODPTR = [ 13, L_ ], ! Pointer to Module RST Entry--used for
                                        !      source line displays only
        DBG$L_DISP_CENTER = [ 14, L_ ], ! Central line number for source display
        DBG$L_DISP_MARKLINE             ! Line number to be marked with "->" in
                        = [ 15, L_ ],   !      source line display (or -1)
        DBG$L_DISP_MINLINE= [ 16, L_ ], ! Minimum source line number in module
        DBG$L_DISP_MAXLINE= [ 17, L_ ], ! Maximum source line number in module
        DBG$L_DISP_CMDLIST= [ 18, L_ ], ! Pointer to display's DEBUG command
                                        !      list entry (or zero)
        DBG$A_DISP_NAME   = [ 19, A_ ]  ! The display's name in Counted ASCII
        TES;

MACRO
        DBG$DISP_ENTRY = BLOCK[,LONG] FIELD(DBG$DISP_FLD) %;

LITERAL
        DBG$K_DISP_ENTSIZE      = 19;   ! Size of the fixed part of the Screen
                                        !      Display Entry in longwords

! Define the literals used to indicate the display kind.  The display kind
! determined how the contents of the display are generated.
!
LITERAL
        DBG$K_DISP_MINKIND      = 0,    ! ---Minimum kind value
        DBG$K_DISP_NOKIND       = 0,    ! No kind--used in calls to indicate no
                                        !      change to the display's kind
        DBG$K_DISP_NORMAL       = 1,    ! Normal display kind--contents deter-
                                        !      mined by direct user input
        DBG$K_DISP_DO           = 2,    ! Contents specified by DO command list
        DBG$K_DISP_SOURCE       = 3,    ! Source display with contents speci-
                                        !      fied by a source command list
        DBG$K_DISP_REGISTER     = 4,    ! Contents is machine register display
        DBG$K_DISP_MAXKIND      = 4;    ! ---Maximum kind value

! Define literals for the display rendition bits.
!
LITERAL
        DBG$M_DISP_REND_BLD     = 1,    ! Bolding rendition mask
        DBG$M_DISP_REND_RV      = 2,    ! Reverse-video rendition mask
        DBG$M_DISP_REND_BLK     = 4,    ! Blinking rendition mask
        DBG$M_DISP_REND_UND     = 8;    ! Underline rendition mask

! Define the literals used to indicate the scrolling direction when scrolling
! a screen display.
!
```

```
LITERAL
        DBG$K_SCROLL_UP         = 1,    ! Scroll the display up
        DBG$K_SCROLL_DOWN       = 2,    ! Scroll the display down
        DBG$K_SCROLL_LEFT       = 3,    ! Scroll the display to the left
        DBG$K_SCROLL_RIGHT      = 4;    ! Scroll the display to the right
```

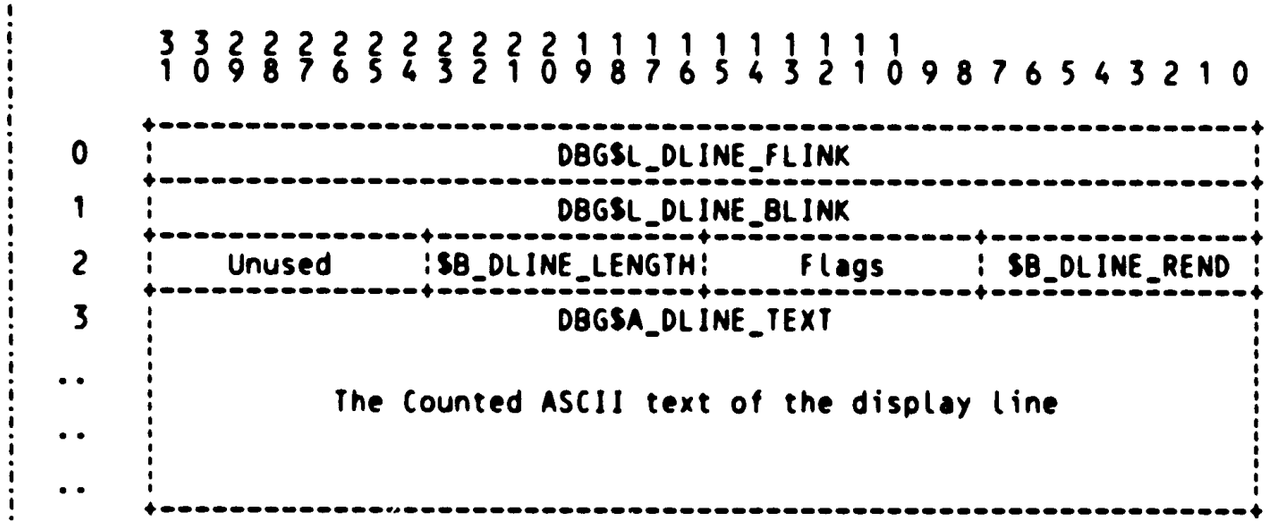
T H E S C R E E N D I S P L A Y L I N E E N T R Y

The Screen Display Line Entry holds the text and other attributes of a line within a screen display. This entry contains forward and backward links which link this Line Entry to the Line Entries for the previous line and the next line of the same display. The Line Entry also contains the line's rendition attributes (such as reverse video, blinking, etc.) which are to be passed to the Screen Management Package. Finally it contains the actual text of the line, including the length of that text, and the length of the text buffer.

Screen Display Line Entries come in two variants, the Normal and the Source Display Line Entry. Source Display Line Entries are used for source displays as the name implies, and Normal Display Line Entries are used for all other kinds of displays. Normal Display Line Entries can optionally include rendition information on a per-character basis.

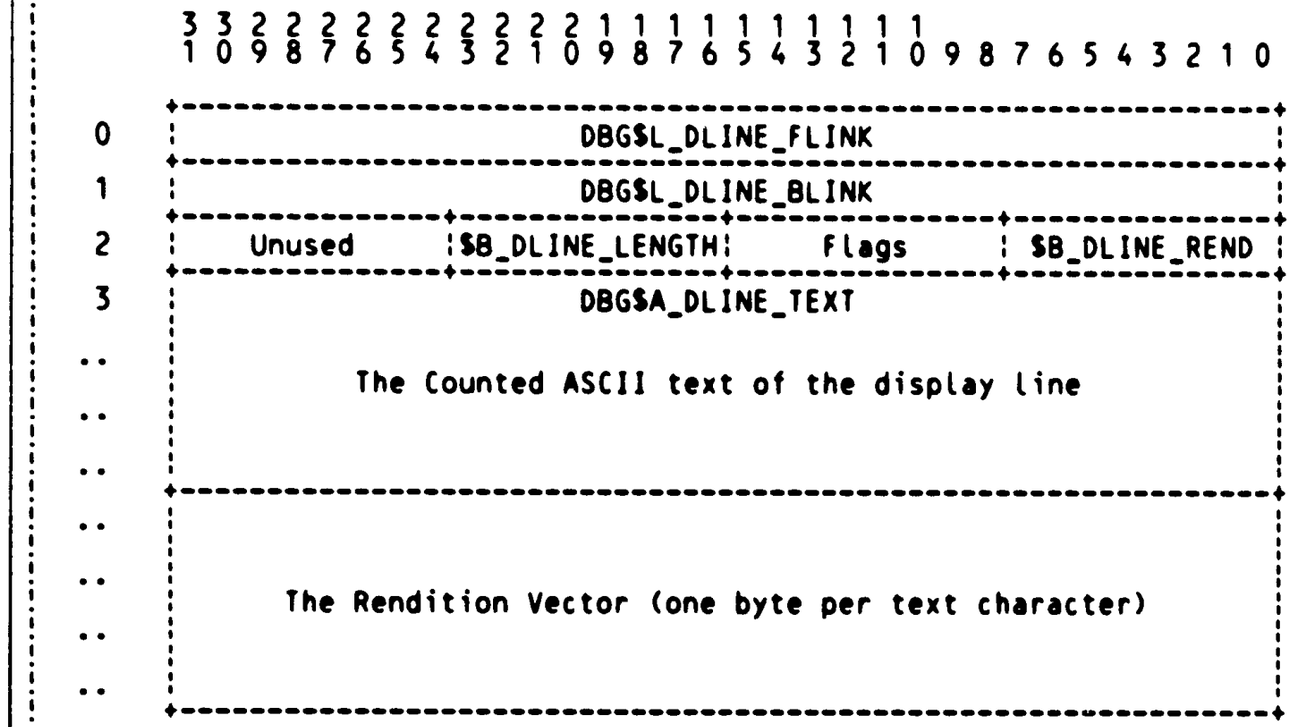
NORMAL DISPLAY LINE ENTRY

The Normal Display Line Entry is used for all screen displays except Source screen displays. This is the format of the Normal Screen Display Line Entry:



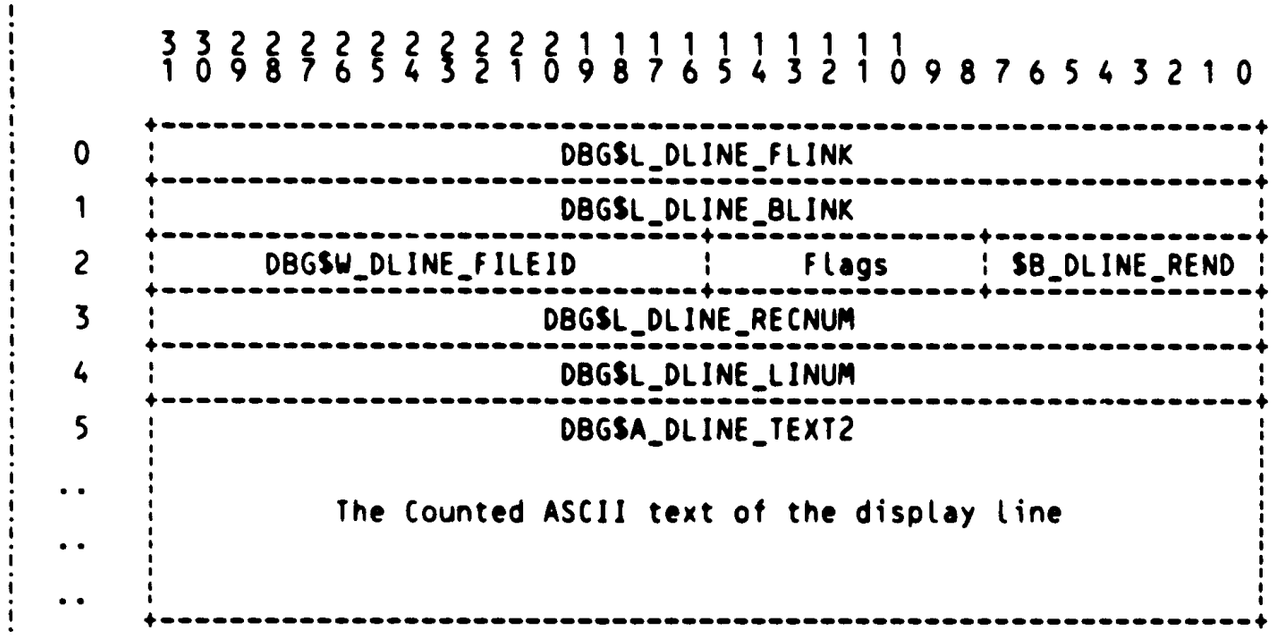
NORMAL DISPLAY LINE ENTRY WITH RENDITION VECTOR

The Normal Display Line Entry may include a "rendition vector" if the screen rendition is not the same for the entire line. In this case, the DBG$V_DLINE_RENDFLG is set in the Display Line Entry's flag field and a "rendition vector" is present at the end of the Display Line Entry, immediately following the Counted ASCII text of the display line. The rendition vector contains one byte of rendition codes for each character in the Counted ASCII text of the line. The rendition vector has the same length as the Counted ASCII text.



SOURCE DISPLAY LINE ENTRY

The Source Display Line Entry is used for source displays only, and contains extra line number information which is not needed for other kinds of displays. The DBG$V_DLINE_SOURCEFLG bit is always set in a Source Display Line Entry. This is the format of the Source Display Line Entry:



Each Display Line Entry is part of a doubly linked list of such entries associated with some display. The fields DBG$L_DISP_START_LINE_PTR and DBG$L_DISP_END_LINE_PTR in the display's Screen Display Entry constitute the list head for this list of Display Line Entries.

The Display Line Entries associated with a display are reused as new text is added to the display contents. For this reason, each entry contains not only the length of the current text line but the length of the entry's text buffer as well. The text buffer will be longer than the current text line if it contained a longer text line earlier.

A pointer to the Screen Display Line Entry is declared as follows:

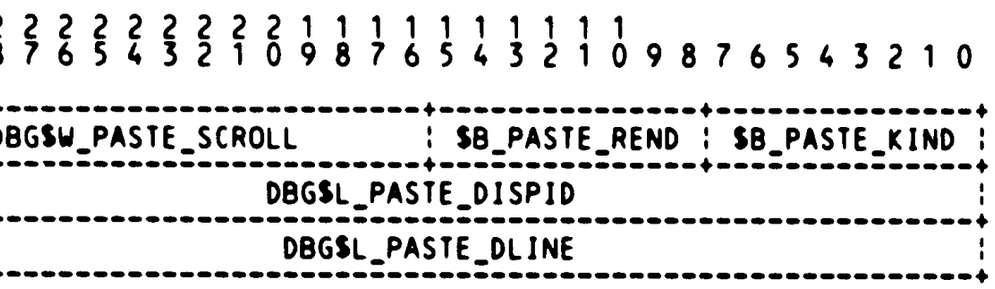
```
DLINE_PTR: REF DBG$DLINE_ENTRY
```

Define the fields of the Screen Display Line Entry.

```
FIELD DBG$DLINE_FLD =
        SET
        DBG$L_DLINE_FLINK = [ 0, L_ ],  ! Forward link to next line entry
        DBG$L_DLINE_BLINK = [ 1, L_ ],  ! Backward link to previous line entry
        DBG$B_DLINE_REND  = [ 2, B0_ ], ! Rendition bits for this line
        DBG$V_DLINE_REND_BLD = [ 2, V0_(0) ],   ! Bolding rendition bit
        DBG$V_DLINE_REND_RV  = [ 2, V0_(1) ],   ! Reverse-video rendition bit
        DBG$V_DLINE_REND_BLK = [ 2, V0_(2) ],   ! Blinking rendition bit
        DBG$V_DLINE_REND_UND = [ 2, V0_(3) ],   ! Underline rendition bit
        !                    = [ 2, B1_ ], ! The flag byte
        DBG$V_DLINE_RENDFLG  = [ 2, V1_(0) ],   ! Line includes rendition vector
        DBG$V_DLINE_SOURCEFLG= [ 2, V1_(1) ],   ! Set for Source Disp Line Entry
        !
        DBG$B_DLINE_LENGTH= [ 2, B2_ ], ! The length of the text buffer (bytes)
        DBG$A_DLINE_TEXT  = [ 3, A_ ],  ! The line's text in Counted ASCII
        !
        DBG$W_DLINE_FILEID= [ 2, W1_ ], ! Source File ID of source line
        DBG$L_DLINE_RECNUM= [ 3, L_ ],  ! Source file record number of line
        DBG$L_DLINE_LINUM = [ 4, L_ ],  ! Line number of source line
        DBG$A_DLINE_TEXT2 = [ 5, A_ ]   ! Source line's text in Counted ASCII
        TES;

MACRO
        DBG$DLINE_ENTRY = BLOCK[,LONG] FIELD(DBG$DLINE_FLD) %;

LITERAL
        DBG$K_DLINE_ENTSIZE     = 3,    ! Size of fixed part of Normal Display
                                        !      Line Entry in longwords
        DBG$K_DLINE_ENTSIZE2    = 5;    ! Size of fixed part of Source Display
                                        !      Line entry in longwords
```

T H E S C R E E N P A S T E B O A R D E N T R Y

The screen pasteboard, on which all displays to be output on the terminal are pasted to determine what occludes what, is represented by a blockvector. The vector is indexed by line number and has one entry (block) for each line on the screen. The length of the vector is actually 21 (not 24) so that the last three lines on the screen are always reserved for user program output and DEBUG input prompting. For each line, the vector contains one Pasteboard Entry of the following format:

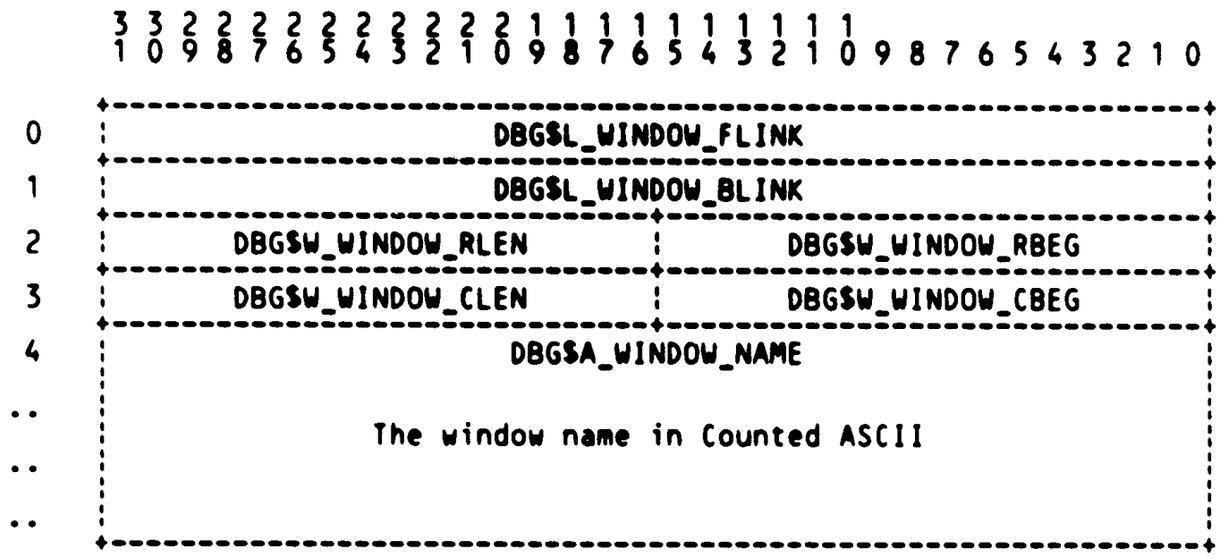



```
! Define the fields of the individual Pasteboard Entry.  Also declare the
! declaration macro and the size of the vector and of each block.

FIELD DBG$PASTE_FLD =
        SET
        DBG$B_PASTE_KIND    = [ 0, B0_ ],! The kind of line this entry holds
        DBG$B_PASTE_REND    = [ 0, B1_ ],! The rendition bits for this line
        DBG$W_PASTE_SCROLL = [ 0, SWT_],! The scrolling amount for this line
        DBG$L_PASTE_DISPID = [ 1, L_ ], ! Pointer to Display Entry for line
        DBG$L_PASTE_DLINE  = [ 2, L_ ]  ! Pointer to Display Line Entry
        TES;

MACRO
        DBG$PASTEBOARD =
                BLOCKVECTOR[DBG$K_PASTE_SIZE, DBG$K_PASTE_ENTSIZE, LONG]
                FIELD(DBG$PASTE_F[D) %;

LITERAL
        DBG$K_PASTE_SIZE    = 21,       ! Number of lines in pasteboard
        DBG$K_PASTE_ENTSIZE = 3;        ! Size of one Pasteboard Entry in
                                        !      longwords

! Define the allowed values of the DBG$B_PASTE_KIND field.

LITERAL
        DBG$K_PASTE_NULL        = 1,    ! Null line--no display covers it
        DBG$K_PASTE_TEXT        = 2,    ! Text line from a display
        DBG$K_PASTE_BLANK       = 3,    ! Blank line within a display
```

```
        DBG$K_PASTE_LABEL       = 4,    ! Top border line with label
        DBG$K_PASTE_BORDER      = 5;    ! Bottom border line on screen
```

T H E S C R E E N W I N D O W E N T R Y

The Screen Window Entry contains all the window parameters associated with a named screen window. It also contains the actual window name. All Screen Window Entries are linked together on a doubly linked list for which the OWN block DBG$SCR_WINDOW_LIST is the list head.



A pointer to a Screen Window Entry is declared as follows:

```
WPTR: REF DBG$WINDOW_ENTRY
```

```
! Define the fields of the Screen Window Entry.

FIELD DBG$WINDOW_FLD =
        SET
        DBG$L_WINDOW_FLINK = [ 0, L_ ], ! Forward link to next Window Entry
        DBG$L_WINDOW_BLINK = [ 1, L_ ], ! Backward link to previous Window Entry
        DBG$W_WINDOW_RBEG  = [ 2, W0_ ],! Beginning row location on screen
        DBG$W_WINDOW_RLEN  = [ 2, W1_ ],! Row length (height) of window
        DBG$W_WINDOW_CBEG  = [ 3, W0_ ],! Beginning column location on screen
        DBG$W_WINDOW_CLEN  = [ 3, W1_ ],! Column length (width) of window
        DBG$A_WINDOW_NAME  = [ 4, A_ ]  ! The window's name in Counted ASCII
        TES;

MACRO
        DBG$WINDOW_ENTRY = BLOCK[,LONG] FIELD(DBG$WINDOW_FLD) %;

LITERAL
        DBG$K_WINDOW_ENTSIZE    = 4;    ! The size of the fixed part of the
```

```
!       Screen Window Entry in longwords
```

S O U R C E D I R E C T O R Y S E A R C H L I S T

The Source Directory Search List is a list structure which keeps track of all source directory names specified by the user via SET SOURCE and SET SOURCE/MODULE=xxx commands. This structure is thus searched when a source file must be opened so that the file is opened in the proper directory.

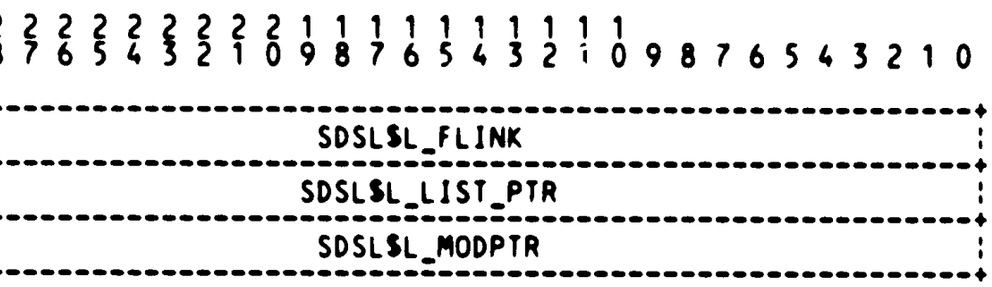
The structure of this list is as follows. Variable DBF$SRC_DIR_LIST in module DBGSOURCE points to a singly linked list of Source Directory Search List Header Blocks. Each Header Block contains a Module RST Entry pointer (which may be zero) and points to a singly linked list of Source Directory Search List Entries. Each such entry contains a directory name as a counted ASCII string.

The list is searched by first searching all the Header Blocks for the Module RST Entry pointer for the current module, i.e. the module from which source lines are to be displayed. If the desired module is not found, the Header Block with the zero Module RST Entry pointer is used instead. The found Header Block then points to the linked list of directory names to use when searching for the desired source file. If there is no Header Block with either the desired module pointer or the zero module pointer, no Source Directory Search List Entries are used--the file name from the Declare Source File DST command is used as is.

The command SET SOURCE/MODULE=modname dir1,...,dirN is represented by a Header Block with a pointer to the Module RST Entry for module "modname" and a pointer to a linked list of Entries. There is one Entry for each of dir1, ..., dirN. The command SET SOURCE dir1,...,dirN is represented by a Header Block with a zero Module RST Entry pointer and of course a pointer to a linked list of Entries for dir1, ..., dirN.

SOURCE DIRECTORY SEARCH LIST HEADER BLOCK

The Source Directory Search List Header Block, as described on the previous page, has the following format:



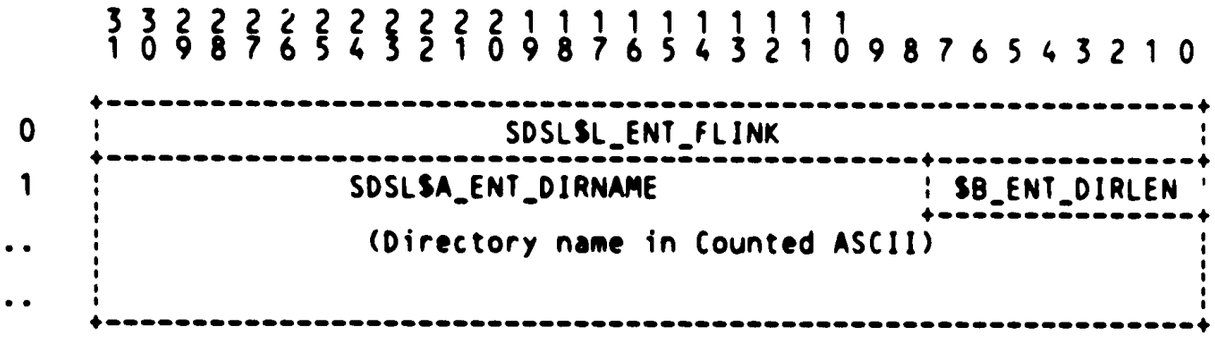
A pointer to a Source Directory Search List Header Block is declared as follows:

```
SDSL_PTR: REF SDSL$HEADER
```

```
! Declare the fields of the Source Directory Search List Header Block.  Also
! declare the declaration macro.
!
FIELD SDSL$FLD_HEADER =
        SET
        SDSL$L_FLINK     = [ 0, L_ ],    ! Forward link to next Header Block
        SDSL$L_LIST_PTR  = [ 1, L_ ],    ! Pointer to list of SDSL Entries
        SDSL$L_MODPTR    = [ 2, L_ ]     ! Pointer to Module RST Entry of module
                                         !      to which search list applies; is
                                         !      zero if for all other modules
        TES;

LITERAL
        SDSL$K_HDR_SIZE = 3;             ! Size of SDSL Header Block in longwords

MACRO SDSL$HEADER = BLOCK[SDSL$K_HDR_SIZE] FIELD(SDSL$FLD_HEADER) %;
```

SOURCE DIRECTORY SEARCH LIST ENTRY

There is one Source Directory Search List Entry for each directory name specified on a SET SOURCE command. The "directory name" may in fact contains a full file name, including extension and version number, but in most cases it is used only to override the directory name as given in the Declare Source File DST command. This is the format of a Source Directory Search List Entry:



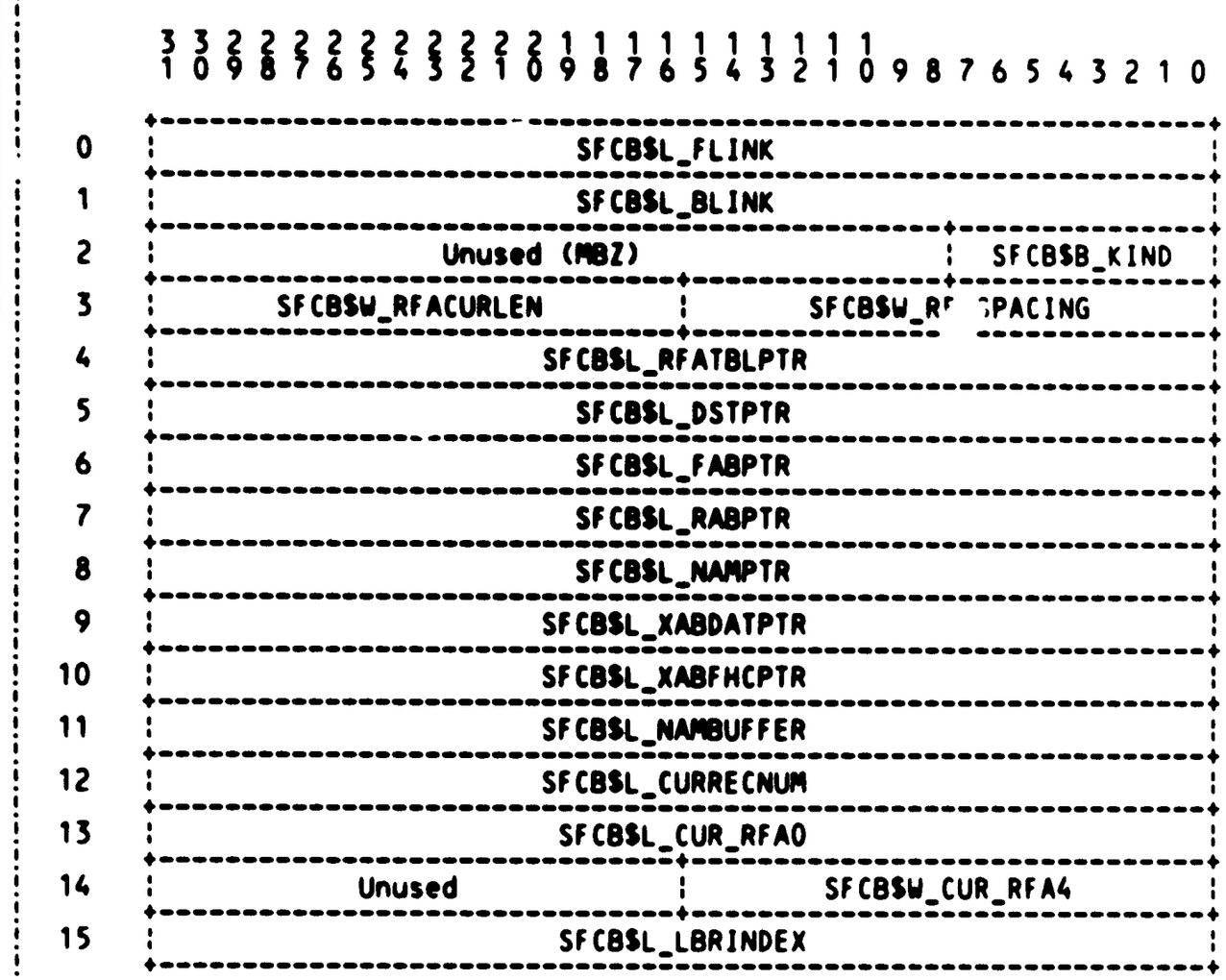
A pointer to a Source Directory Search List Entry is declared like this:

```
SDSL_ENT_PTR: REF SDSL$ENTRY
```

```
! Declare the fields of the Source Directory Search List Entry.  Also declare
! the declaration macro.

FIELD SDSL$FLD_ENTRY =
        SET
        SDSL$L_ENT_FLINK    = [ 0, L_ ], ! Forward link to next SDSL Entry
        SDSL$B_ENT_DIRLEN   = [ 1, B0_ ],! Length in characters of directory name
        SDSL$A_ENT_DIRNAME  = [ 1, A1_ ] ! First character of directory name
        TES;

LITERAL
        SDSL$K_ENT_SIZE = 1;             ! Size of fixed portion of SDSL Entry
                                         !      in longwords

MACRO SDSL$ENTRY = BLOCK[,LONG] FIELD(SDSL$FLD_ENTRY) %;
```

S O U R C E F I L E C O N T R O L B L O C K

This is the format of the Source File Control Block:



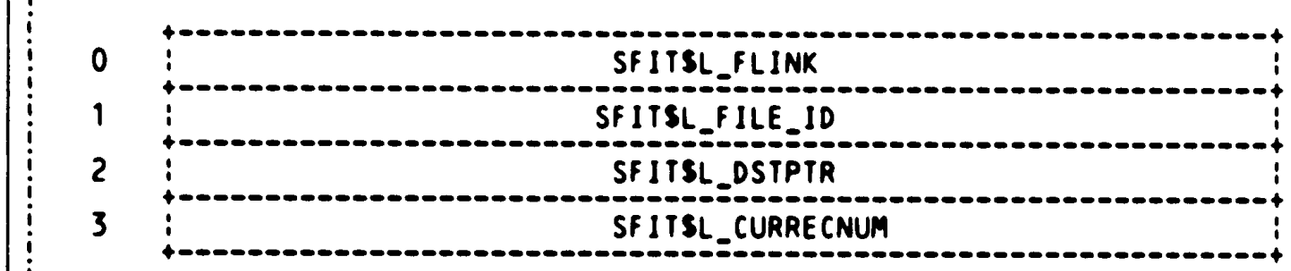
The Source File Control Blocks (SFCBs) keep track of all currently open source files. This includes modules within source libraries as well as normal RMS files. Each such block contains all information needed to read source records from the corresponding file. These blocks form a circular doubly linked list, where variable DBG$SRC_SFCB_PTR in module DBGSOURCE points to the first block on the list. The blocks are always maintained in order so that the Most Recently Used SFCB is first on the list and the Least Recently Used block is last. This ordering allows DEBUG to close the Least Recently Used source file when a file must be closed in order to open a new file.

```
!
!       DEBUG always keeps a fixed number of Source File Control Blocks on the
!       list.  This number is given by variable DBG$SRC_MAX_FILES in module
!       DBGSOURCE.  It can be changed by the user with the SET MAX_SOURCE_FILES
!       command.  This number limits the number of source files which are open
!       at the same time.  Such a limit is required to prevent DEBUG from using
!       up too many of the limited number of channels the user can have open at
!       any one time.  Thus, when all SFCBs are in use and a new source file
!       must be opened, the Least Recently Used file is closed and its SFCB is
!       reused for the new file.
!
!       A pointer to a Source File Control Block is declared as follows:
!
!                       SFCB_PTR: REF SFCB$BLOCK


! Declare the fields of the Source File Control Block.  Also declare the
! declaration macro.

FIELD SFCB$FLD_DEF =
        SET
        SFCB$L_FLINK     = [ 0, L_ ],     ! Forward link to next SFCB
        SFCB$L_BLINK     = [ 1, L_ ],     ! Backward link to previous SFCB
        SFCB$B_KIND      = [ 2, B0_ ],    ! The kind of source file this is
        SFCB$_RFASPACING=[ 3, W0_ ],      ! The allocated length of the RFA table
        SFCB$W_RFACURLEN= [ 3, W1_ ],     ! The used length of the RFA table
        SFCB$L_RFATBLPTR= [ 4, L_ ],      ! Pointer to Record File Address (RFA)
                                          !     table for this source file
        SFCB$L_DSTPTR    = [ 5, L_ ],     ! Pointer to DST Declare Source File
                                          !     command for this source file
        SFCB$L_FABPTR    = [ 6, L_ ],     ! Pointer to the FAB for this file
        SFCB$L_RABPTR    = [ 7, L_ ],     ! Pointer to the RAB for this file
        SFCB$L_NAMPTR    = [ 8, L_ ],     ! Pointer to the NAM block for this file
        SFCB$L_XABDATPTR= [ 9, L_ ],      ! Pointer to the XAB Date and Time block
                                          !     for this file
        SFCB$L_XABFHCPTR= [ 10, L_ ],     ! Pointer to XAB File Header Character-
                                          !     istics block for this file
        SFCB$L_NAMBUFFER= [ 11, L_ ],     ! Pointer to buffer for NAM block file
                                          !     name strings for this file
        SFCB$L_CURRECNUM= [ 12, L_ ],     ! Record number at which the source file
                                          !     is currently positioned
        SFCB$L_CUR_RFA0 = [ 13, L_ ],     ! RFA at which the source file is cur-
                                          !     rently positioned (first 4 bytes)
        SFCB$W_CUR_RFA4 = [ 14, W0_ ],    ! RFA at which the source file is cur-
                                          !     rently positioned (last 2 bytes)
        SFCB$L_LBRINDEX = [ 15, L_ ]      ! Library file index used by librarian
        TES;

LITERAL
        SFCB$K_SIZE      = 16;            ! Size of SFCB in longwords

MACRO SFCB$BLOCK = BLOCK[SFCB$K_SIZE] FIELD(SFCB$FLD_DEF) %;

! Declare the possible values of the SFCB$B_KIND field.
!
```

```
LITERAL
        SFCB$K_NOTUSED  = 1,            ! This SFCB is not used (is available)
        SFCB$K_RMSFILE  = 2,            ! This SFCB is used for an RMS file
        SFCB$K_LBRFILE  = 3,            ! This SFCB is used for a module within
                                        !      a source library
        SFCB$K_FILE_UNAVAIL  = 4;       ! This SFCB is occupied by a file which
                                        !      is presently not available
```

S O U R C E F I L E I D T A B L E

The Source File ID Table is used during the decoding of the Source Line Correlation DST Record to keep track of the File IDs used to reference the current module's source files. The Source File ID Table consists of a singly linked list of Source File ID Table Entries, where each entry gives a File ID, a pointer to the corresponding Declare Source File command, (This command gives all needed information about the identity and nature of the source file.), and the current source record number. This table is used locally in routine DBG$SRC_TYPE_LNUM_SOURCE in module DBGSOURCE. This is the format of each Source File ID Table Entry:



A pointer to a Source File ID Table Entry is declared as follows:

```
SFIT_PTR: REF SFIT$ENTRY
```

Declare the fields of the Source File ID Table Entry. Also declare the declaration macro.

```
FIELD SFIT$FLD_DEF =
        SET
        SFIT$L_FLINK      = [ 0, L_ ],    ! Forward link to next SFIT entry
        SFIT$L_FILE_ID    = [ 1, L_ ],    ! File ID value for this source file
        SFIT$L_DSTPTR     = [ 2, L_ ],    ! Pointer to DST Declare Source File
                                          !      command for this source file
        SFIT$L_CURRECNUM = [3, L_]        ! Current source record number for
                                          !      this source file
        TES;

LITERAL
        SFIT$K_SIZE      = 4;             ! Size of SFIT entry in longwords

MACRO SFIT$ENTRY = BLOCK[SFIT$K_SIZE] FIELD(SFIT$FLD_DEF) %;
```

```
!       S O U R C E   F I L E   R E C O R D   F I L E   A D D R E S S   T A B L E
!
!       This is the format of each Record File Address Table entry:
!
!        3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
!        1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
!       +---------------------------------------------------------------+
!   0   !                        RFATBL$L_RFA0                          !
!       +---------------------------------------------------------------+
!   1                                   !        RFATBL$W_RFA4         !
!                                       +------------------------------+
!
!
!       The Record File Address Table correlates record numbers in a source
!       file with RFA's for that file. It is used during the TYPE command
!       and other commands that cause source lines to be displayed, to
!       help locate the desired record more efficiently.
!
!       There is one RFA table for each open file. The table is allocated
!       during DBG$SRC_INIT and a pointer to the table is placed in the
!       Source File Control Block. The table consists of a fixed number
!       of entries (RFATBL_SIZE_ENTRIES, defined in DBGSOURCE).
!
!       For every N records in the file, there is one RFA entry. That is,
!       the first entry is the RFA for record 1, the next for record (1+N),
!       the next for record (1+2N), and so on. The spacing N can be adjusted
!       dynamically and is kept in the Source File Control Block.
!
!       A pointer to an RFA table is declared as follows:
!
!               RFATBL_PTR : REF RFATBL$BLOCKVECTOR(RFATBL_SIZE_ENTRIES)
!
!
! Declare the fields of an RFA table entry. Also declare the declaration macro.

FIELD RFATBL$FLD_DEF =
        SET
        RFATBL$L_RFA0   = [ 0 , L_ ] ,  ! Block number part of RFA
        RFATBL$W_RFA4   = [ 4 , WO_]    ! Byte offset part of RFA
        TES;

LITERAL
        RFATBL$K_SIZE   = 6;            ! Table entry size in bytes

MACRO
        RFATBL$BLOCKVECTOR(N) =
                BLOCKVECTOR [N, RFATBL$K_SIZE, BYTE] FIELD(RFATBL$FLD_DEF) %;
```

S T A T I C A D D R E S S T A B L E D E F I N I T I O N S

The Static Address Table (SAT) consists of a two-level structure. There is a Program Static Address Table which specifies what static addresses (code addresses and static data addresses) belong to what modules. This table consists of a linked list of SAT entries where each entry specifies an address range and a pointer to the corresponding Module RST Entry. The list is ordered by start address so that low start addresses come before higher start addresses. It should be noted that a given address may be covered by more than one SAT entry because different modules can have global PSECTs in common--Fortran COMMON blocks cause this situation to arise. Given an address, the Program Static Address Table can thus be searched to find the module that contains the address.

On a second level, there is a Module Static Address Table for each module that is SET. This SAT is a linked list of exactly the same structure as the Program SAT except that the RST pointer in the SAT entry points to the RST entry of the symbol, not the module, containing the address range. This list is also ordered by start address and again more than one symbol may contain the same address (due to Fortran EQUIVALENCEing for example). If two entries have the same start address, the entry with the larger end address is placed first in the Module SAT; this is crucial to the correct behavior of the DBG$STA_SETCONTEXT routine where routine addresses are compared to PC values from the VAX call stack. The Module RST Entry contains a pointer (RST$L_SAT_PTR) which points to the first SAT entry on the module's SAT chain.

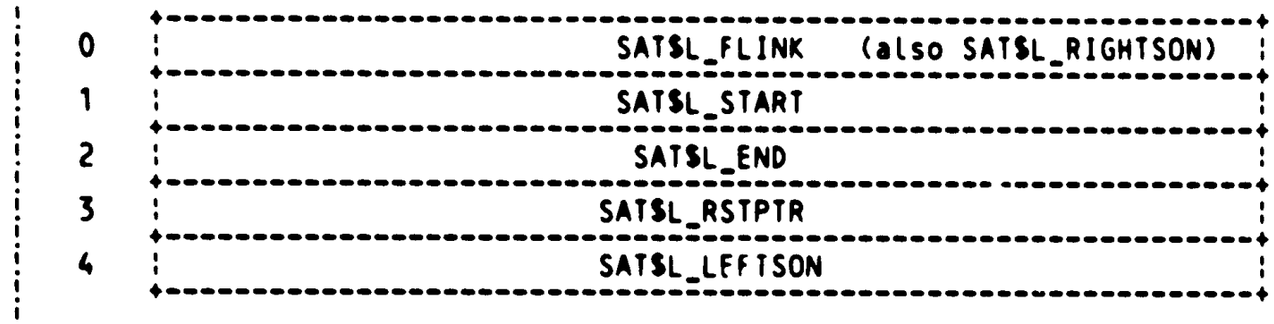
Note (R. Title)- for a future release of DEBUG, I intend to turn the SAT chains into binary trees, instead of linked lists. The reason for this is that for large programs, the SAT chains can grow quite long (e.g., 1000 entries), and it becomes quite expensive to search this list linearly.

The tree search algorithm will work something like this: If the desired address is greater than the address in the SAT$L_END field of the current node, follow the "right son" pointer. If it is less than the address in the SAT$L_START field of the current node, then follow the "left son" pointer. If it is in the range, then we have found the desired SAT entry and we are done.

During the transition to this new data structure, the same four fields are being kept in the SAT entry in the same place (so that the old code still works). The "right son" field is overlaid with the FLINK field, so that the tree search algorithm will work on the old linked lists (a linear list is just an extremely unbalanced tree, with no left branches). The "left son" field will occupy the fifth longword.

The individual Static Address Table entry has this format:

```
 3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
```



A pointer to a Static Address Table entry is declared as follows:

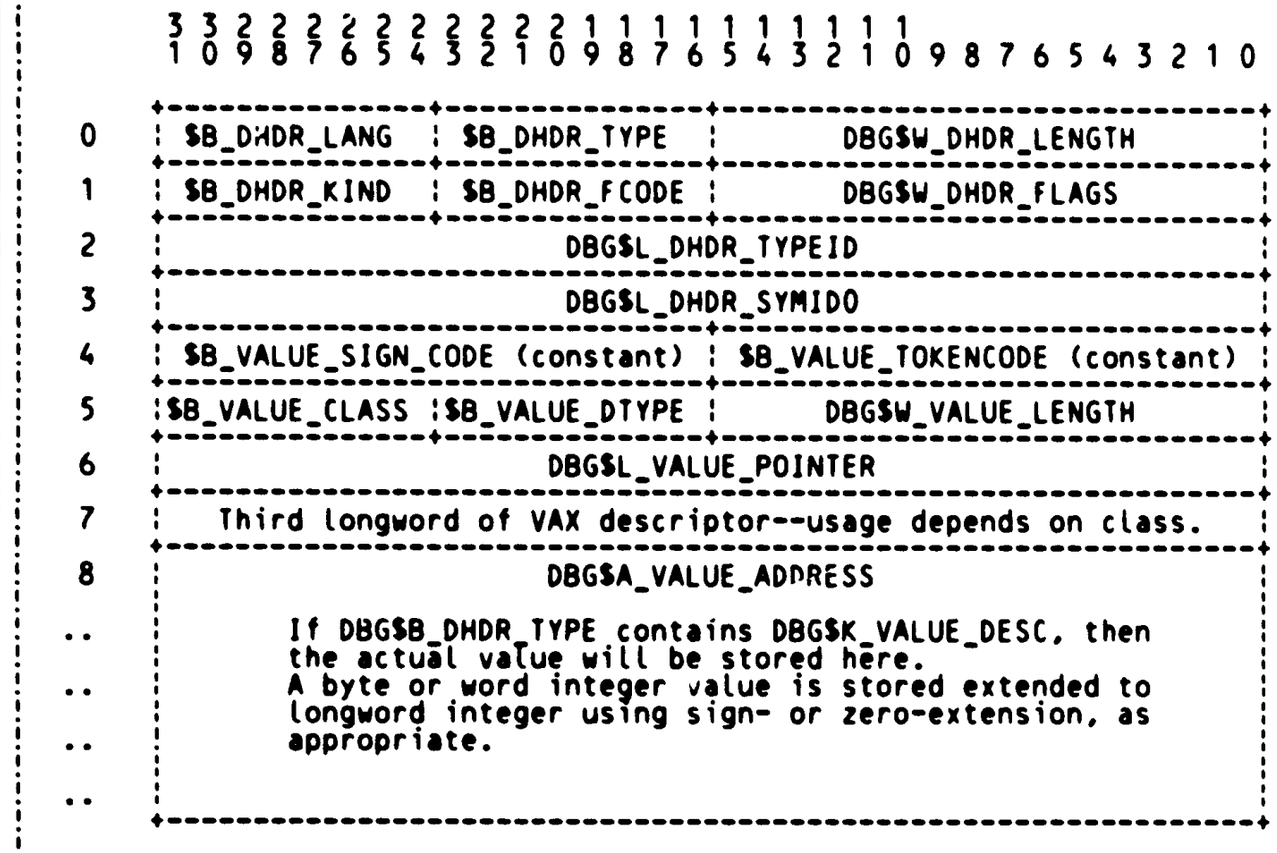
```
SATPTR: REF SAT$ENTRY
```

Field definitions for the SAT entry.

```
FIELD SAT$FLD_DEF =
        SET
        SAT$L_FLINK     = [ 0, L_ ],    ! Forward link to next SAT entry on the
                                        !     chain--zero terminate the chain.
        SAT$L_RIGHTSON  = [ 0, L_ ],    ! Link to right son in binary tree.
        SAT$L_START     = [ 1, L_ ],    ! Start address of address range
        SAT$L_END       = [ 2, L_ ],    ! End address of address range
        SAT$L_RSTPTR    = [ 3, L_ ],    ! Pointer to RST entry of module or sym-
                                        !     bol containing this address range
        SAT$L_LEFTSON   = [ 4, L_ ]     ! Link to left son in binary tree.
        TES;

LITERAL
        SAT$K_ENTSIZE   = 5;            ! Size of one SAT entry in longwords

MACRO
        SAT$ENTRY = BLOCK[SAT$K_ENTSIZE] FIELD(SAT$FLD_DEF) %;
```

V A L U E D E S C R I P T O R D E F I N I T I O N S

Value Descriptors describe language values generated during the processing of language expressions in DEBUG commands like EVAL and DEPOSIT. The language-independent version of the Value Descriptor consists of the Value and Primary Descriptor header fields followed by a 3-longword VAX Standard Descriptor followed by the actual value described by the Value Descriptor. It thus has the following format:



A pointer to a Value Descriptor is declared as follows:

```
VALPTR: REF DBG$VALDESC
```

Define the fields of the language-independent Value Descriptor. Also define the declaration macro.

```
FIELD DBG$VALUE_FIELDS =
        SET
```

```
        DBG$W_VALUE_TOKENCODE=[ 4, W0_ ],       ! Value of TOKEN$W_CODE for constants
        DBG$W_VALUE_SIGN_CODE=[ 4, W1_ ],       ! Sign code (Value of Unary -/+
                                                !    TOKEN$W_CODE) for constants
        DBG$A_VALUE_VMSDESC = [ 5, A_  ],       ! Address of VAX/VMS descriptor
        DBG$B_VALUE_CLASS   = [ 5, B3_ ],       ! DSC$B_CLASS   sub-field
        DBG$B_VALUE_DTYPE   = [ 5, B2_ ],       ! DSC$B_DTYPE   sub-field
        DBG$W_VALUE_LENGTH  = [ 5, W0_ ],       ! DSC$W_LENGTH  sub-field
        DBG$L_VALUE_POINTER = [ 6, L_  ],       ! DSC$A_POINTER sub-field
        DBG$B_VALUE_SCALE   = [ 7, B0_ ],       ! DSC$B_SCALE   sub-field
        DBG$B_VALUE_DIGITS  = [ 7, B1_ ],       ! DSC$B_DIGITS  sub_field
        DBG$V_VALUE_FL_BINSCALE = [7, 19, 1, 0], ! DSC$V_FL_BINSCALE sub-field
        DBG$L_VALUE_POS     = [ 7, L_  ],       ! DSC$L_POS     sub-field
        DBG$A_VALUE_ADDRESS = [ 8, A_  ],       ! First byte of value
        DBG$L_VALUE_VALUE0  = [ 8, L_  ],       ! First longword of value
        DBG$L_VALUE_VALUE1  = [ 9, L_  ]        ! Second longword of value
        TES;

MACRO
        DBG$VALDESC = BLOCK [,LONG] FIELD(DBG$DHDR_FIELDS, DBG$VALUE_FIELDS) %;


! Define the "base size" of a value descriptor in longwords. For the cases
! where the value is actually stored in the descriptor, the total size
! is the base size plus the number of longwords needed to hold the actual
! value.
!
LITERAL
        DBG$K_VALDESC_BASE_SIZE = 8;
```

```
!               O L D   D E B U G G E R   D E F I N I T I O N S
!
!
!       These definitions are used by old Debugger modules which also need to
!       use the new RST (and other) definitions.
!
LITERAL
        ! constants used to identify individual rst flags
        rst$f_global = 1,
        rst$f_nonzlength = 2,
        rst$f_modset    = 3;

MACRO

        !
        !  Get the srm type address from a dst with address calculation commands
        !  The type is following the name string
        !    7 - size of fixed portion of dst
        !    1 - byte to hold count of name string
        !    dst[dstr_name] - contains count of characters in name string
        !
        !    add this distance to base address to get address of byte
        !       containing the srm type

        CAD_SRM_TYPEA(DST) =
                ( .DST + 7 + 1 + .DST[dst$b_name] ) %,

        !  Get the srm type from a dst with address calculation commands

        CAD_SRM_TYPE(DST) =
                ( .(CAD_SRM_TYPEA(DST))<0,8,0>)   %,

        !  Get the address of the calculation commands in a dst which contains
        !   them.  They start one byte past the srm type.

        CAD_COMMANDS(DST) =
                ( CAD_SRM_TYPEA(DST) + 1 ) %;


MACRO
        RST_UNITS( bytes ) =

                ( ((bytes) + %upval-1)/%upval )
        %;

MACRO
                                ! DEBUG tells the RST module about ASCII
                                ! strings by passing a counted string pointer.
        CS_POINTER      = REF VECTOR[1,BYTE] %;

LITERAL

        ! We will never print "symbol+offset" when the
        ! upper bound for "symbol" is 0 and when
        ! the offset is greater than RST_MAX_OFFSET
```

```
        RST$K_MAX_OFFSET                = %X'100';

!+
! Since scope definitions are recursive, we must
! stack ROUTINE BEGINs in the routine ADD_MODULE.
! It is no coincidence that this stack limit is the
! same as the limit on the length (in elements) of
! symbol pathnames.
!-

LITERAL
        MAX_SCOPE_DEPTH = dbg$k_max_pathname;    ! Routines can be nested to a maximum depth.
```

```
FIELD
        VALU_FIELD_SET =
    SET
        VALU_NT_PTR     = [  0,0,32,0 ],        ! Associated NT pointer.
        VALU_VALUE      = [  4,0,32,0 ]         ! The actual value.
    TES;

!+
! Declare an occurrence or REF to a VALUE_DESCRIPTOR
! via the following macros.
!-

LITERAL
        VALU_DESC_SIZE  = 8;            ! Each one is 2 longwords long.

MACRO
        VALU_DESCRIPTOR = BLOCK[ VALU_DESC_SIZE, BYTE ] FIELD( VALU_FIELD_SET ) %;
```

```
!++
!  Array Bounds Descriptor
!
!  An array bounds Descriptor is used to pass around all needed
!  information about an array and its associated dimensions.
!  Like VALU_DESCRIPTORs, they are simply 2-longword blocks,
!  but this might change.
!
!       !------longword-----!
!
!       !------------------!
!       ! address of array !
!       !------------------!
!       ! length of array  !
!       !------------------!
!
!  Such Descriptors must be accessed via the following
!  field names.
!--

FIELD
        ARRAY_BNDS_SET =
    SET
        ARRAY_ADDRESS   = [  0,0,32,0 ],        ! Beginning address of array.
        ARRAY_LENGTH    = [  4,0,32,0 ]         ! Size, in bytes, of array.
    TES;

!+
! Declare an occurrence or REF to an array bounds
! descriptor via the following macros.
!-

LITERAL
        ARRAY_BNDS_SIZE = 8;            ! Each one is 2 longwords long.
MACRO
        ARRAY_BNDS_DESC = BLOCK[ ARRAY_BNDS_SIZE, BYTE ] FIELD( ARRAY_BNDS_SET ) %;
```

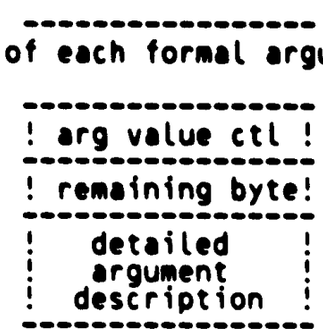
```
LITERAL
        SL_ACCE_INIT    = 0,    ! See above.   "SL" --> SAT/LVT
        SL_ACCE_RECS    = 1,
        SL_ACCE_SORT    = 2,
        SL_ACCE_FREE    = 3;


!+
! You declare an occurrence or REF of an SAT datum via
! the macro, SAT_RECORD.
!-

MACRO
        SAT_RECORD      = BLOCK[ ] FIELD( sat$fld_def ) %;
```

```
!++
! Now we bring in the GST definitions directly from STARLET.REQ.
!
!       The format of one of these concatenated GSD records is a single
!       leading byte containing the value 1, indicating that the record
!       is indeed a GSD record.
!
!       Each entry in the record has a fixed number of overhead
!       bytes followed by a symbol name that is a variable number of
!       bytes. The entries we are interested in processing are the
!       global symbol definitions and entry point symbol and mask
!       definitions. The other defined type, PSECT definition, is
!       noted only because it must be successfully passed over. The
!       format of each of these types is illustrated below:
!
!       Global symbol definition:
!
!               ----------------
!       0       !   GSD type 1   !
!               ----------------
!       1       !   data type    !       ignored for now
!               ----------------
!       2       !      flag      !       bit 1 set means that this is
!       3       !      bytes     !       a definition. ignore bit 0.
!               ----------------
!       4       !  psect index   !       ignored.
!               ----------------
!       5       !                !
!               !     value      !       4 bytes
!               !                !
!               ----------------
!       9       !                !
!               !     symbol     !       stock counted character
!               !      name      !       string.
!               !                !
!               ----------------
!
!       The entry point symbol and mask definition entry is identical to
!       the global symbol definition illustrated above, with the
!       addition of a two byte field for the procedure's register save
!       mask. This two byte field is located after the symbol value
!       field (which is an entry point address).
!
!               ----------------
!       0       !   GSD type 2   !
!               ----------------
!       1       !   data type    !       ignored for now
!               ----------------
!       2       !      flag      !       not relevant for
!       3       !      bytes     !       entry point def.
!               ----------------
!       4       !  psect index   !       ignored
!               ----------------
!               !                !
!       5       !     value      !       4 bytes
```

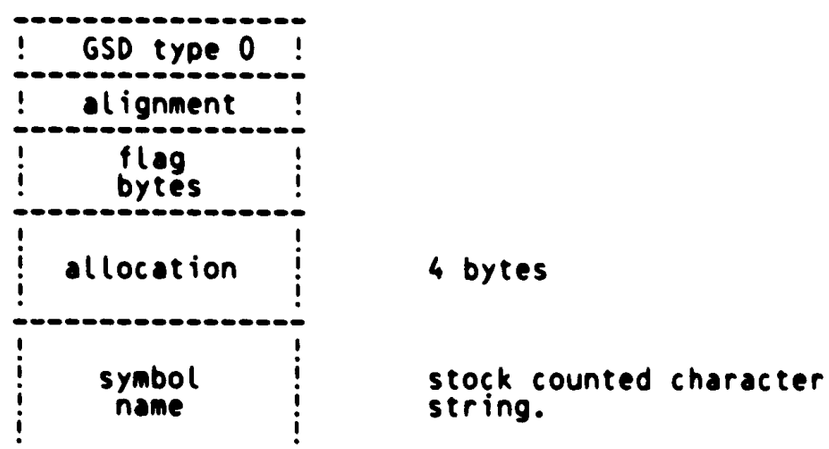
	GSD type 3	
1	data type	ignored for now
2 3	flag bytes	bit 1 set indicates that this is a definition. Bit 0 ignored.
4	psect index	ignored
5	value	4 bytes
9 10	register save mask	ignored, 2 bytes
11	symbol name	stock counted character string
	min # of args	1 byte
	max # of args	1 byte
	formal arg #1 description	
	. . .	
	formal arg #n description	

The procedure definition with formal argument descriptions is identical to the "GSD type 2" definition above with the addition of fields that describe each formal argument.
Following the symbol name there is one byte containing the minimum number of arguments allowed, and one byte containing the maximum number of arguments.
These bytes are followed by a series of records of 2 - 257 bytes that describe each of the formal arguments (the number of these records is equal to the maximum number of arguments for the procedure.

Format of each formal argument description.



PSECT definition:




```
The format of GST records is defined as a BLOCK
(even though the records are variable sized)
with the following fields:

--

FIELD
        GST_FIELD_SET =
    SET
        GST_ENTRY_TYPE  = [  0,0, 8,0 ],        ! Type of GST record.
        GST_IS_DEFN     = [  2,1, 1,0 ],        ! This flag implies whether o no
                                                ! the record is an entry definition.
        GST_VALUE       = [  5,0,32,0 ],        ! The value of the global.
                                                ! (This won't work for PSECTs)
        GST_P_NAME_CS   = [  8,0, 8,0 ],        ! Character count of psect name

        ! The following 2 pairs are mutually
        ! exclusive.  The first one is for GST
        ! records of type GST_GLOBAL_DEFN, the
        ! second is for GST_ENTRY_DEFN or GST_PROC_DEFN.

        GST_G_NAME_CS   = [ 9,0, 8,0 ],         ! The symbol name is a counted string.
```

```
!                                               ! A dotted reference to this field
!                                               ! picks up the count, an undotted
!                                               ! one addresses the counted string.
!       GST_G_NAME_ADDR = [10,0, 8,0 ],         ! The name string itself.  An undotted
!                                               ! reference is the address of the name,
!                                               ! a dotted one is the 1st character.
!
!
!       GST_E_NAME_CS   = [11,0, 8,0 ],         ! The entry name is a counted string.
!                                               ! A dotted reference to this field
!                                               ! picks up the count, an undotted
!                                               ! one addresses the counted string.
!       GST_E_NAME_ADDR = [12,0, 8,0 ],         ! The name string itself.  An undotted
!                                               ! reference is the address of the name,
!                                               ! a dotted one is the 1st character.
!       GST_P_MAX_ARG   = [ 1,0, 8,0 ],         ! Maximum number of formal arguments.
!       GST_P_REM_CNT   = [ 1,0, 8,0 ]          ! Remaining byte count of argument
!                                               ! descriptor.
!   TES;
!
!+
! You declare an occurrence or REF of a GST datum via:
!-
!
!LITERAL
!       GST_RECORD_SIZE = 43;   ! Each GST record is at most 43 bytes long.
!MACRO
!       GST_RECORD = BLOCK[ GST_RECORD_SIZE, BYTE] FIELD( GST_FIELD_SET ) %;
!
!
!LITERAL
!       GST_PSECT_OVERHEAD = 9,         ! Minimum number of bytes in psect def
!       GST_GLOBAL_OVERHEAD = 10,       ! Minimum number of bytes in a global entry
!       GST_ENTRY_OVERHEAD = 12,        ! Minimum number of bytes in entry
!                                       !       point symbol and mask definition
!       GST_ARGDSC_OVERHEAD = 2,        ! Minimum size of formal argument descriptor
!       GST_MINMAX_OVERHEAD = 2,                ! Size of min. and max. overhead in GST
!       GST_RECORD_TYPE = 0,            ! Type of GST record
!       GST_TYPE        = 1;            ! Record Type is GST
!
!+
! The GST record types are defined as:
!-
!
!LITERAL
!       ! GST types:
!
!       GST_LOWEST      = 1,            ! We don't support global PSECTs now.
!
!       GST_PSECT_DEFN  = 0,            ! P-SECT record.
!       GST_GLOBAL_DEFN = 1,            ! A global symbol definition record.
!       GST_ENTRY_DEFN  = 2,            ! An entry point definition.
!       GST_PROC_DEFN   = 3,            ! A procedure with formal argument desc.
!
!       GST_HIGHEST     = 3;            ! Highest one we support.
```

```
!++
! BLISS uses 'non-standard' DST records to encode
! most of its local symbol information.  These records
! are like most DST records except that the TYPE
! information is variable-sized.
!--

FIELD
        BLZ_FIELD_SET =
    SET
        BLZ_SIZE        = [  0,0, 8,0 ],        ! First byte is record size in bytes.

        ! The next byte contains DSC$K_DTYPE_Z, or we
        ! wouldn't be applying this structure to a given
        ! DST record.

        BLZ_TYP_SIZ     = [  2,0, 8,0 ],        ! Type info takes up this
                                                !  many bytes.
        BLZ_TYPE        = [  3,0, 8,0 ],        ! Which type of type Zero
                                                !  this corresponds to.
        BLZ_ACCESS      = [  4,0, 8,0 ],        ! Access field.

        ! Sub fields of _ACCESS are offset from beginning
        ! of the BLZ record.

                BLZ_ACCES_TYPE  = [  4,0, 2,0 ],        ! Type of access,
                BLZ_ACCES_BASD  = [  4,2, 2,0 ],        ! based or not,
                BLZ_ACCES_BREG  = [  4,4, 4,0 ],        ! associated register.

        BLZ_STRUCT      = [  5,0,3,0 ],                 ! Type of STRUCTURE reference.
        BLZ_REF         = [  5,7,1,0 ],                 ! Indicates whether symbol has REF attribute

        ! **** The following only work when BLZ_TYP_SIZ is 3.

        BLZ_VALUE       = [  6,0,32,0 ],        ! DST VALUE field.
        BLZ_NAME_CS     = [ 10,0, 8,0 ],        ! The symbol name is a counted string.
                                                ! A dotted reference to this field
                                                ! picks up the count, an undotted
                                                ! one addresses the counted string.
        BLZ_NAME_ADDR   = [11,0, 8,0 ],         ! The name string itself.  An undotted
                                                ! reference is the address of the name,
                                                ! a dotted one is the 1st character.

        ! The following fields are in the variable length part of the DST record.
        ! They should only be applied once the structure type has been determined
        ! or errors could result.

        U_ALLOC_STRUC   = [ 6, 0, 32, 0],       ! no of units alloc (except BLOCKVECTOR)
        U_ALLOC_BVEC    = [10, 0, 32, 0],       ! no of units alloc for BLOCKVECTOR
        SIGN_EXT_VEC    = [10, 7,  1, 0],       ! sign ext field for VECTOR
        UNIT_SIZE_BLOCK = [10, 0,  8, 0],       ! unit size for BLOCK
        UNIT_SIZE_BVEC  = [14, 0,  8, 0],       ! unit size for BLOCKVECTOR
        UNIT_SIZE_VEC   = [10, 0,  4, 0]        ! unit size for VECTOR
    TES;

!+
```

```
! You declare a REF to a BLZ_DST datum via:
!-

LITERAL
        BLZ_REC_SIZ     = 54;   ! Each DST record is at most 54 bytes long.

MACRO
        BLZ_RECORD = BLOCK[ BLZ_REC_SIZ, BYTE] FIELD( BLZ_FIELD_SET ) %;


!+
! The type zero sub types,
! as defined in CP0021.MEM,
! must be within the following
! range.
!-

LITERAL

        ! Type Zero Sub-Types:

        BLZ_LOWEST      = 1,    ! Lowest variable type we support.

        BLISS_Z_FORMAL  = 1,    ! Description of a ROUTINE formal.
        BLISS_Z_SYMBOL  = 2,    ! A BLISS LOCAL symbol.

        BLZ_HIGHEST     = 2;    ! Highest variable type we support.
```

```
!+
!  Dummy descriptors
!
!  Whenever a type DSC$K_DTYPE_CAD item is being accessed by descriptor
!  we must build a dummy one in free storage and put the current address
!  in it.  The address of this dummy descriptor is then used by the rest
!  of the debugger in referencing the item.  To be able to free up all this
!  space, the items are kept on a linked list. At the end of command processing
!  this list is walked down and all the space is returned to the free storage
!  manager.
!
!  Each entry on the linked list is a 3 longword vector
!
!       |-----------------------------------------------------------|
!       |     pointer to next item on the linked list               |
!       |-----------------------------------------------------------|
!       |     pointer to the dummy descriptor                       |
!       |-----------------------------------------------------------|
!       |     size in bytes of the dummy descriptor                 |
!       |-----------------------------------------------------------|
!
!  The global variable  DBG$GL_DLISHEAD if nonzero points to the first
!  item on the list.

LITERAL
        DLIS_ENTRY =    3 ,             ! Size in longwords of entry on list
        DLIS_LINK  =    0 ,             ! Pointer to next item on list
        DLIS_POINTER = 1  ,             ! Pointer to dummy descriptor
        DLIS_DESCSIZE=  2 ;             ! Size in bytes of descriptor
```

```
!       E N D   O F   D B G L I B . R E Q
```

S0DEBUG
MAP

DBGLIB
REQ

DEBUG

DEBUG
MAP

DBGSSISHR
MAP

DBGEXT
REQ

DBGMSG
MDL

DBGGEN
REQ

DBGPROLOG
REQ

DSTRECRDS
REQ