```
TTTTTTTTTTTTTTTT  RRRRRRRRRRRR        AAAAAAAAA        CCCCCCCCCCCC  EEEEEEEEEEEEEEE
TTTTTTTTTTTTTTTT  RRRRRRRRRRRRR       AAAAAAAAA        CCCCCCCCCCCCC  EEEEEEEEEEEEEEE
TTTTTTTTTTTTTTTT  RRRRRRRRRRRRR       AAAAAAAAA        CCCCCCCCCCCCC  EEEEEEEEEEEEEEE
      TTT         RRR          RRR   AAA         AAA   CCC            EEE
      TTT         RRR          RRR   AAA         AAA   CCC            EEE
      TTT         RRR          RRR   AAA         AAA   CCC            EEE
      TTT         RRR          RRR   AAA         AAA   CCC            EEE
      TTT         RRR          RRR   AAA         AAA   CCC            EEE
      TTT         RRR          RRR   AAA         AAA   CCC            EEE
      TTT         RRRRRRRRRRRRR      AAA         AAA   CCC            EEEEEEEEEEEE
      TTT         RRRRRRRRRRRRR      AAA         AAA   CCC            EEEEEEEEEEEE
      TTT         RRRRRRRRRRRRR      AAA         AAA   CCC            EEEEEEEEEEEE
      TTT         RRR    RRR         AAAAAAAAAAAAAAA   CCC            EEE
      TTT         RRR    RRR         AAAAAAAAAAAAAAA   CCC            EEE
      TTT         RRR    RRR         AAAAAAAAAAAAAAA   CCC            EEE
      TTT         RRR        RRR     AAA         AAA   CCC            EEE
      TTT         RRR        RRR     AAA         AAA   CCC            EEE
      TTT         RRR        RRR     AAA         AAA   CCC            EEE
      TTT         RRR     RRR        AAA         AAA   CCC            EEE
      TTT         RRR     RRR        AAA         AAA   CCC            EEE
      TTT         RRR        RRR     AAA         AAA      CCCCCCCCCCCC  EEEEEEEEEEEEEEE
      TTT         RRR        RRR     AAA         AAA      CCCCCCCCCCCC  EEEEEEEEEEEEEEE
      TTT         RRR        RRR     AAA         AAA      CCCCCCCCCCCC  EEEEEEEEEEEEEEE
```

FILEID**TBKDST

```
TTTTTTTTTT  BBBBBBBB    KK      KK  DDDDDDDD     SSSSSSSS  TTTTTTTTTT
TTTTTTTTTT  BBBBBBBB    KK      KK  DDDDDDDD     SSSSSSSS  TTTTTTTTTT
    TT      BB      BB  KK      KK  DD      DD  SS            TT
    TT      BB      BB  KK      KK  DD      DD  SS            TT
    TT      BB      BB  KK    KK    DD      DD  SS            TT
    TT      BB      BB  KK    KK    DD      DD  SS            TT
    TT      BBBBBBBB    KKKKKK      DD      DD    SSSSSS      TT
    TT      BBBBBBBB    KKKKKK      DD      DD    SSSSSS      TT
    TT      BB      BB  KK    KK    DD      DD          SS    TT
    TT      BB      BB  KK    KK    DD      DD          SS    TT
    TT      BB      BB  KK      KK  DD      DD          SS    TT        ....
    TT      BB      BB  KK      KK  DD      DD          SS    TT        ....
    TT      BBBBBBBB    KK      KK  DDDDDDDD  SSSSSSSS      TT        ....
    TT      BBBBBBBB    KK      KK  DDDDDDDD  SSSSSSSS      TT        ....
```

```
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
```

```
!----------------------------------------------------------------------
!
!       DSTRECRDS -- DEFINITION FILE FOR THE DEBUG SYMBOL TABLE
!
!----------------------------------------------------------------------
!
! 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
!       Bruce Olsen     August, 1980.
!       Bert Beander    August, 1981.
!       Bert Beander    November, 1983.

! MODULE FUNCTION
!       This REQUIRE file describes the structure of the Debug Symbol Table
!       generated by the VAX compilers and interpreted by the VAX Debugger.
!       It includes definitions for all field names and literals used in
!       building or interpreting the Debug Symbol Table (DST).

! DISCLAIMER
!       This interface is not supported by Digital.  While the Debug Symbol
!       Table interface is believed to be correctly described here, Digital
!       does not guarantee that all descriptions in this definition file are
!       correct and complete.  Also, while this interface is expected to be
!       reasonably stable across releases, Digital cannot guarantee that it
!       will not change in future releases of VAX DEBUG, VAX VMS, the VAX
!       compilers, or other software.  Upward-compatible additions to this
!       interface are more likely than incompatible changes, but individuals
!       and organizations who use this interface stand some risk that their
!       work will be partially or wholly invalidated by future releases of
!       VAX DEBUG or other Digital software.  Digital reserves the right to
```

```
!       make future incompatible changes to the Debug Symbol Table interface.
```

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

PURPOSE OF THE DEBUG SYMBOL TABLE

The Debug Symbol Table (DST) is the symbol table that the VAX compilers produce to pass symbol table information to the VAX Debugger and to the VAX Traceback facility. The DST is a language-independent symbol table in the sense that all VAX compilers output symbol information in the same format, regardless of source language. This symbol information is emitted into the object modules produced by the compiler. It is then passed through the linker into the executable image file that the linker generates. DEBUG or TRACEBACK can then retrieve the symbol information from the image file.

The purpose of the Debug Symbol Table is thus to permit the Traceback facility to give a symbolic stack dump on abnormal program termination and to permit DEBUG to support fully symbolic debugging. Other Digital software may also use the DST information for various purposes.

To support these purposes, the Debug Symbol Table represents all major aspects of program structure and data representation. It can represent modules, routines, lexical blocks, labels, and data symbols and it can represent all nesting relationships between such symbols. It can also describe line number and source line information. It can describe all data types supported by DEBUG, including complex types such as record structures and enumeration types. In addition, it can describe arbitrarily complex value and address computations.

The Debug Symbol Table is solely intended to support compiled languages, not interpreted languages. The DST representation assumes that source lines have been compiled into VAX instructions and that those instructions are actually executed, not interpreted. Such DEBUG facilities as breakpoints and single-stepping will not work if this assumption is violated. Similarly, it is assumed that data objects have addresses that can be accessed directly when these objects are examined or deposited into. DST information is thus generated by all compilers that VAX DEBUG supports, but not by the interpreters for languages such as APL or MUMPS.

GENERAL STRUCTURE OF THE DST

This section describes the general structure of the Debug Symbol Table. It explains how the DST is generated by the various VAX compilers, how it is passed along to the executable image file by the linker, and how it is accesses in the image file by DEBUG or TRACEBACK. This section also describes in general terms how the DST is structured internally: how it is subdivided into modules, routines, lexical blocks, and individual symbols, how nesting relationships are represented, and how data symbols, including their values and data types, are represented. The exact formats of the various Debug Symbol Table records and other fine-grained detail are described later in this definition file, not here, but the coarse structure of the DST and how that structure is accessed are outlined in this section.

GENERATION OF THE DST

The Debug Symbol Table (DST) is generated by the compilers for all VAX languages supported by DEBUG. During compilation, the compiler outputs the DST for the module being compiled into the corresponding object file. When the linker is invoked, it does relocation and global-symbol resolution on the DST text and then outputs it into the executable image file. Beyond knowing what must be relocated, the linker has no special knowledge of the format or contents of the DST. Finally, the Debugger reads the DST information from the executable image file during a debugging session, or Traceback reads it when giving a traceback in response to an unhandled severe exception during image execution.

A compiler outputs DST information in the form of two kinds of object records, TBT records and DBT records. (See the linker manual for a full description of the VAX object language accepted by the linker.) All "traceback" information goes into the TBT records and all "symbol" information goes into the DBT records. When the user later links using the plain LINK command, only the DST information in the TBT records are copied to the executable image file. These records contain enough information for Traceback to give a call-stack traceback. If the user links with the LINK/DEBUG command, all information in both the TBT and the DBT records are copied to the executable image file. These records together give all DST information needed for full symbolic debugging. The user can also link with LINK/NOTRACEBACK, in which case no DST information at all is copied to the executable image file.

It is not possible to have the linker copy the DBT records without also copying the TBT records; the information in the TBT records is required for the information in the DBT records to make sense.

The "traceback" information in the TBT records includes all Module Begin and End DST records, all Routine Begin and End DST records, all Lexical Block Begin and End DST records, and all Line Number PC-Correlation DST records. It may also include Version Number DST records. All other DST records should be included in DBT records.

Most VAX compilers have a /DEBUG qualifier which in its most general form has two subqualifiers: /DEBUG=([NO]TRACEBACK,[NO]SYMBOLS). The unadorned /DEBUG qualifier is equivalent to /DEBUG=(TRACEBACK,SYMBOLS); it causes all DST information to be output. /DEBUG=TRACEBACK causes only the traceback information (the TBT records) to be output by the compiler. /DEBUG=(NOTRACE,NOSYMBOL) causes no DST information to be output at all. Finally, /DEBUG=(NOTRACE,SYMBOLS) causes all DST information except Line Number PC-Correlation DST records to be output (this combination is largely pointless although it saves some DST space). Note that the module, routine, and lexical block information, which counts as traceback information, must be output if any symbol information is output since it defines the scopes within which other symbols are defined.

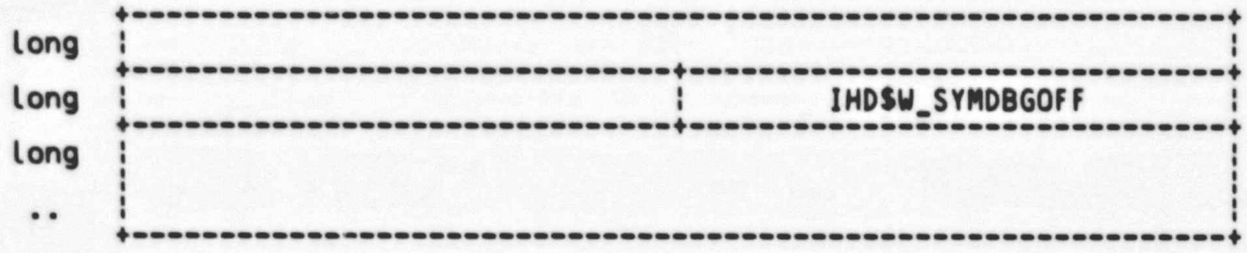
When the linker outputs the Debug Symbol Table to the executable image file, it may also output two more image sections: the Global Symbol Table (GST) and the Debug Module Table (DMT). These two tables are generated if the LINK/DEBUG command is used, not otherwise. The Global Symbol Table contains records for all global symbols known to the linker in the current user program. DEBUG uses the GST as a symbol table of last resort when DST information is not available, either because the module containing some global symbol was compiled without DST information being output or because the module is not set (with SET MODULE) in the current debugging session. The GST information is not as complete as the DST information for the same symbols because the GST has no type description (the linker does not need to know about data types).

The Debug Module Table (DMT) is an indexing structure for the DST. It contains one record for each module in the DST. This record contains a pointer to the start of the DST for the corresponding module, the size of the DST for that module, the number of PSECTs in that module, and the address ranges of all those PSECTs. The DMT allows DEBUG to initialize its Module Table and its Static Address Table without actually having to read through the entire DST; because the DMT is very small compared to the DST, it can be scanned much more efficiently.

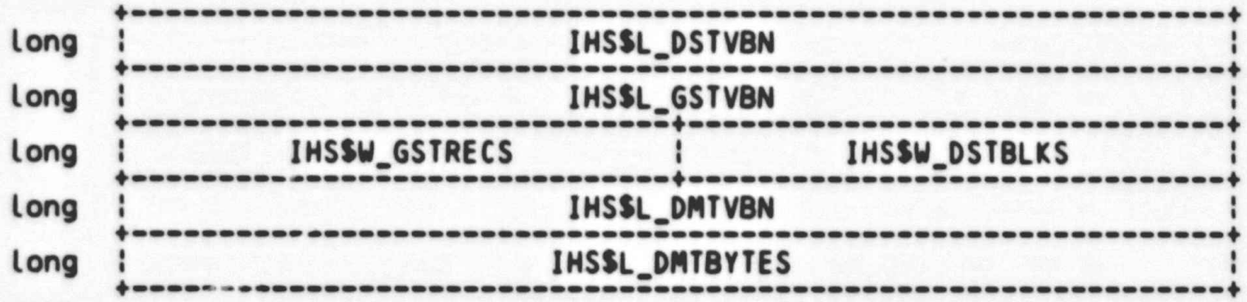
The details of how the DST, the GST, and the DMT are accessed in the executable image file are explained in the next section.

LOCATION OF THE DST WITHIN THE IMAGE FILE

The Debug Symbol Table is accessed through pointer information found in the executable image file header block. This header block contains a pointer in a fixed location (IHD$W_SYMDBGOFF) which points to a small block later in the header which gives the size and location of the Debug Symbol Table (DST), the Global Symbol Table (GST), and the Debug Module Table (DMT). 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. The Image Header Symbol Table Descriptor (IHS) in turn has the following format:



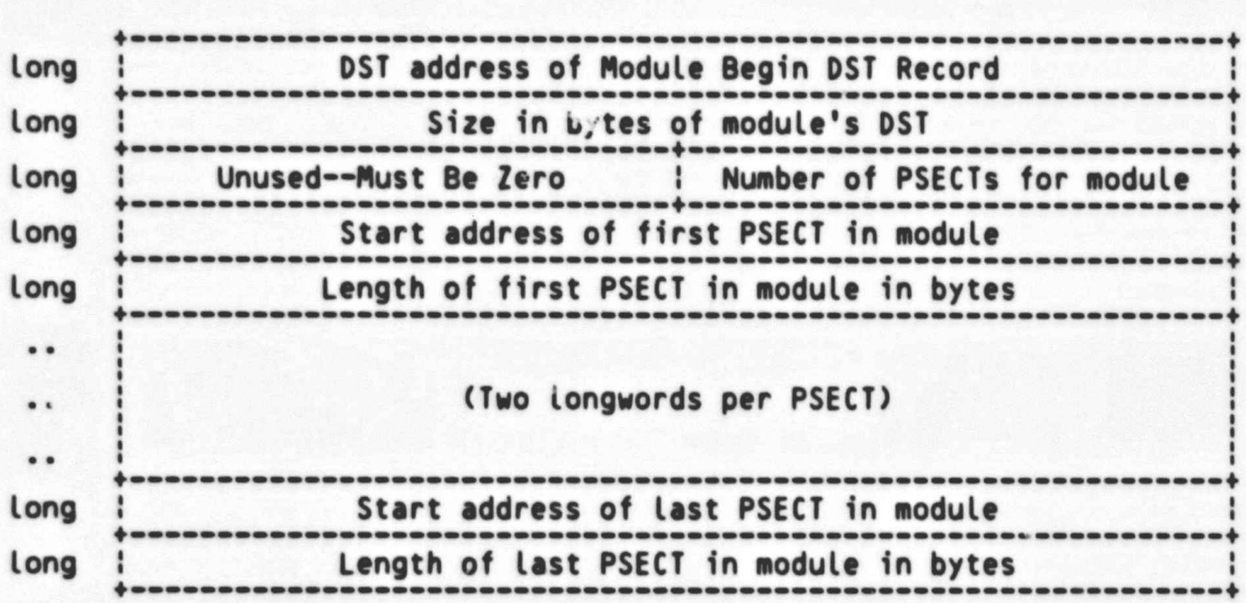
Here IHS$W_DSTBLKS and IHS$L_DSTVBN give the size (in blocks) and location (Virtual Block Number) of the Debug Symbol Table (DST) within the executable image file. The fields IHS$W_GSTRECS and IHS$L_GSTVBN give the size (in GST records) and start location (Virtual Block Number) of the Global Symbol Table (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 also defined in SYS$LIBRARY:LIB.L32.

Pointers to the Image Header and the Image Header Symbol Table Descriptor are declared as follows:

```
IHDPTR: REF BLOCK[,BYTE]
IHSPTR: REF BLOCK[IHS$K_LENGTH,BYTE]
```

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 DMT). IHS$L_DMTBYTES then gives the size of the DMT in bytes. The DMT is only built if the user did a LINK/DEBUG; if he did not, IHS$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 DEBUG's Run-Time Symbol Table (RST) and Program Static Address Table (Program SAT). Using the DMT is much faster than the alternative 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 DMT information, but is estimated by multiplying the DST size of each module by an appropriate scale factor.

OVERALL STRUCTURE OF THE DST

The Debug Symbol Table consists of a contiguous sequence of DST records. Each DST record contains a two-byte header which gives the length of the record in bytes and the type of the record. The structure of the rest of the record (if any) is determined by the record type. The length of the DST in 512-byte blocks is given in the image file header; if the DST does not fill the last block, that block is zero-padded to the end.

The largest structural unit within the DST is the module. Each module represents the symbol table information of a separately compiled object module. The DST for a module always begins with a Module Begin DST record and ends with a Module End DST record. The Module Begin DST record gives the name of the module and the source language in which it was written. The Module End DST record simply marks the end of the module and contains no other information. As noted above, if present, the Debug Module Table (DMT) points to the Module Begin DST record of each module represented in the DST. DEBUG uses the DMT (if present) to locate all modules in the DST.

The DST as a whole thus always begins with the Module Begin DST record for the first module in the DST. It is followed by the symbol information for that module. Then comes the Module End DST record for that module. Immediately after that Module End DST record comes the Module Begin DST record for the next module, and so on to the end of the whole DST, where the Module End DST record for the last module is found. The rest of that image file block is zero-filled to the next block boundary. Note that there is no break between modules in the DST.

NESTING WITHIN THE DST

For most languages, the symbol table must represent a variety of nesting relationships. Routines are nested within modules, data symbols are declared within routines, and even routines are nested within routines. Certain data constructs, in particular record structures, contain additional nesting relationships. In the Debug Symbol Table, such nesting relationships are represented by Begin-End pairs of DST records. We have already seen above that the largest subunit of the DST, namely the module, is represented by a Module Begin DST record and a Module End DST record bracketting the DST information for the module.

This principle extends to other nesting relationships. The DST information for a routine is thus represented by a Routine Begin DST record and a Routine End DST record enclosing the DST information for all symbols

local to or nested within that routine. Similarly, lexical blocks (such as BEGIN-END blocks or their equivalents in various languages) are represented by Block Begin and Block End DST records enclosing the symbol DST records local to that lexical block. The nesting of routines and blocks within one another to any depth (within reason) is represented by the proper nesting of the corresponding Begin and End DST records.

An example may help clarify this notion. The following example shows a program in a fictitious language along the corresponding sequence of DST records:

Program Structure	DST Record Sequence
MODULE M =	Module Begin M
BEGIN	
VAR SYM_M1: INTEGER;	Data SYM_M1 (DTYPE_L)
VAR SYM_M2: REAL;	Data SYM_M2 (DTYPE_F)
ROUTINE R1 =	Routine Begin R1
BEGIN	
VAR SYM_R11: BOOLEAN;	Data SYM_R11 (BOOLEAN)
VAR SYM_R12: INTEGER;	Data SYM_R12 (DTYPE_L)
END;	Routine End (for R1)
ROUTINE R2 =	Routine Begin R2
BEGIN	
VAR SYM_R21: DOUBLE;	Data SYM_R21 (DTYPE_D)
VAR SYM_R22: INTEGER;	Data SYM_R22 (DTYPE_L)
ROUTINE R2A =	Routine Begin R2A
BEGIN	
VAR SYM_R2A: BYTE;	Data SYM_R2A (DTYPE_B)
BEGIN	Block Begin (no name)
VAR BLK_V1: WORD;	Data BLK_V1 (DTYPE_W)
ROUTINE R2BLKR =	Routine Begin R2BLKR
BEGIN	
FOO:BEGIN	Block Begin FOO
VAR FOO_V:REAL;	Data FOO_V (DTYPE_F)
END;	Block End (for FOO)
VAR R2BLK_V2:REAL;	Data R2BLK_V2 (DTYPE_F)
END;	Routine End (for R2BLKR)
VAR BLK_V2: DOUBLE;	Data BLK_V2 (DTYPE_D)
END;	Block End (for no name)
END;	Routine End (for R2A)
VAR SYM_R23: REAL;	Data SYM_R23 (DTYPE_F)
END;	Routine End (for R2)
END;	Module End

Here module (compilation unit) M contains two module-level data items,

SYM_M1 and SYM_M2, and two routines, R1 and R2. Routine R2 in turn contains several local data symbols (SYM_R21, SYM_R22, and SYM_R23) and a nested routine R2A. R2A in turn contains an anonymous BEGIN-END block, that blocks contains two local data symbols BLK_V1 and BLK_V2 and a local routine R2BLKR, local routine R2BLKR contains a data symbol and a labelled BEGIN-END block FOO, and block FOO contains one local symbol, FOO_V. All this nesting is represented by Begin and End DST records in the Debug Symbol Table as illustrated on the right.

Additional nesting must be represented for data. A record (called a structure in some languages) is a composite data object containing some number of record components of various data types. A record component may itself be a record. In addition, some languages allow records to have "variants" (as in PASCAL), which imposes additional structure that must be represented in the DST.

A record type is represented by a Record Begin and Record End DST record pair bracketting the DST records for the record components. This notion is illustrated by this program segment and the corresponding DST:

Program Structure	DST Record Sequence
TYPE RECTYP =	Record Begin (RECTYP)
RECORD OF	
COMP1: INTEGER;	Data COMP1 (DTYPE_L)
COMP2: REAL;	Data COMP2 (DTYPE_F)
COMP3: DOUBLE;	Data COMP3 (DTYPE_D)
END;	Record End (for RECTYP)

Here RECTYP is a record type. Each object of this type is a record containing three components, COMP1, COMP2, and COMP3. This structure is represented in the DST by a Record Begin DST record followed by Data DST records for the components followed by a Record End DST record. The addresses specified in the component DST records are bit or byte offsets from the start of the RECTYP record as a whole.

In this example, the Record Begin DST record for RECTYP may in fact represent either a record type or a record object. A field in the Record Begin DST record indicates which. However, let us assume that RECTYP defines a record type. How do we then declare objects of that type? The following example illustrates how:

Program Structure	DST Record Sequence
	Data REC1 (SepTypSpec)
TYPE RECTYP =	Record Begin (RECTYP)
RECORD OF	
COMP1: INTEGER;	Data COMP1 (DTYPE_L)
COMP2: REAL;	Data COMP2 (DTYPE_F)
COMP3: DOUBLE;	Data COMP3 (DTYPE_D)
END;	Record End (for RECTYP)

```
VAR REC1: RECTYP;
VAR REC2: RECTYP;
```

```
Data REC2 (SepTypSpec)
Type Spec DST record
    (Indirect Type Spec
     pointing to RECTYP)
```

Here the same record type RECTYP is defined. Two objects of that type are also defined, REC1 and REC2. Both data objects are represented by Separate Type Specification DST records. Such a DST record must be immediately followed by a DST record that defines the symbol's data type. The REC1 Separate Type Specification DST record is immediately followed by the RECTYP Record Begin DST record; hence REC1 is of the RECTYP data type. The REC2 Separate Type Specification DST record is immediately followed by a Type Specification DST record. This record contains an Indirect Type Specification that points back to the Record Begin DST record for RECTYP. Hence REC2 is also of that record type.

Records may be nested in the sense that a record component may itself be an object of some record type. A record component of a record type is represented the same way as any other object of a record type, namely by a Separate Type Specification DST record. This record must be followed by a Record Begin DST record or by a Type Specification DST record that points to a Record Begin DST record. The record component can also be represented by a Record Begin DST record directly if this record is marked as defining an object rather than a type.

Record variants, as found in PASCAL, introduce additional structure. A detailed description of how variants are represented in the DST is found in the section on "Record Structure DST Records" later in this definition file. Here we will only give an example that illustrates the general scheme that is used:

Program Structure	DST Record Sequence
	Data REC1 (SepTypSpec)
TYPE RECTYP =	Record Begin (RECTYP)
RECORD OF	
COMP1: INTEGER;	Data COMP1 (DTYPE_L)
CASE TAG: BOOLEAN OF	Data TAG (BOOLEAN)
	Variant Set Begin
	(tag variable = TAG)
FALSE: (	Variant Value for FALSE
COMP2: REAL;	Data COMP2 (DTYPE_F)
COMP3: DOUBLE);	Data COMP3 (DTYPE_D)
TRUE: (	Variant Value for TRUE
COMP4: INTEGER);	Data COMP4 (DTYPE_L)
END CASE;	Variant Set End
END;	Record End (for RECTYP)

```
VAR REC1: RECTYP;
```

Nesting is also used to describe enumeration types as found in PASCAL and some other languages. An enumeration type is described by an Enumeration Type Begin DST record followed by Enumeration Type Element DST records for all the enumeration literals of the type followed by an Enumeration Type End DST record. Any actual object of the enumeration type must be described by a Separate Type Specification DST record. This example illustrates what the DST for an enumeration type looks like:

Program Structure	DST Record Sequence

`TYPE COLOR = (`
`  RED,`
`  GREEN,`
`  BLUE`
`  );`
	Data HUE (SepTypSpec)
	Enum Type Begin COLOR
	Enum Type Element RED
	Enum Type Element GREEN
	Enum Type Element BLUE
	Enum Type End (COLOR)

`VAR HUE: COLOR;` `VAR PAINT: COLOR;`	Data PAINT (SepTypSpec) Type Spec DST record (Indirect Type Spec pointing to COLOR)

A more detailed description is found in the section entitled "Enumeration Type DST Records" later in this definition file.

For some DST record types, DEBUG ignores all nesting relationships below the module level. Line Number PC-Correlation DST records, for example, may be scattered throughout the DST for a module. DEBUG treats all such DST records as defining the line number information for the module as a whole, regardless of how they may be scattered within or outside the routines and blocks of the module. Similarly, Source File Correlation DST records may be scattered throughout the DST for a module. Records such as these can be generated wherever the compiler finds it most convenient to generate them.

DATA REPRESENTATION IN THE DST

Data Symbols are described in the DST by a variety of representations. Fundamentally, all such representations give three pieces of information about each data symbol: its name, its address or value, and its data type. DEBUG needs additional information about a data symbol, in particular its scope of declaration, but that information is implicit in the nesting structure of the DST as described above.

The name is given by a Counted ASCII string in the data symbol's DST

record. The value or address can be given by a five-byte encoding containing one byte of control information and a longword address, offset, or value. However, if this five-byte encoding is not adequate to describe the address or value, escapes to a more complex value specification later in the DST record are available. The data type may be represented by a one-byte type code, but if that is not adequate there are several escapes to a more complex type description elsewhere in the DST.

The standard five-byte value specification can specify any 32-bit or smaller literal value, any static byte address, any register address, and any address that can be formed by one indexing operation off a register or one indirection or both. If a VAX Standard Descriptor exists for the symbol in user memory, the five-byte encoding can describe the descriptor address by any of the above means; the actual data address is then retrieved from the descriptor.

The standard five-byte value specification is adequate for the bulk of all data symbols. However, there are cases when it is inadequate. It cannot describe literal values longer than 32 bits, it cannot describe very complex address computations, and it cannot describe bit addresses unless an appropriate descriptor is available in user memory. For these cases, the first byte of the five-byte encoding must have one of several special escape values. The remaining longword then contains (in most cases) a pointer to a more complex value specification later in the same DST record. That more complex value specification may consist of a VAX Standard Descriptor or a "VS-Follows" Value Specification. A VS-Follows Value Specification can, in the most complex case, contain a routine to be executed by DEBUG to compute the desired value or address. This routine may even call compiler-generated thunks when the complexity of the address computation so requires.

The details of these more complex value specifications are given in the section entitled "DST Value Specifications" later in this definition file. The point being made here is simply that the DST provides a simple and compact value specification mechanism that is adequate for all simple cases, but it also provides several escapes to arbitrarily complex DST Value Specifications. These complex value specifications are capable of describing all known address and value computations required by the languages supported by DEBUG.

Data type specifications are done in a similar way. For all simple, atomic data types, a single type byte describes the data type of a data symbol. However, there are several escape mechanisms for more complex data types. One mechanism is to take the type information from a VAX Standard Descriptor found either in user memory or in the DST. Another is to use a Separate Type Specification DST record for the data symbol. The data type is then described by a second DST record which immediately follows the Separate Type Specification DST record. This second record must be a Record Begin DST record (describing a record type), an Enumeration Type Begin DST record (describing an enumeration type), or a Type Specification DST record. A Type Specification DST record can describe any data type supported by DEBUG. It contains a DST Type Specification for the data type in question. This Type Specification may be an Indirect Type Specification, pointing to a DST record elsewhere in the DST that defines the data type. Alternatively, it may describe the desired data type directly and may be as complex as the data type requires.

!
!	DST Type Specifications are described in a separation section elsewhere
!	in this definition file. The point being made here is simply that the
!	simple one-byte type specification is available for simple data types,
!	but several escapes to arbitrarily complex DST type specifications are
!	available when the simple type specification is inadequate.

F I E L D A C C E S S M A C R O S

The following macros are used in defining BLISS field names for all data structures in the Debug Symbol Table. These macros supply the position, size, and sign-extension values when used in FIELD declarations for BLOCK and BLOCKVECTOR data structures. They are used instead of their numeric equivalents because they are clearer and less error-prone. The various generic forms (as specified by the letters in the names) are as follows:

A	Materialized address
L	Longword
W	Zero-extended word
B	Zero-extended byte
V	Zero-extended bit field
SW	Sign-extended word
SB	Sign-extended byte
SV	Sign-extended bit field

The "A" form should be used whenever the field being defined is such that only the address of the field may be materialized in a structure reference; that is, fetch and store operations on the field are not valid. An example of such a field is an ASCII string.

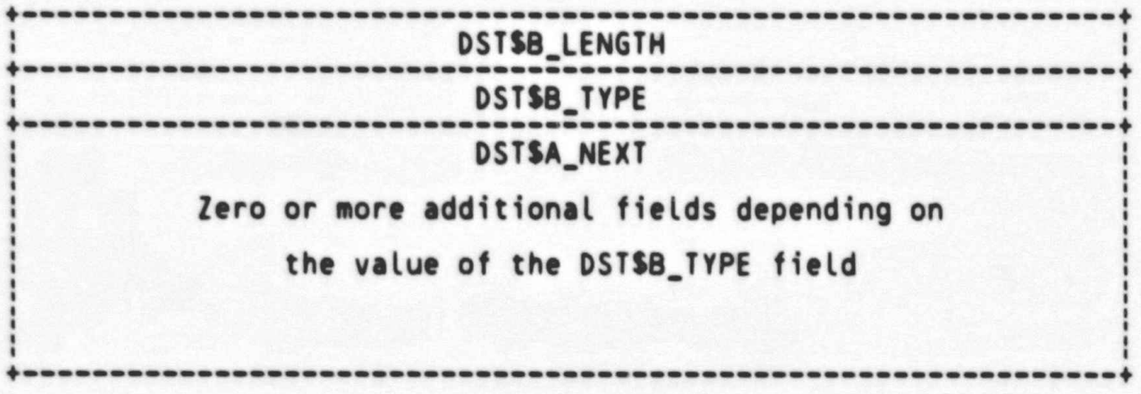
Each of the "V" and "SV" forms take one or two parameters. The first parameter is the bit position within the longword or byte and the second is the field size in bits. The second parameter is optional; if omitted, it defaults to 1. Thus V_(5) means bit 5 while V_(5,3) means the 3-bit field starting at bit 5 and ending at bit 7. Bit positions are counted from the low-order (least significant) end of the longword, starting at zero.

This following field access macros are used in DSTRECRDS.REQ. Their actual definitions are found in STRUCDEF.REQ, but are shown here for the convenience of the reader.

```
MACRO
        A_      =  0,  0,  0 %,         ! Address of a field
        L_      =  0, 32,  0 %,         ! Longword
        W_      =  0, 16,  0 %,         ! Word, zero-extended
        B_      =  0,  8,  0 %,         ! Byte, zero-extended
        V_(P,S) = P, %IF %NULL(S) %THEN ! %ELSE S %FI, 0 %, ! Unsigned
                                        !      bit field
        SW_     =  0, 16,  1 %,         ! Word, sign-extended
        SB_     =  0,  8,  1 %,         ! Byte, sign-extended
        SV_(P,S)= P, %IF %NULL(S) %THEN ! %ELSE S %FI, 1 %, ! Signed
                                        !      bit field

! Bring in the field access macro definitions from STRUCDEF.L32.

LIBRARY 'LIB$:STRUCDEF.L32';
```

THE DST RECORD HEADER FORMAT

All DST records have the same general format, consisting of a fixed two-byte header followed by zero or more fields whose format is determined by the DST record's type. This is the format of all DST records:



These fields appear in all DST records.

```
FIELD DST$HEADER_FIELDS =
        SET
        DST$B_LENGTH    = [ 0, B_ ],    ! The length of this DST record, not
                                        !      including this length byte
        DST$B_TYPE      = [ 1, B_ ],    ! The type of this DST record
        DST$A_NEXT      = [ 1, A_ ]     ! The next DST record starts at this
                                        !      location plus DST$B_LENGTH
        TES;
```

SUPPORTED VALUES FOR DST$B_TYPE

All supported values of the DST record type field (DST$B_TYPE) are listed here. If the value is in the range of DSC$K_DTYPE_LOWEST to DSC$K_DTYPE_HIGHEST, it is a VAX Standard Type Code and gives the data type of the object being defined. In this case, the record is a Standard Data DST Record or one of its variants. Otherwise, the type value must be in the range DST$K_LOWEST to DST$K_HIGHEST or it may be DST$K_BLI. In these cases, the type code denotes the type of the DST record and the format of the record is determined by type value. All other type codes are unsupported by DEBUG. The type codes between DSC$K_DTYPE_HIGHEST and DST$K_LOWEST are reserved for future use by Digital. The type codes in the range 192 - 255 are potentially reserved for use by customers, although DEBUG does not support any such type codes. DEBUG ignores all records with unsupported type codes.

VAX STANDARD TYPE CODES

As mentioned above, VAX Standard Type Codes can be used as DST record type codes for data symbols. The type code then gives the data type of the symbol in addition to indicating that the DST record has the Standard Data DST record format or a variant thereof.

```
! All VAX Standard Type Codes are listed here for convenience.  They are
! commented out since they are actually declared in STARLET.REQ.
LITERAL
        ! DSC$K_DTYPE_Z         = 0,    ! Unspecified (May not appear in DST).
        ! DSC$K_DTYPE_V         = 1,    ! Bit.
        ! DSC$K_DTYPE_BU        = 2,    ! Byte logical.
        ! DSC$K_DTYPE_WU        = 3,    ! Word logical.
        ! DSC$K_DTYPE_LU        = 4,    ! Longword logical.
        ! DSC$K_DTYPE_QU        = 5,    ! Quadword logical.
        ! DSC$K_DTYPE_B         = 6,    ! Byte integer.
        ! DSC$K_DTYPE_W         = 7,    ! Word integer.
        ! DSC$K_DTYPE_L         = 8,    ! Longword integer.
        ! DSC$K_DTYPE_Q         = 9,    ! Quadword integer.
        ! DSC$K_DTYPE_F         = 10,   ! Single-precision floating.
        ! DSC$K_DTYPE_D         = 11,   ! Double-precision floating.
        ! DSC$K_DTYPE_FC        = 12,   ! Complex.
        ! DSC$K_DTYPE_DC        = 13,   ! Double-precision Complex.
        ! DSC$K_DTYPE_T         = 14,   ! ASCII text string.
        ! DSC$K_DTYPE_NU        = 15,   ! Numeric string, unsigned.
        ! DSC$K_DTYPE_NL        = 16,   ! Numeric string, left separate sign.
        ! DSC$K_DTYPE_NLO       = 17,   ! Numeric string, left overpunched sign.
        ! DSC$K_DTYPE_NR        = 18,   ! Numeric string, right separate sign.
        ! DSC$K_DTYPE_NRO       = 19,   ! Numeric string, right overpunched sign
        ! DSC$K_DTYPE_NZ        = 20,   ! Numeric string, zoned sign.
```

```
        ! DSC$K_DTYPE_P         = 21,   ! Packed decimal string.
        ! DSC$K_DTYPE_ZI        = 22,   ! Sequence of instructions.
        ! DSC$K_DTYPE_ZEM       = 23,   ! Procedure entry mask.
        ! DSC$K_DTYPE_DSC       = 24,   !       Descriptor, used for arrays of
        !                               !           dynamic strings
        ! DSC$K_DTYPE_OU        = 25,   ! Octaword logical
        ! DSC$K_DTYPE_O         = 26,   ! Octaword integer
        ! DSC$K_DTYPE_G         = 27,   ! Double precision G floating, 64 bit
        ! DSC$K_DTYPE_H         = 28,   ! Quadruple precision floating, 128 bit
        ! DSC$K_DTYPE_GC        = 29,   ! Double precision complex, G floating
        ! DSC$K_DTYPE_HC        = 30,   ! Quadruple precision complex, H floating
        ! DSC$K_DTYPE_CIT       = 31,   ! COBOL intermediate temporary
        ! DSC$K_DTYPE_BPV       = 32,   ! Bound Procedure Value
        ! DSC$K_DTYPE_BLV       = 33,   ! Bound Label Value
        ! DSC$K_DTYPE_VU        = 34,   ! Bit Unaligned
        ! DSC$K_DTYPE_ADT       = 35,   ! Absolute Date-Time
        !                       = 36,   ! Unused (not supported by DEBUG)
        ! DSC$K_DTYPE_VT        = 37,   ! Varying Text
        !
        ! The next two values are used for range checking of the type values
        ! in DST entries.  They are used mainly in CASE statements.
        !
        DSC$K_DTYPE_LOWEST      = 1,    ! Lowest  DTYPE data type we support
        DSC$K_DTYPE_HIGHEST     = 37;   ! Highest DTYPE data type we support


!       INTERNAL TYPE CODES FOR DEBUG


        The following definitions are used internally in DEBUG, but are not
        supported in the DST.  They should be deleted here if they are made
        into standard VAX type codes declared in STARLET.REQ.  These numbers
        may change from one release of DEBUG to the next because they must
        always be larger than DSC$K_DTYPE_HIGHEST.


! Define DEBUG-internal type codes.

LITERAL
        DSC$K_DTYPE_AC          = 38,   ! ASCIC Text
        DSC$K_DTYPE_AZ          = 39,   ! ASCIZ Text
        DSC$K_DTYPE_TF          = 40,   ! Boolean True/False (length in bits)
        DSC$K_DTYPE_SV          = 41,   ! Signed bit-field (aligned)
        DSC$K_DTYPE_SVU         = 42,   ! Signed bit-field (unaligned)
        DSC$K_DTYPE_FIXED       = 43,   ! Fixed binary, used for FIXED in ADA
                                        !       and FIXED BINARY in PL/I.  This
                                        !       code is used the type conversion
                                        !       tables in DBGEVALOP.

        ! The following literals are used as CASE statement bounds internally
        ! in DEBUG for the range of DTYPE codes used.
        !
```

```
        DBG$K_MINIMUM_DTYPE     = 0,    ! Lowest internal DEBUG dtype value
        DBG$K_MAXIMUM_DTYPE     = 43;   ! Highest internal DEBUG dtype value

        ! The following definition is only used internally in DEBUG.  It is
        ! a DTYPE code that is temporarily put into a Value Descriptor to
        ! tell the address expression interpreter that the Value Descriptor
        ! came from a literal constant.  It does not have to be in the above
        ! range because it is only used during the parsing of address expres-
        ! sions.  After the address expression has been parsed, if the DTYPE
        ! is LITERAL, it is then changed to DSC$K_DTYPE_L.
        !
LITERAL
        DSC$K_DTYPE_LITERAL     = 191;  ! Value is from a literal constant


!       OTHER DST TYPE CODES
!
!       The following literals are the DST type codes other than VAX Standard
!       Type Codes which can appear in DST$B_TYPE.  Each indicates the format
!       of the record which contains it and most indicate the kind of object
!       being described by that record.  When new DST records are defined, the
!       type code is assigned by making DST$K_LOWEST one smaller and using that
!       value.  The type codes above DST$K_HIGHEST (191) are reserved, the idea
!       being that the DTYPEs 192 - 255 are architecturally reserved to users.
!       DEBUG ignores all DST records whose type codes are not DST$K_BLI, in
!       the range from DSC$K_DTYPE_LOWEST to DSC$K_DTYPE_HIGHEST, or in the
!       range DST$K_LOWEST TO DST$R_HIGHEST.


! Define all Additional Debug Symbol Table record type codes.  Note that the
! BLISS Special Cases record has code zero (for historical reasons).  All
! other type codes are in the range DST$K_LOWEST to DST$K_HIGHEST.
!
LITERAL
        DST$K_BLI       = 0,            ! BLISS Special Cases Record
        !-------------------------------!----------------------------
        DST$K_LOWEST    = 153,          ! Lowest numbered DST record in this
                                        !       range--used for range checking
        DST$K_VERSION   = 153,          ! Version Number Record
        DST$K_COBOLGBL  = 154,          ! COBOL Global Attribute Record
        DST$K_SOURCE    = 155,          ! Source File Correlation Record
        DST$K_STATLINK  = 156,          ! Static Link Record
        DST$K_VARVAL    = 157,          ! Variant Value Record
        DST$K_BOOL      = 158,          ! Atomic object of type BOOLEAN,
                                        !       Allocated one byte.
                                        !         low order bit = 1 if TRUE,
                                        !         low order bit = 0 if FALSE.
        DST$K_EXTRNXT   = 159,          ! External-Is-Next Record (Obsolete)
        DST$K_GLOBNXT   = 160,          ! Global-Is-Next record (Obsolete)
        DSC$K_DTYPE_UBS = 161,          ! DEBUG internal use only (unaligned
                                        !       bit string) (Obsolete)
        DST$K_PROLOG    = 162,          ! Prolog Record
```

```
        DST$K_SEPTYP    = 163,          ! Separate Type Specification Record
        DST$K_ENUMELT   = 164,          ! Enumerated Type Element Record
        DST$K_ENUMBEG   = 165,          ! Enumerated Type Begin Record
        DST$K_ENUMEND   = 166,          ! Enumerated Type End Record
        DST$K_VARBEG    = 167,          ! Variant Set Begin Record
        DST$K_VAREND    = 168,          ! Variant Set End Record
        DST$K_OVERLOAD  = 169,          ! Overloaded Symbol record
        DST$K_DEF_LNUM  = 170,          ! Definition Line Number Record
        DST$K_RECBEG    = 171,          ! Record Begin Record
        DST$K_RECEND    = 172,          ! Record End Record
        DST$K_CONTIN    = 173,          ! Continuation Record
        DST$K_VALSPEC   = 174,          ! Value Specification Record
        DST$K_TYPSPEC   = 175,          ! Type Specification Record
        DST$K_BLKBEG    = 176,          ! Block Begin Record
        DST$K_BLKEND    = 177,          ! Block End Record
        DST$K_COB_HACK  = 178,          ! COBOL Hack Record (Obsolete)
        !               = 179,          ! Reserved to DEBUG
        !               = 180,          ! Reserved to DEBUG
        DST$K_ENTRY     = 181,          ! Entry Point Record
        DST$K_LINE_NUM_REL_R11          ! Threaded Code PC-Correlation
                        = 182,          !     Record (Obsolete)
        DST$K_BLIFLD    = 183,          ! BLISS Field Record
        DST$K_PSECT     = 184,          ! PSECT Record
        DST$K_LINE_NUM  = 185,          ! Line Number PC-Correlation Record
        DST$K_LBLORLIT  = 186,          ! Label-or-Literal Record
        DST$K_LABEL     = 187,          ! Label Record
        DST$K_MODBEG    = 188,          ! Module Begin Record
        DST$K_MODEND    = 189,          ! Module End Record
        DST$K_RTNBEG    = 190,          ! Routine Begin Record
        DST$K_RTNEND    = 191,          ! Routine End Record
        DST$K_HIGHEST   = 191;          ! Highest numbered DST record in this
                                        !     range--used for range checking

! NOTE TO DEVELOPERS:
!
! New DST Records should not be added at this end of the DST record number
! range.  VAX Standard Type Codes 192 - 255 are reserved to users.  Hence
! DEBUG does not use type codes in that range, even though DEBUG does not
! support user-defined type codes.  New DST record numbers should be allocated
! by decrementing DST$K_LOWEST and using that number for the new DST record.
```

M O D U L E D S T R E C O R D S

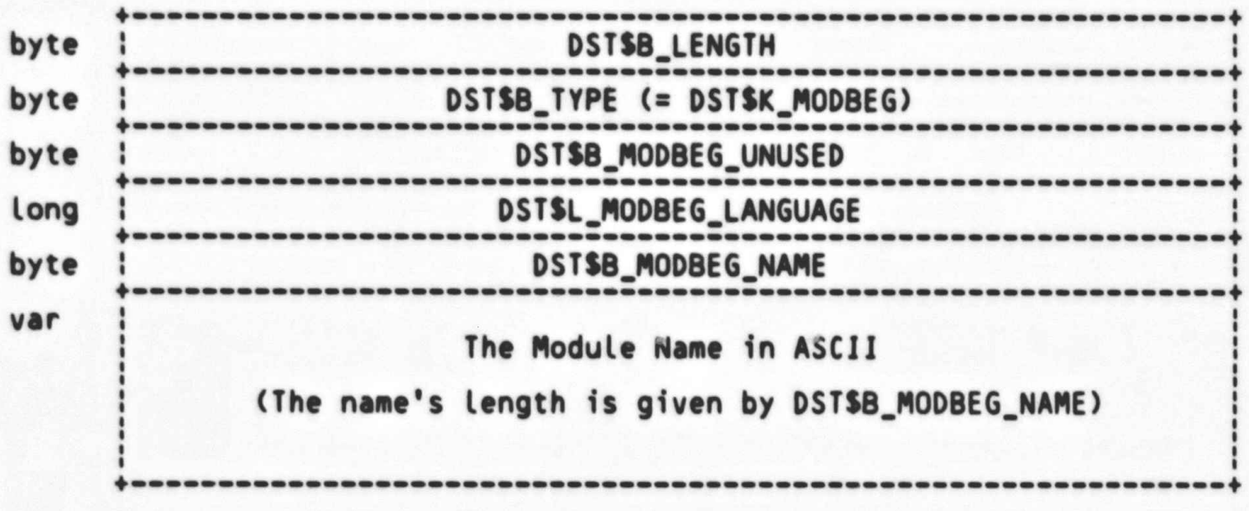
The Debug Symbol Table for each separately compiled module must be enclosed within a Module-Begin/Module-End pair of DST records. The Module Begin DST record must thus be the very first DST record for any separately compiled module (i.e., any object file) and the Module End DST record must be the very last DST record for the module. Only one Module-Begin/Module-End pair is allowed in what the linker sees as a single object module. (If multiple Module-Begin/Module-End pairs are included in one object module, DEBUG will only see the first such pair and ignore the rest because the linker will only tell DEBUG about the location of the first Module Begin record.)

The Module-Begin/Module-End pair defines a symbolic scope which contains all symbols defined by DST records within that pair. The module has the name given in the Module Begin DST record. The language of the object module is also encoded in the Module Begin record.

THE MODULE BEGIN DST RECORD

The Module Begin DST Record marks the beginning of the DST for a module. This DST record also gives the name of the module and the source language in which the module was written. The Module Begin DST Record must be the the first DST record of every compilation unit ("module") and it must be matched by a Module End DST Record that ends the DST for that module. Only one Module Begin DST Record is allowed to appear in the DST for a separately compiled object module.

This is the format of the Module Begin DST Record:




```
! Define the fields and size of the Module Begin DST Record.

FIELD DST$MODBEG_FIELDS =
        SET
        DST$B_MODBEG_UNUSED     = [ 2, B_ ],    ! Unused--Must Be Zero
        DST$L_MODBEG_LANGUAGE   = [ 3, L_ ],    ! Language code of language in
                                                !       which module was written
        DST$B_MODBEG_NAME       = [ 7, B_ ]     ! Count byte in name counted
                                                !       ASCII string
        TES;

LITERAL
        DST$K_MODBEG_SIZE       = 8;    ! Size in bytes of the fixed part of
                                        !       the Module Begin DST record

! Define all the language codes that may appear in the DST$L_MODBEG_LANGUAGE
! field of the Module Begin DST record.  (Note that DEBUG may not actually
! support all languages that have language codes.)
!
LITERAL
```

```
        DST$K_MIN_LANGUAGE = 0,         ! Smallest language code
        DST$K_MACRO      = 0,           ! Macro
        DST$K_FORTRAN    = 1,           ! Fortran
        DST$K_BLISS      = 2,           ! Bliss
        DST$K_COBOL      = 3,           ! Cobol
        DST$K_BASIC      = 4,           ! Basic
        DST$K_PLI        = 5,           ! PL/I
        DST$K_PASCAL     = 6,           ! Pascal
        DST$K_C          = 7,           ! C
        DST$K_RPG        = 8,           ! RPG
        DST$K_ADA        = 9,           ! Ada
        DST$K_UNKNOWN    = 10,          ! Language Unknown
        DST$K_MAX_LANGUAGE = 10;        ! Largest language code

! Here also we define all the same language codes using names with the DBG$
! prefix.  This prefix is used in DEBUG for historical reasons.  These names
! may eventually be discarded.
!
LITERAL
        DBG$K_MIN_LANGUAGE = DST$K_MIN_LANGUAGE, ! Smallest language code
        DBG$K_MACRO      = DST$K_MACRO,  ! Macro
        DBG$K_FORTRAN    = DST$K_FORTRAN,! Fortran
        DBG$K_BLISS      = DST$K_BLISS,  ! Bliss-32
        DBG$K_COBOL      = DST$K_COBOL,  ! Cobol
        DBG$K_BASIC      = DST$K_BASIC,  ! Basic
        DBG$K_PLI        = DST$K_PLI,    ! PL/I
        DBG$K_PASCAL     = DST$K_PASCAL, ! Pascal
        DBG$K_C          = DST$K_C,      ! C
        DBG$K_RPG        = DST$K_RPG,    ! RPG
        DBG$K_ADA        = DST$K_ADA,    ! Ada
        DBG$K_UNKNOWN    = DST$K_UNKNOWN,! Language Unknown
        DBG$K_MAX_LANGUAGE = DST$K_MAX_LANGUAGE; ! Largest language code


! Language UNKNOWN requires some special explanation.  DEBUG supports "unknown"
! languages with a standard set of DEBUG functionality.  This standard set in-
! cludes all language-independent functionality plus "vanilla-flavored" language
! expressions.  Identifiers are assumed to allow A - Z, 0 - 9, $, and _.  Symbol
! references may include subscripting (using round () or square [] parentheses)
! and record component selection (using dot-notation as in A.B.C).  Most simple
! operators are allowed in language expressions.
!
! While not officially supported, language UNKNOWN is intended as an escape for
! compilers which do not yet have true DEBUG support.  By specifying language
! code DST$K_UNKNOWN in the DST$L_MODBEG_LANGUAGE field, such languages can
! take advantage of whatever support DEBUG provides for unknown languages.  If
! and when true DEBUG support is provided, a new language code for the new
! language can be allocated by incrementing DST$K_MAX_LANGUAGE by one and as-
! signing that language code to the new language.
!
! DEBUG treats any out-of-range language code in the Module Begin DST record as
! being equivalent to language UNKNOWN.  Use of the DST$K_UNKNOWN language code
! or any out-of-range language code is intended for internal use by Digital
! only.  DEBUG's unknown-language support is not officially supported and is
! subject to possibly incompatible changes in future releases of DEBUG.
```

```
!
! Internally, DEBUG treats the language code as a byte value.  Hence any lan-
! guage code above 255 is truncated to its low-order eight bits.
```

```
!       THE MODULE END DST RECORD
!
!
!       The Module End DST Record must be the last DST record in the DST for a
!       compilation unit.  Its sole purpose is to mark the end of the DST for
!       a separately compiled object module.  There can be only one Module End
!       DST record per module, matching the previous Module Begin DST record.
!       This is its format:
!
!       +---------------------------------------------------------------+
! byte  !                      DST$B_LENGTH (= 1)                       !
!       +---------------------------------------------------------------+
! byte  !                    DST$B_TYPE (= DST$K_MODEND)                !
!       +---------------------------------------------------------------+
!
! Define the size in bytes of the Module End DST Record.
!
LITERAL
        DST$K_MODEND_SIZE       = 2;    ! Size of Module End record in bytes
```

R O U T I N E D S T R E C O R D S

A routine is represented in the Debug Symbol Table by a pair of DST records, namely a Routine Begin DST record which is matched with a later Routine End DST record. All DST records between the Routine Begin and the Routine End DST records represent the symbols that are declared in that routine or in nested routines or blocks. Nested routines are represented in the DST by nested Routine-Begin/Routine-End pairs. Lexical blocks (BEGIN-END blocks or the like, depending on the language) may also be nested freely outside or inside routines, provided all blocks and routines are properly nested.

Consider the following example of nested blocks and routines. If routine R1 contains a nested routine R2 and a lexical block B1 and if block B1 contains routine R3 and Block B2, the DST would have the following sequence of DST records:

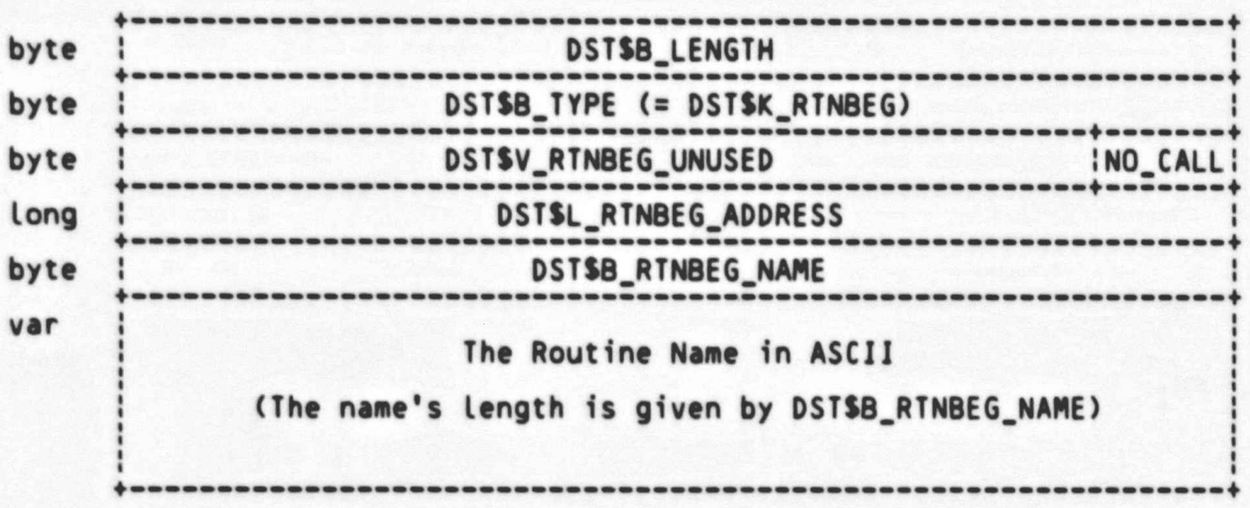
```
Module Begin for whole module
  ...module-level data DST records...
  Routine Begin for R1
    ...local data DST records for R1...
    Routine Begin for R2
      ...local data DST records for R2...
    Routine End for R2
    Block Begin for B1
      ...local data DST records for B1...
      Routine Begin for R3
        ...local data DST records for R3...
      Routine End for R3
      Block Begin for B2
        ...local data DST records for B2...
      Block End for B2
    Block End for B1
  Routine End for R1
Module End for whole module
```

In addition to defining a symbol scope, the Routine-Begin/Routine-End pair defines the name and address range of the corresponding routine. The name and start address is found in the Routine Begin DST record and the byte length of the routine is found in the Routine End DST record. It is assumed that the start address is also the entry point to the routine. The Routine Begin record also indicates whether the routine uses a CALLS/CALLG linkage or a JSB/BSB linkage.

THE ROUTINE BEGIN DST RECORD

The Routine Begin DST record marks the beginning of a routine and the associated scope. This record contains the routine's name and start address and indicates whether the routine is a CALLS/CALLG routine or a JSB/BSB routine. It must be matched by a Routine End DST record later in the DST, except if the language of the current module is MACRO. (Since MACRO routines have entry points but no well defined end points, the Routine End record can and must be omitted for this language. This exception applies to no other language.)

This is the format of the Routine Begin DST record:




```
! Define the fields and size of the Routine Begin DST record.

FIELD DST$RTNBEG_FIELDS =
        SET
        DST$V_RTNBEG_UNUSED  = [ 2, V_(0, 7) ], ! Unused--Must Be Zero
        DST$V_RTNBEG_NO_CALL = [ 2, V_(7, 1) ], ! This bit is set if this rou-
                                                !      tine is invoked with a
                                                !      JSB or BSB rather a CALLS
                                                !      or CALLG instruction
        DST$L_RTNBEG_ADDRESS = [ 3, L_  ],      ! The routine's start address
                                                !      and entry point address
        DST$B_RTNBEG_NAME    = [ 7, B_ ]        ! The count byte of the rou-
                                                !      tine's Counted ASCII name
        TES;

LITERAL
        DST$K_RTNBEG_SIZE       = 8;    ! Byte size of the fixed part of the
                                        !      Routine Begin DST record
```

```
!       THE ROUTINE END DST RECORD
!
!       The Routine End DST Record marks the end of a routine's scope in the
!       DST.  It also contains the byte length of the routine's code.  (Note
!       that Routine End DST records must be omitted for language MACRO but
!       are mandatory for all other languages.)  This is the format of the
!       Routine End DST record:
!
!       +---------------------------------------------------------------+
!  byte !                      DST$B_LENGTH (= 6)                       !
!       +---------------------------------------------------------------+
!  byte !                  DST$B_TYPE (= DST$K_RTNEND)                  !
!       +---------------------------------------------------------------+
!  byte !                      Unused (Must Be Zero)                    !
!       +---------------------------------------------------------------+
!  long !                       DST$L_RTNEND_SIZE                       !
!       +---------------------------------------------------------------+
!
!
! Define the fields of the Routine End DST record.

FIELD DST$RTNEND_FIELDS =
        SET
        DST$L_RTNEND_SIZE  = [ 3, L_  ] ! The length of the routine in bytes
        TES;
```

L E X I C A L B L O C K D S T R E C O R D S

A "Lexical Block" is any programming language construct other than a routine that defines a scope within which symbols can be declared. What distinguishes a "block" from a "routine", from DEBUG's point of view, is that a block is always entered by jumping to it or simply falling into it while a routine is always entered by a call instruction of some sort. A routine has a entry point that can be called; a block does not. Hence BEGIN-END blocks in BLISS and PL/I are blocks and so are Paragraphs and Sections in COBOL. Subroutines, functions, and procedures, on the other hand, are "routines".

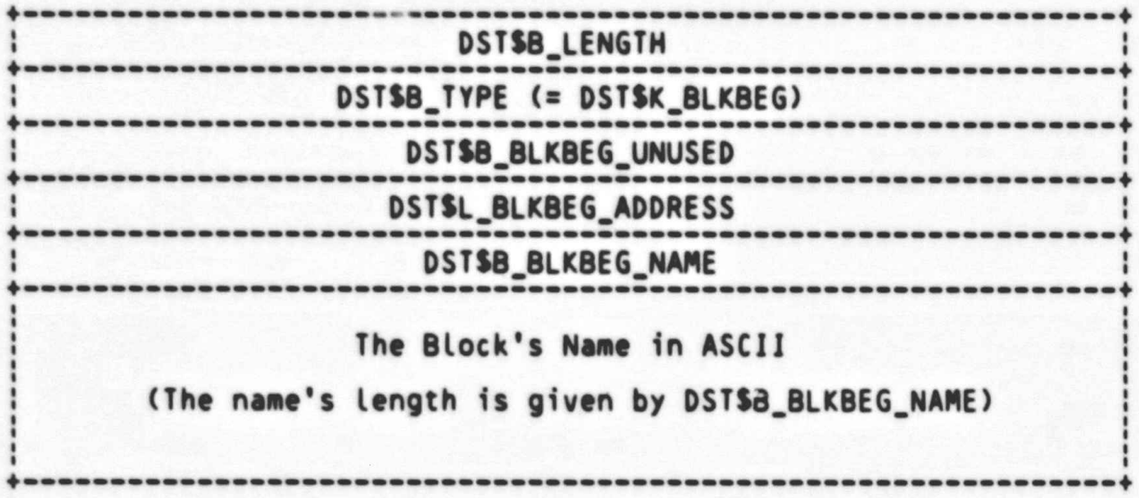
Blocks and routines do have one thing in common, however. Both define syntactic units within which other symbols can be defined. The purpose of representing blocks in the DST is to define the scopes they enclose and to give the address ranges of the corresponding bodies of code.

A lexical block is represented in the Debug Symbol Table by a pair of DST records, namely a Block Begin DST record which is matched with a later Block End DST record. All DST records between the Block Begin and the Block End DST record represent the symbols that are declared in that lexical block or in nested routines or blocks. Nested blocks are represented in the DST by properly nested Block-Begin/Block-End pairs. Routines and blocks may freely be nested within one another, using the appropriate proper nesting of the corresponding Begin and End DST records.

The start address of a block's code is given in the Block Begin DST record and the byte length of that code is given in the Block End DST record. The name of the block is given in the Block-Begin record. If a block has no name (which is common for BEGIN-END blocks), the null name is given (the name of length zero). Blocks with null names cannot be explicitly referenced in DEBUG, but line numbers within such blocks can be used to specify breakpoint locations or symbol scopes.

THE BLOCK BEGIN DST RECORD

The Block Begin DST Record marks the beginning of a lexical block and the associated scope. This record contains the block's name and start address. It must be matched by a Block End DST record later in the DST. This is the format of the Block Begin DST record:




```
! Define the fields of the Block Begin DST record.
!
FIELD DST$BLKBEG_FIELDS =
        SET
        DST$B_BLKBEG_UNUSED     = [ 2, B_ ],    ! Unused--Must Be Zero
        DST$L_BLKBEG_ADDRESS    = [ 3, L_ ],    ! The block's start address
        DST$B_BLKBEG_NAME       = [ 7, B_ ]     ! The count byte of the block's
                                                !       Counted ASCII name
        TES;
```

```
!       THE BLOCK END DST RECORD
!
!
!       The Block End DST Record marks the end of a lexical block's scope in
!       the DST.  It also contains the byte length of the block's code.  This
!       is the format of the Block End DST record:
!
!       +---------------------------------------------------------------+
! byte  !                    DST$B_LENGTH (= 6)                         !
!       +---------------------------------------------------------------+
! byte  !                  DST$B_TYPE (= DST$K_BLKEND)                  !
!       +---------------------------------------------------------------+
! byte  !                    Unused (Must Be Zero)                      !
!       +---------------------------------------------------------------+
! long  !                     DST$L_BLKEND_SIZE                         !
!       +---------------------------------------------------------------+
!
!
! Define the fields of the Block End DST record.
!
FIELD DST$BLKEND_FIELDS =
        SET
        DST$L_BLKEND_SIZE  = [ 3, L_  ] ! The byte length of the lexical block
        TES;
```

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

Data symbols are represented in the Debug Symbol Table by data DST records which come in several varieties. All such DST records give three pieces of information about each symbol: the data type of the symbol, the value or address or the symbol, and the name of the symbol.

The Standard Data DST record is the simplest form of data DST symbol record and is used for most ordinary atomic data objects. It represents the data type by a one-byte VAX Standard Type Code. It represents the value or address of the symbol by a simple five-byte encoding capable of specifying 32-bit literal values, absolute addresses, register locations, and addresses computed as offsets from a register, possibly including indirection. It is also possible to specify that the computed address is the address of a VAX Standard Descriptor for the data symbol. Finally, the name is represented as a Counted ASCII character string.

There are several reasons why a Standard Data DST record may not be adequate to represent a data symbol. First, the symbol's data type may be too complicated to represent by a one-byte type code. In this case, one of several available escape mechanisms must be used so that expanded type information can be included in the symbol's DST information. Second, if the symbol is a literal (a named constant), its value may be too large to fit in one longword. In this case, an expanded value specification must be used. And third, if the symbol is a variable, its address may be specified by a more complicated computation than can be represented in the Standard Data DST record. In this case, an escape to a more complicated value specification must be used.

Expanded type specifications come in three main forms: Descriptor Format DST records, Separate Type Specification DST records, and various specialized DST records that handle various special kinds of data types such as record structures, enumeration types, and BLISS structures.

Descriptor Format DST records are used when the data object must be described by a VAX Standard Descriptor and has a static address. A packed decimal data object, for example, must be described by a descriptor that specifies the object's length and scale factor. If a descriptor exists in user memory at run-time, the Standard Data DST record can be used, but otherwise it is necessary to include the descriptor directly in the DST within a Descriptor Format DST record. These DST records are used for all static arrays and other data objects that can be described by VAX Standard Descriptors.

For data types that can be described by neither one-byte type codes nor VAX Standard Descriptors, a Separate Type Specification DST record must be used. In this case the DST record's type field indicates that the type specification is found is a separate DST record which immediately follows the present DST record. The DST record that follows must be a Type Specification, Record Begin, or Enumeration Type Begin

DST record. These records can describe all data types supported by DEBUG in full detail.

As mentioned above, the third data type "escape" mechanism is to use one of a number of specialized DST records that describe data symbols of special kinds. BLISS structures and fields, for example, are described by special DST records, as are enumeration type elements. These DST records will not be further described in this section; they are described elsewhere in this definition file.

Expanded "Value Specifications" must be used for data symbols whose values or addresses are too long or too complicated to be described by the Standard Data DST record. A D-Floating constant, for example, has too large a value (8 bytes) to fit in a Standard Data DST record. A "based variable" in PL/I may require a complicated computation or even a call on a compiler-generated thunk to compute the variable's address. For these and other cases, a Trailing Value Specification DST record must be used. Such a record includes a Value Specification which may be arbitrarily complex.

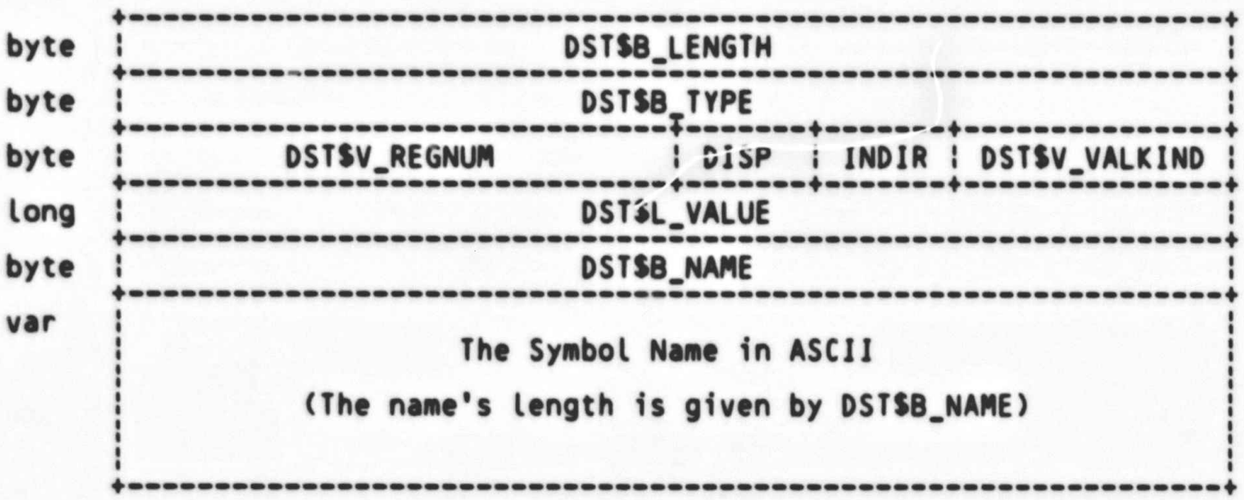
Trailing Value Specification DST records are sometimes used to specify both type and address information. An array with dynamic array bounds, for instance, must be described in the DST if no descriptor exists in user memory at run-time. A Trailing Value Specification can be used to compute the entire descriptor for such an array at DEBUG-time. The descriptor then gives both the array address and type information such as the element type and the array bounds.

THE STANDARD DATA DST RECORD

The Standard Data DST record is used to describe most simple scalar data objects such as integers, floating-point numbers, and complex numbers. The data type is represented by the one-byte VAX Standard Type Code in the DST$B_TYPE field. The value DST$K_BOOL is also accepted; it denotes that the data symbol is a Boolean variable or value which is TRUE if the low-order bit is set and FALSE otherwise.

The value specification in the Standard Data DST record indicates the symbol's value or address or how to compute the symbol's address. The details are found below.

This is the format of the Standard Data DST record:



Define the fields of the Standard Data DST record. These fields are also used in many other DST records of similar formats.

```
FIELD DST$STD_FIELDS =
        SET
        DST$B_VFLAGS     = [ 2, B_ ],     ! Value-Flags (access information)
        DST$V_VALKIND    = [ 2, V_(0,2)], ! How to interpret the specified value
        DST$V_INDIRECT   = [ 2, V_(2) ],  ! Set if address of address is produced
                                          !      by indicated computation (do an
                                          !      indirection to compute address)
        DST$V_DISP       = [ 2, V_(3) ],  ! Set if content of DST$L_VALUE is used
                                          !      as a displacement off a register
                                          !      specified in DST$V_REGNUM
        DST$V_REGNUM     = [ 2, V_(4,4)], ! Number of register used in displace-
                                          !      ment mode addressing
        DST$L_VALUE      = [ 3, L_  ],    ! Value, address, or bit offset
        DST$B_NAME       = [ 7, B_ ]      ! Count byte of the symbol name field,
                                          !      a counted ASCII string
```

```
        TES;

! Define all special values that may appear in the DST$B_VFLAGS field.  If one
! of these values appears in that field, the DST$L_VALUE field has some special
! meaning indicated by the special value.  In such cases, the DST$B_VFLAGS sub-
! fields have no meaning.  Not all of these special values may appear in a
! Standard Data DST record (see the comments below), but they are all listed
! here for completeness.  Note that these values (with one exception) all have
! the top four bits set--hence they cannot be normal VFLAGS values since the
! REGNUM field cannot contain 15 (indicating the PC) in a normal VFLAGS value.
!
LITERAL
        DST$K_VFLAGS_NOVAL      = 128,  ! A flag which indicates that no value
                                        !       is specified, i.e. the object
                                        !       being described is a type.  This
                                        !       value may only appear in a Record
                                        !       Begin DST record.
        DST$K_VFLAGS_UNALLOC    = 249,  ! This value is DST$B_VFLAGS signals a
                                        !       data item that was never
                                        !       allocated (and hence has no
                                        !       address). For example, PASCAL
                                        !       does not allocate variables
                                        !       that are not referenced.
        DST$K_VFLAGS_DSC        = 250,  ! This value in DST$B_VFLAGS signals a
                                        !       Descriptor Format DST record
        DST$K_VFLAGS_TVS        = 251,  ! This value in DST$B_VFLAGS signals a
                                        !       Trailing Value Spec DST record
        DST$K_VS_FOLLOWS        = 253,  ! Value Specification Follows (allowed
                                        !       only in a Trailing Value Spec)
        DST$K_VFLAGS_BITOFFS    = 255;  ! A flag indicating that DST$L_VALUE
                                        !       contains a bit offset (used
                                        !       only for record components)

! Provided the DBG$B_VFLAGS field does not have one of the above special values,
! the DBG$V_VALKIND field indicates what kind of value or address is computed
! by the value computation.  The possible values of this field are defined here.
!
LITERAL
        DST$K_VALKIND_LITERAL   = 0,    ! DST$L_VALUE contains a literal value
        DST$K_VALKIND_ADDR      = 1,    ! Computation produces the address of
                                        !       the data object
        DST$K_VALKIND_DESC      = 2,    ! Computation produces the address of a
                                        !       VAX Standard Descriptor for the
                                        !       data object
        DST$K_VALKIND_REG       = 3;    ! Value is contained in the register
                                        !       whose number is in DST$L_VALUE

! If the DST$K_VFLAGS field does not contain one of the special values listed
! above, then the computation that produces the value or address of the data
! object proceeds as follows:
!
!   1.  If the VALKIND field contains DST$K_VALKIND_LITERAL, the symbol is a
!       constant whose value is given by the DST$L_VALUE field.  Such constants
```

can be up to 32 bits long.

2. If the VALKIND field contains DST$K_VALKIND_REG, the symbol is a variable bound to a register. The register number of that register is given by the DST$L_VALUE field.

3. Otherwise, the symbol is a variable with a non-register address. To compute that address, the DST$L_VALUE field is picked up.

4. If the DST$V_DISP bit is set, the contents of the register whose register number is given by the DST$V_REGNUM field is added to the value picked up from the DST$L_VALUE field.

5. If the DST$V_INDIRECT bit is set, the address computed so far is treated as the address of a pointer that points to the actual data object. In other words, an indirection is done.

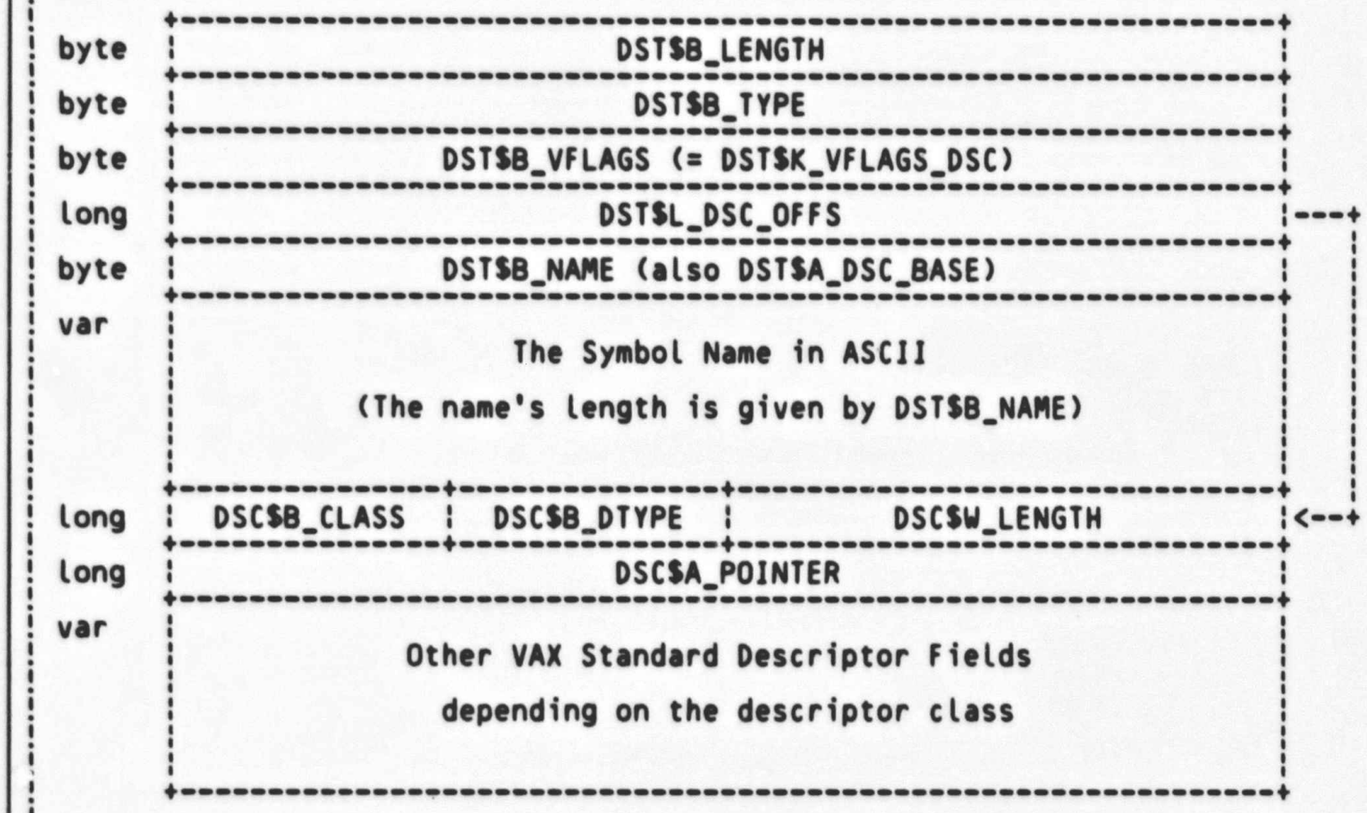
6. If the value of the VALKIND field is DST$K_VALKIND_ADDR, the address computed so far is treated as the address of the data object.

7. If the value of the VALKIND field is DST$K_VALKIND_DESC, the address computed so far is treated as the address of a VAX Standard Descriptor for the data object. The actual address of the object, along with its other attributes such as type and size, must therefore be retrieved from that descriptor.

As this description indicates, moderately complicated address computations can be specified in the Standard Data DST record. For example, the address of the second formal parameter to a routine, passed by reference, can be described by making DST$V_REGNUM = 12 (for register AP), DST$L_VALUE = 8 (to indicate an offset of 8 bytes from AP to get at the second longword in the argument vector), DST$V_DISP = 1 (to indicate that DST$L_VALUE is to be treated as a displacement off AP), and DST$V_INDIRECT = 1 (to indicate an indirection since the argument is passed by reference). DST$V_VALKIND = DST$K_VALKIND_ADDR in this case. If the parameter were passed by descriptor, however, DST$V_VALKIND should be DST$K_VALKINKD_DESC, with all other fields having the same values as in the passed-by-reference case.

THE DESCRIPTOR FORMAT DST RECORD

The Descriptor Format DST record is used when a VAX Standard Descriptor must be included in the DST for a static symbol. It includes the descriptor directly in the DST record right after the name field. This record is essentially identical to the Standard Data DST record except that the DST$B_VFLAGS field has the special value DST$K_VFLAGS_DSC and the DST$L_VALUE field is a relative byte offset to the VAX descriptor later in the record. This is the format:

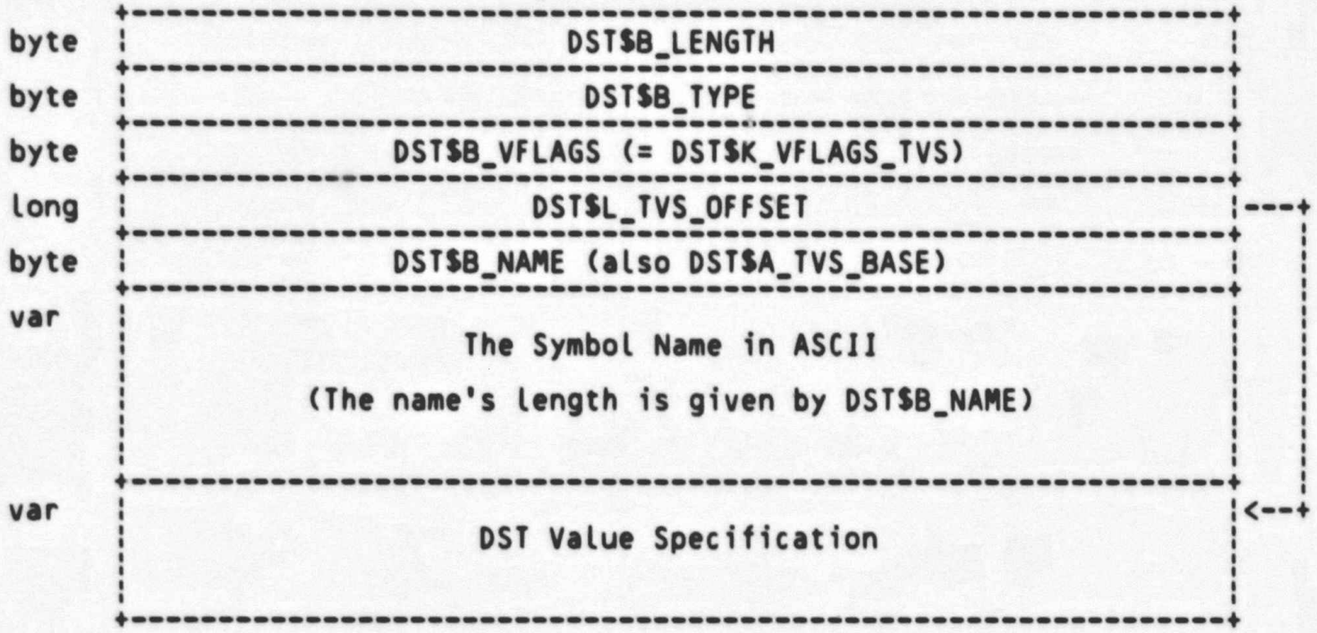



```
! Define the fields of the Descriptor Format DST record.

FIELD DST$DSC_FIELDS =
        SET
        DST$L_DSC_OFFS  = [ 3, L_  ],   ! Offset in bytes to descriptor
                                        !       from DST$A_DSC_BASE
        DST$A_DSC_BASE  = [ 7, A_  ]    ! Descriptor starts at this loc-
                                        !       ation + DST$L_DSC_OFFS
        TES;
```

```
! Note that the address of the descriptor is computed as follows:
!
!       DST_RECORD[DST$A_DSC_BASE] + .DST_RECORD[DST$L_DSC_OFFS]
```

THE TRAILING VALUE SPECIFICATION DST RECORD

The Trailing Value Specification DST record is used when an expanded value specification is needed to compute a data symbol's value or address. It includes a Value Specification directly in the DST record right after the name field. This record is essentially identical to the Standard Data DST record except that the DST$B_VFLAGS field has the special value DST$K_VFLAGS_TVS and the DST$L_VALUE field is a relative byte offset to the Value Specification later in the record. This is the format:

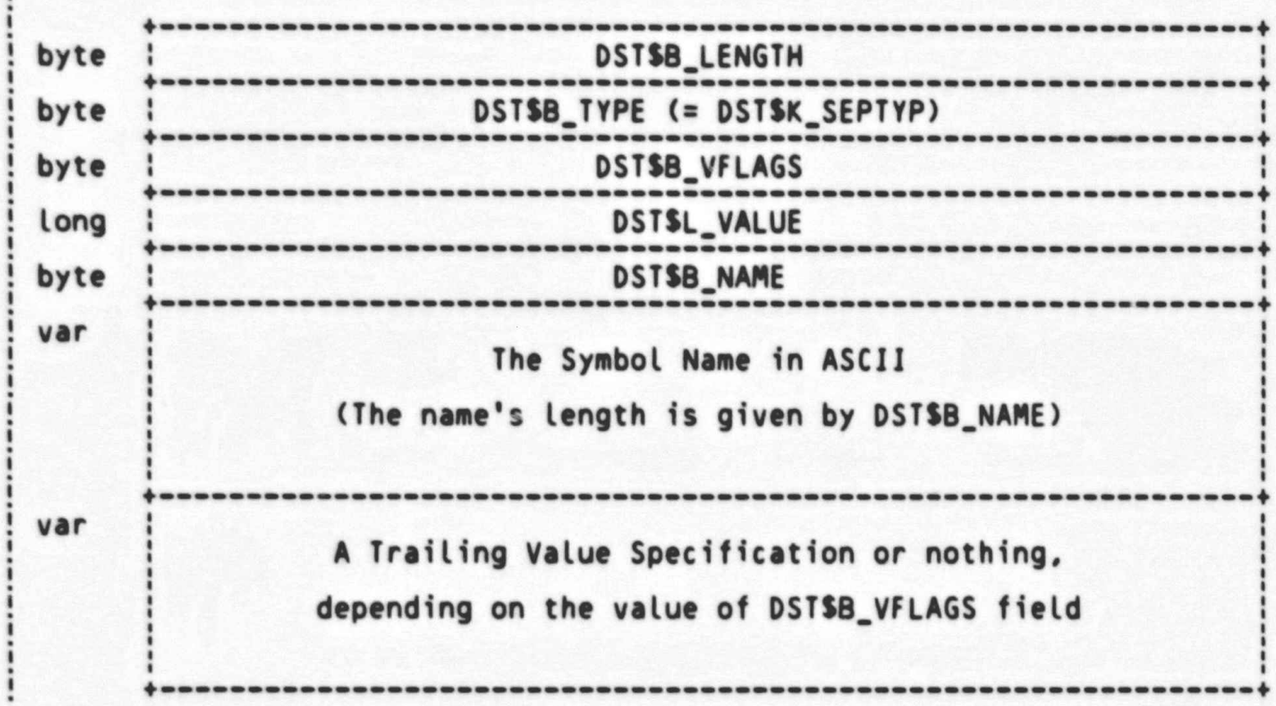



```
! Define the fields of the Trailing Value Specification DST record.

FIELD DST$TVS_FIELDS =
        SET
        DST$L_TVS_OFFSET = [ 3, L_  ],  ! Offset in bytes to trailing Value Spec
                                        !       from DST$A_TVS_BASE
        DST$A_TVS_BASE   = [ 7, A_  ]   ! Trailing Value Spec starts at this
                                        !       location + .DST$L_TVS_OFFSET
        TES;

! Note that the address of the trailing Value Specification is computed as
! follows:

        DST_RECORD[DST$A_TVS_BASE] + .DST_RECORD[DST$L_TVS_OFFSET]
```

```
! Also note that Value Specifications are described in a separate section
! later in this definition file.
```

THE SEPARATE TYPE SPECIFICATION DST RECORD

The Separate Type Specification DST record is used when the data type of the symbol being described is too complex to be described by a a one-byte type code or a VAX Standard Descriptor. This DST record must be immediately followed by a Type Specification, Record Begin, or Enumeration Type Begin DST record which describes the data type of the data symbol. (Only Continuation DST records may intervene.) The format of the Separate Type Specification DST record is essentially identical to that of the Standard Data DST record. It may contain a Trailing Value Specification if necessary to describe the symbol's value or address. This is the format of the record:



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

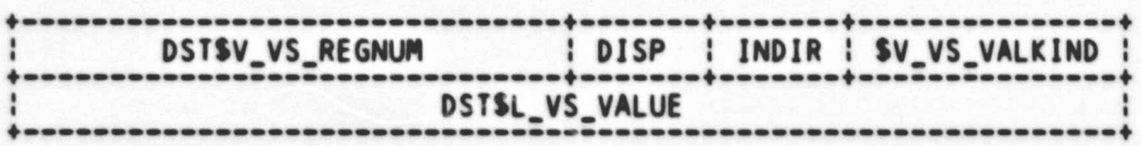
A DST Value Specification specifies the value or address of some symbol. Value Specification can occur in a number of places in the Debug Symbol Table. The simplest forms of Value Specifications occur in the Standard Data DST record. A somewhat more complicated form occurs in Descriptor Format DST records where a VAX Standard Descriptor is included in the DST record to give more complete address information (and type information). The Trailing Value Specification DST record has a simple five-byte Value Specification at the beginning of the record which points to a more complex Value Specification at the end of the record. That more complex Value Specification can be any kind of Value Specification, including the most general forms.

In addition, Value Specifications may occur in a number of Type Specifications. In these cases, they typically generate values (as opposed to addresses), such as subrange bounds for a subrange data type, or they generate full VAX Standard Descriptors in order to specify some sort of data type, such as a dynamic array.

All Value Specifications start with one byte, the DST$B_VS_VFLAGS field. In Standard Data DST records, this field and the DST$B_VFLAGS field are synonymous. If this field has one of the special values DST$K_VFLAGS_xx described in the Standard Data DST Record section above, the format of the Value Specification depends on that value. Otherwise the VFLAGS field is interpreted as a set of subfields, namely DSTV_VS_REGNUM, DSTV_VS_DISP, DST$V_VS_INDIRECT, and DST$K_VS_VALKIND. This is also described in detail in the Standard Data DST Record section above.

STANDARD VALUE SPECIFICATIONS

As indicated above, if the DST$B_VS_VFLAGS field does not have a special value, the Value Specification is a Standard Value Specification and has the following structure:



Define the fields of the various kinds of Value Specifications. Also define the declaration macro.

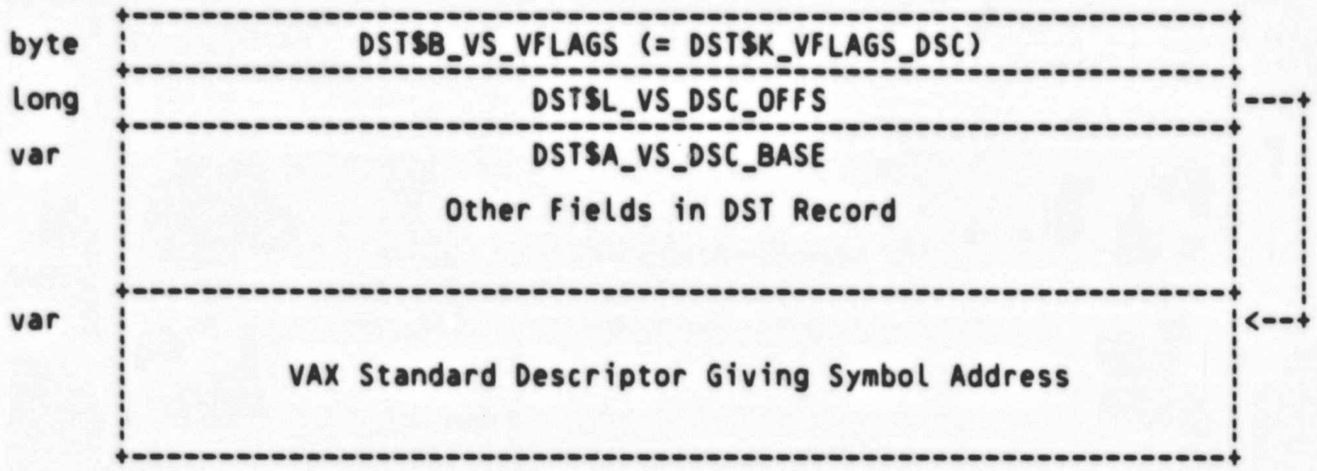
```
FIELD DST$VS_HDR_FIELDS =
        SET
        DST$B_VS_VFLAGS         = [ 0, B_ ],    ! Value-flags (access info)
        DST$V_VS_VALKIND        = [ 0, V_(0,2) ], ! How to interpret the value
```

```
        DST$V_VS_INDIRECT       = [ 0, V_(2) ],   ! Set to get indirection
        DST$V_VS_DISP           = [ 0, V_(3) ],   ! Set for register displacement
        DST$V_VS_REGNUM         = [ 0, V_(4,4) ], ! Register number for indexing
        DST$L_VS_VALUE          = [ 1, L_ ],      ! Value, address, or bit offset
        DST$L_VS_DSC_OFFS       = [ 1, L_ ],      ! Offset in bytes to descriptor
                                                  !      from DST$A_VS_DSC_BASE
        DST$A_VS_DSC_BASE       = [ 5, A_ ],      ! Descriptor starts at this loca-
                                                  !      tion + DST$L_VS_DSC_OFFS
        DST$L_VS_TVS_OFFSET     = [ 1, L_ ],      ! Offset in bytes to Value Spec
                                                  !      from DST$A_VS_TVS_BASE
        DST$A_VS_TVS_BASE       = [ 5, A_ ],      ! Value Spec starts at this loca-
                                                  !      tion + DST$A_VS_TVS_OFFSET
        DST$W_VS_LENGTH         = [ 1, W_ ],      ! Length of Value Spec in bytes
                                                  !      not counting the VFLAGS
                                                  !      and VS_LENGTH fields
        DST$B_VS_ALLOC          = [ 3, B_ ],      ! Allocation indicator
        DST$A_VS_MATSPEC        = [ 4, A_ ]       ! Location of Materialization
                                                  !      Specification
        TES;

MACRO
        DST$VAL_SPEC = BLOCK[,BYTE] FIELD(DST$VS_HDR_FIELDS) %;

! The following literal values may appear in the DST$B_VS_ALLOC field.
!
LITERAL
        DST$K_VS_ALLOC_STAT     = 1,    ! Value is static
        DST$K_VS_ALLOC_DYN      = 2;    ! Value is dynamic

! Define the fields of the Materialization Specification.  Also define the
! declaration macro.
!
FIELD DST$MS_FIELDS =
        SET
        DST$B_MS_KIND           = [ 0, B_ ],      ! The kind of value produced
        DST$B_MS_MECH           = [ 1, B_ ],      ! The mechanism whereby produced
        DST$B_MS_FLAGBITS       = [ 2, B_ ],      ! Flag bits
        DST$V_MS_NOEVAL         = [ 2, V_(0) ],   ! Purpose of this bit not clear
        DST$V_MS_DUMARG         = [ 2, V_(1) ],   ! Include dummy argument on call
        DST$A_MS_MECH_SPEC      = [ 3, A_ ],      ! Location of Mechanism Spec
        DST$L_MS_MECH_RTNADDR   = [ 3, L_ ]       ! Routine address for call on
                                                  !      compiler-generated thunk
        TES;

MACRO
        DST$MATER_SPEC = BLOCK[,BYTE] FIELD(DST$MS_FIELDS) %;

! The following values may appear in the DST$B_MS_KIND field.
!
LITERAL
        DST$K_MS_BYTADDR        = 1,    ! The value is a byte address
        DST$K_MS_BITADDR        = 2,    ! The value is a bit address (a longword
                                        !      byte address plus a longword bit
```

```
                                        !       offset from the byte address)
        DST$K_MS_BITOFFS        = 3,    ! The value is a bit offset (normally a
                                        !       bit offset from the start of a
                                        !       record--used for record components)
        DST$K_MS_RVAL           = 4,    ! The value is a literal value (constant)
        DST$K_MS_REG            = 5,    ! The value is a register number (the
                                        !       address is a register address)
        DST$K_MS_DSC            = 6;    ! The value is a VAX standard descriptor

! The following values may appear in the DST$B_MS_MECH field.
!
LITERAL
        DST$K_MS_MECH_RTNCALL   = 1,    ! Routine call on a compiler-
                                        !       generated thunk
        DST$K_MS_MECH_STK       = 2;    ! DST Stack Machine routine
```

DESCRIPTOR VALUE SPECIFICATIONS

If the DST$B_VS_VFLAGS field has the special value DST$K_VFLAGS_DSC, this is a Descriptor Value Specification. Such a Value Specification contains an offset relative to the end of the Value Specification that points to a VAX Standard Descriptor later in the same DST record. That descriptor then contains the actual address that the Value Specification seeks to specify. This is thus the format:



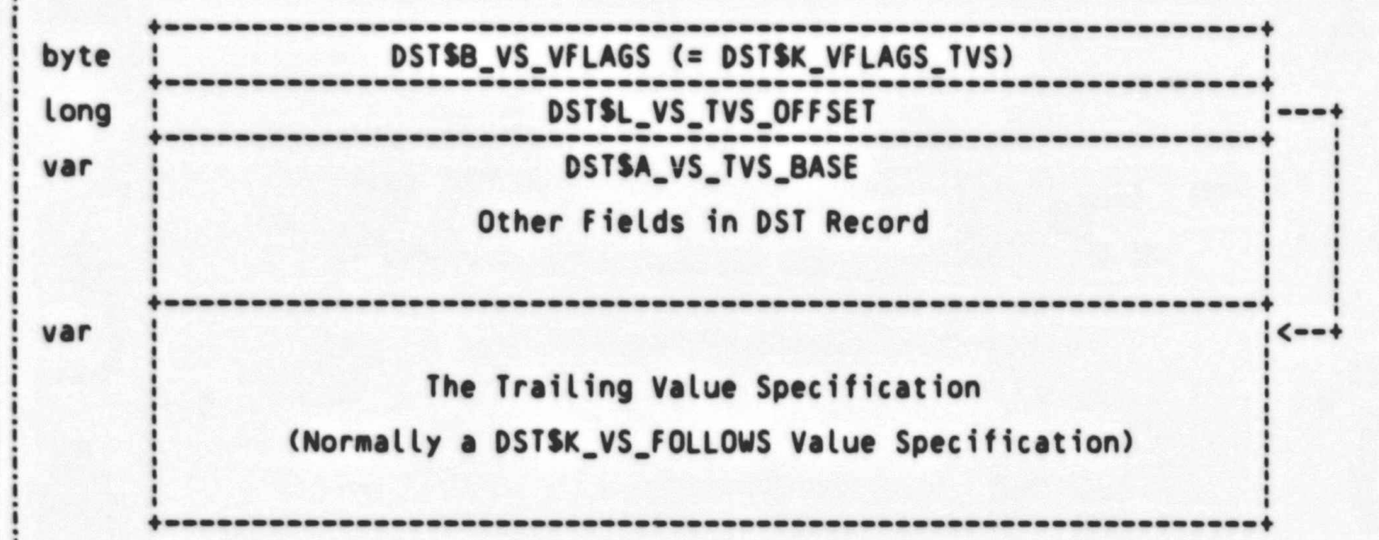
The address of the VAX Standard Descriptor is computed as follows:

```
DSC_PTR = VS_PTR[DST$A_VS_DSC_BASE] + .VS_PTR[DST$L_VS_DSC_OFFS];
```

TRAILING VALUE SPEC VALUE SPECIFICATIONS

If the DST$B_VS_VFLAGS field has the special value DST$K_VFLAGS_TVS, this is a Trailing Value Spec Value Specification. Such a Value Specification contains a pointer relative to DST$A_VS_TVS_BASE that points to another Value Specification later in the same DST record. This second Value Specification is normally of the most general and powerful form of Value Specification, namely the VS-Follows Value Specification. In effect, the Trailing Value Spec format is a five-byte Value Specification (small enough to fit in a Data DST Record) which points to a larger Value Specification elsewhere in the same DST record. This larger Value Specification can be arbitrarily large and complex in order to do whatever computation is necessary to obtain the desired value, address, or descriptor.

This is the format of the Trailing Value Spec Value Specification:



The address of the Trailing Value Specification is computed as follows:

```
TVS_PTR = VS_PTR[DST$A_VS_TVS_BASE] + .VS_PTR[DST$L_VS_TVS_OFFSET];
```

VS-FOLLOWS VALUE SPECIFICATIONS

If the DST$B_VS_VFLAGS field has the special value DST$K_VS_FOLLOWS, this is a VS-Follows Value Specification. This is the most general and powerful form of Value Specification. The specification itself can be arbitrarily long, but it can also do an arbitrarily complex computation in order to compute the desired value, address, or descriptor. This is the format of the VS-Follows Value Specification:

A VS-Follows Value Specification contains a Materialization Specification which indicates how the value is materialized. This specification indicates what kind of value is being produced, by what mechanism it is produced, and in detail how it is produced. It also contains some flag bits.

The kind of value being produced can be a 32-bit byte address, a 64-bit bit address (a byte address followed by a 32-bit bit offset), a bit offset (relative to the start of a record--used only for record components), a literal value (a constant or "R-value"), a register address, or an actual VAX Standard Descriptor. VAX Standard Descriptors are mainly produced by Value Specification within Type Specifications where a descriptor must be built to describe a data type such as an array type with run-time subscript bounds.

Values can be produced by two mechanisms. One is a routine call on a compiler-generated thunk. In this case, the compiler generates a routine in the object code which when called produces the desired value. The address of the routine is specified in the Mechanism Specification. The other mechanism is a DST Stack Machine routine. The DST Stack Machine is a virtual machine which DEBUG emulates. To use it, the compiler generates code for this virtual machine which, when executed at DEBUG-time, produces the desired value. The DST Stack Machine form of Mechanism Specification constitutes the most general and powerful form of value specification supported by DEBUG.

CALLS ON COMPILER-GENERATED THUNKS

The Routine Call Mechanism Specification specifies the address of a compiler-generated routine (a thunk) which DEBUG can call to perform the desired value computation. This form of Mechanism Specification must be used for PL/I "BASED" variables since the address of such a variable can depend on the value returned by a user-defined function. In this case, the Mechanism Specification consists of a single longword giving the address of the compiler-generated thunk to call.

This is the format of the whole Value Specification when the Routine Call Mechanism Specification is used:

byte	DST$B_VS_VFLAGS (=DST$K_VS_FOLLOWS)
word	DST$W_VS_LENGTH (= 8)
byte	DST$B_VS_ALLOC (= DST$K_VS_ALLOC_DYN)
byte	DST$B_MS_KIND
byte	DST$B_MS_MECH (= DST$K_MS_MECH_RTNCALL)
byte	DST$B_MS_FLAGBITS
long	DST$L_MS_MECH_RTNADDR

The called routine is passed the address of a vector of register values as its one argument. This vector contains all register value for the scope (call frame) in which the symbol having this Value Specification is declared. The vector contains the values of registers R0 - R11, AP, FP, SP, PC, and PSL in that order. The routine is allowed to use all such values in its computations, but is not allowed to change the contents of the register vector. In addition, the routine is passed the value of FP (the Frame Pointer) in register R1.

The value of the routine should be returned to DEBUG in register R0.

The DST$V_MS_DUMARG bit should be set in the DST$B_MS_FLAGBITS field if the called routine expects to return a value longer than one longword. If DST$V_MS_DUMARG is set, the address of an octaword (four-longword) buffer is passed as the first argument to the called routine with the expectation that the routine's value will be returned to this buffer. The address of the register vector is then the second argument.

THE DST STACK MACHINE

The DST Stack Machine is a virtual machine emulated by DEBUG. This machine can push and pop values on a stack and can perform a variety of arithmetic and logical operations. It can also call compiler-generated thunks. The DST Stack Machine is used when a value must be computed at DEBUG-time and the Standard Format Value Specification is not adequate and a compiler-generated thunk to do the whole computation seems undesirable. In such cases, the compiler can generate a Mechanism Specification which consists of code for the Stack Machine. At DEBUG-time, when the value in question is needed, DEBUG will interpret this code until the STOP instruction is encountered. The value that remains on the top of the Stack Machine stack is then taken to be the desired value.

The format of the whole Value Specification when a DST Stack Machine Mechanism Specification is used is as follows:

byte	DST$B_VS_VFLAGS (=DST$K_VS_FOLLOWS)
word	DST$W_VS_LENGTH
byte	DST$B_VS_ALLOC
byte	DST$B_MS_KIND
byte	DST$B_MS_MECH (= DST$K_MS_MECH_STK)
byte	DST$B_MS_FLAGBITS
var	DST$A_MS_MECH_SPEC DST Stack Machine Routine

Here the DST$B_VS_ALLOC field should have the value DST$K_VS_ALLOC_DYN if any kind of address is computed and DST$K_VS_ALLOC_STAT if a literal value (a constant) is computed. The need for this field is not clear since DEBUG ignores it at present.

The stack upon which the DST Stack Machine operates consists of 256 locations where each location is a longword. The stack grows toward smaller addresses and shrinks toward larger addresses; in this regard it is like the VAX call stack. A DST Stack Machine Routine consists of a sequence of Stack Machine instructions ending in a STOP instruction (DST$K_STK_STOP). When the machine stops, the top location or locations on the stack constitute the value of the routine. The length of the value is determined by the DST$B_MS_KIND field.

```
!       The DST Stack Machine supports the instructions tabulated in the re-
!       mainder of this section.  Each instruction consists of a one-byte op-
!       code followed by zero or more operand bytes, depending on the op-code.
!       In this description, the "top" stack cell refers to the most recently
!       pushed cell still on the stack and the "second" cell refers to the next
!       most recently pushed cell still on the stack.  Each cell contains a
!       longword value.
!
!
! Define the Push Register instructions.  These instructions push the indicated
! register value on the Stack Machine stack.  The register values are taken from
! the scope (call frame) of the symbol for which the value is being computed.
!
LITERAL
        DST$K_STK_LOW           = 0,    ! Lower bound for range checking
        DST$K_STK_PUSHR0        = 0,    ! Push the value of register R0
        DST$K_STK_PUSHR1        = 1,    ! Push the value of register R1
        DST$K_STK_PUSHR2        = 2,    ! Push the value of register R2
        DST$K_STK_PUSHR3        = 3,    ! Push the value of register R3
        DST$K_STK_PUSHR4        = 4,    ! Push the value of register R4
        DST$K_STK_PUSHR5        = 5,    ! Push the value of register R5
        DST$K_STK_PUSHR6        = 6,    ! Push the value of register R6
        DST$K_STK_PUSHR7        = 7,    ! Push the value of register R7
        DST$K_STK_PUSHR8        = 8,    ! Push the value of register R8
        DST$K_STK_PUSHR9        = 9,    ! Push the value of register R9
        DST$K_STK_PUSHR10       = 10,   ! Push the value of register R10
        DST$K_STK_PUSHR11       = 11,   ! Push the value of register R11
        DST$K_STK_PUSHRAP       = 12,   ! Push the value of the AP
        DST$K_STK_PUSHRFP       = 13,   ! Push the value of the FP
        DST$K_STK_PUSHRSP       = 14,   ! Push the value of the SP
        DST$K_STK_PUSHRPC       = 15;   ! Push the value of the PC

! Define the Push Immediate instructions.  These instructions are used to push
! constant values on the Stack Machine stack.  The constant value to push comes
! immediately after the instruction op-code.  For the signed and unsigned in-
! structions, the value to push is zero-extended or sign-extended to 32 bits
! as appropriate.  In the case of the Push Immediate Variable instruction, the
! byte after the op-code gives the byte length of the constant value to push.
! The constant value to push then follows immediately after that length byte.
! The constant value is zero-extended to the nearest longword boundary on the
! high-address end and the resulting block is pushed onto the stack.
!
LITERAL
        DST$K_STK_PUSHIMB       = 16,   ! Push Immediate Byte (signed)
        DST$K_STK_PUSHIMW       = 17,   ! Push Immediate Word (signed)
        DST$K_STK_PUSHIML       = 18,   ! Push Immediate Longword (signed)
        DST$K_STK_PUSHIM_VAR    = 24,   ! Push Immediate Variable
        DST$K_STK_PUSHIMBU      = 25,   ! Push Immediate Byte Unsigned
        DST$K_STK_PUSHIMWU      = 26;   ! Puch Immediate Word Unsigned

! Define the Push Indirect instructions.  For these instructions, the top stack
! cell is popped and the one, two, or four bytes at the address given by the
! popped cell are sign extended to 32 bits and pushed on the stack.  For the
```

```
! unsigned instructions, the value is instead zero-extended to 32 bits and
! pushed on the stack.
!
LITERAL
        DST$K_STK_PUSHINB       = 20,   ! Push Indirect Byte (signed)
        DST$K_STK_PUSHINW       = 21,   ! Push Indirect Word (signed)
        DST$K_STK_PUSHINL       = 22,   ! Push Indirect Longword (signed)
        DST$K_STK_PUSHINBU      = 27,   ! Push Indirect Byte Unsigned
        DST$K_STK_PUSHINWU      = 28;   ! Push Indirect Word Unsigned

! Define the arithmetic and logical instructions.  These instruction pop the
! top two cells on the stack, perform the indicated operation on these operands,
! and push the result back onto the stack.
!
LITERAL
        DST$K_STK_ADD           = 19,   ! Add--The top two cells on the stack
                                        !       are popped from the stack and
                                        !       added together.  The resulting
                                        !       sum is pushed onto the stack.
        DST$K_STK_SUB           = 29,   ! Subtract--The second cell on the stack
                                        !       is subtracted from the top cell.
                                        !       Both are popped from the stack.
                                        !       The resulting difference is then
                                        !       pushed onto the stack.
        DST$K_STK_MULT          = 30,   ! Multiply--The top two stack cells are
                                        !       popped from the stack and multi-
                                        !       plied.  The resulting product is
                                        !       then pushed onto the stack.
        DST$K_STK_DIV           = 31,   ! Divide--The top stack cell is divided
                                        !       by the second stack cell.  Both
                                        !       are popped from the stack.  Their
                                        !       quotient is then pushed onto the
                                        !       stack.
        DST$K_STK_LSH           = 32,   ! Logical Shift--Shift the second stack
                                        !       cell by the number of bits given
                                        !       by the top stack cell; pop both
                                        !       operands and push the shifted
                                        !       second cell on the stack
        DST$K_STK_ROT           = 33;   ! Rotate--Rotate the second stack cell
                                        !       by the number of bits given by
                                        !       the top stack cell; pop both
                                        !       operands and push the rotated
                                        !       second cell on the stack

! Define the Copy and Exchange instructions.  These instructions make a copy
! of the top stack cell or exchange the top two cells on the stack.
!
LITERAL
        DST$K_STK_COP           = 34,   ! Copy--A copy of the top stack cell
                                        !       is pushed onto the stack
        DST$K_STK_EXCH          = 35;   ! Exchange--The top two stack cells are
                                        !       interchanged
```

```
! Define the Store instructions.  Following the op-code, these instructions
! contain a byte which is interpreted as a signed offset into the stack.  The
! low-order byte, word, or longword of the top stack cell is stored into the
! byte, word, or longword given by the current stack pointer plus four plus
! the signed offset into the stack.  (In short, the offset is an offset from
! the second stack cell.)  After that, the top stack cell is popped.  These
! instructions permit values to be stored into stack locations other than the
! top or second stack cell.
!
LITERAL
        DST$K_STK_STO_B         = 36,   ! Store Byte into Stack
        DST$K_STK_STO_W         = 37,   ! Store Word into Stack
        DST$K_STK_STO_L         = 38;   ! Store Longword into Stack


! Define the Pop instruction.  This instruction simply pops the top stack cell,
! meaning that the top stack cell is removed from the stack and discarded.
!
LITERAL
        DST$K_STK_POP           = 39;   ! Pop Top Stack Cell


! Define the Stop instruction.  This instruction stops the DST Stack Machine and
! is required at the end of every DST Stack Machine routine.  Whatever value is
! left at the top of the stack when the Stop instruction is executed is taken to
! be the value of the Stack Machine routine.  This value may be a longword (a
! byte address, for example), two longwords (byte address and bit offset), any
! size literal value (an H-Floating literal, for instance), or a full VAX Stan-
! dard Descriptor, depending on the value of the DST$B_MS_KIND field.
!
LITERAL
        DST$K_STK_STOP          = 23;   ! Stop the Stack Machine


! Define the Routine Call instructions.  These instructions call a compiler-
! generated routine (a thunk) whose address is given by the top stack cell.
! Before the call actually occurs, the top stack cell is popped.  The value
! that is returned by the thunk is then pushed onto the stack.
!
! The Routine Call instruction works as follows.  The address of the thunk to
! to be called is taken from the top stack cell.  The top cell is then popped.
! The thunk, which is called with a CALL instruction, gets two arguments.  The
! first argument is the address of a vector of register values for the scope
! (call frame) of the symbol to which this Value Specification belongs.  This
! vector contains the values of registers R0 - R11, AP, FP, SP, PC, and PSL in
! that order; the called thunk is free to read any value it wants from this
! vector but may not store into it.  The second parameter is a pointer to the
! top of the DST Stack Machine stack after the thunk address has been popped.
! A Stack Machine routine can thus compute arguments to the thunk and push them
! on the stack before pushing the thunk address and calling the thunk.  In
! addition, the value of FP in the symbol's scope is passed to the thunk in
! register R1.  The routine's value is expected to be returned in register R0.
! This value is pushed onto the stack.
!
! The Routine Call With Alternate Return instruction works this same way except
! that the address of an octaword buffer (4 longwords) is passed to the thunk
```

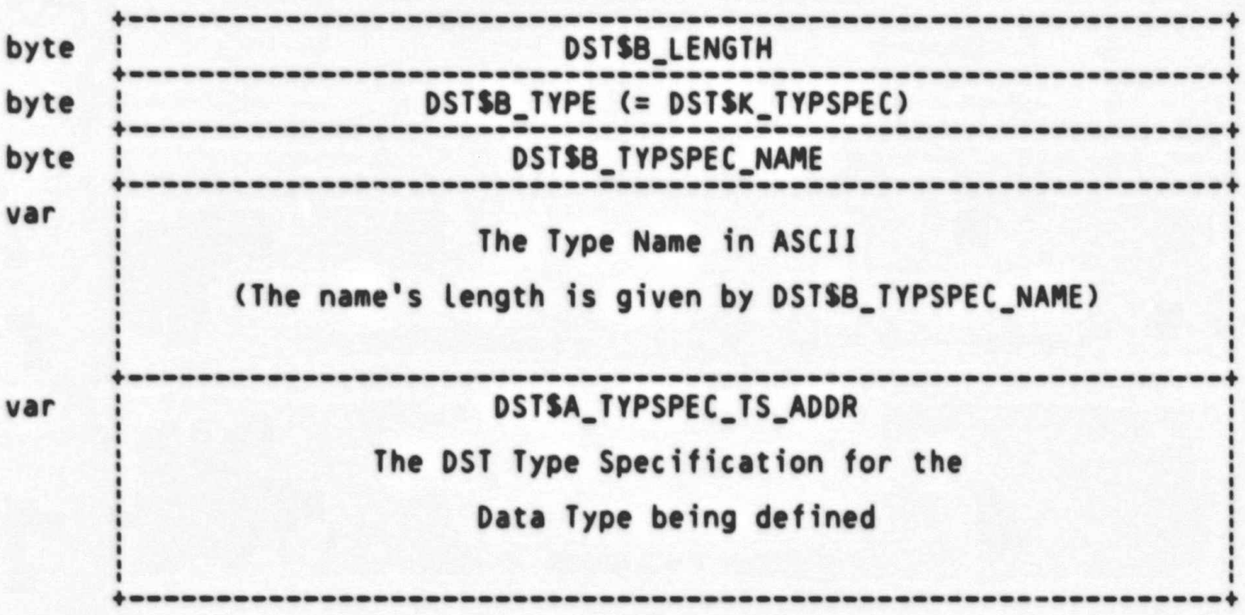
```
! as the first argument, with the register vector being the second argument and
! the stack address being the third argument.  In this case, the routine value
! is expected to be returned to the octaword buffer, not in register R0.  The
! whole octaword buffer is then pushed onto the stack.
!
LITERAL
        DST$K_STK_RTNCALL       = 40,   ! Routine Call (value returned in R0)
        DST$K_STK_RTNCALL_ALT   = 41;   ! Routine Call With Alternate Return


! Define the Push Record Address instructions.  These instructions push the
! address of the outer-most or inner-most record structure for which the cur-
! rent symbol is a record component.  They are used for constructing VAX Stan-
! dard Descriptors on the Stack Machine stack when some part of the descriptor
! depends on some other component of the same record.  In PL/I, for instance,
! the subscript bounds of an array component of a record may depend on another
! component of that record.  In such cases, the only way to get the address of
! that other component in the current record is to use one of the Push Record
! Address instructions.  The Push Outer Record Address instruction pushes the
! address of the outer-most record of which the current symbol is a component
! while the Push Inner Record Address instruction pushes the address of the
! inner-most record of which the current symbol is a component.
!
LITERAL
        DST$K_STK_PUSH_OUTER_REC = 42,  ! Push Outer Record Address
        DST$K_STK_PUSH_INNER_REC = 43;  ! Push Inner Record Address


! Define the highest op-code value accepted by the DST Stack Machine.  This
! value is used for op-code range checking.
!
LITERAL
        DST$K_STK_HIGH          = 43;   ! Upper bound for range checking

! END OF VALUE SPECIFICATION DESCRIPTION.
```

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

The Type Specification DST record gives the most general data type description available in the Debug Symbol Table. It contains the name of the data type being described and a DST Type Specification that describes the type. The type name is used in languages where data types can be named, such as PASCAL. If no type name exists, the null name (the name of zero length) is specified in this record. DST Type Specifications are described in detail in the next section of this definition file.

Type Specification DST records either immediately follow Separate Type Specification DST records or are pointed to by Indirect Type Specifications or Novel Length Type Specifications elsewhere in the DST for the current module.

This is the format of the Type Specification DST record:



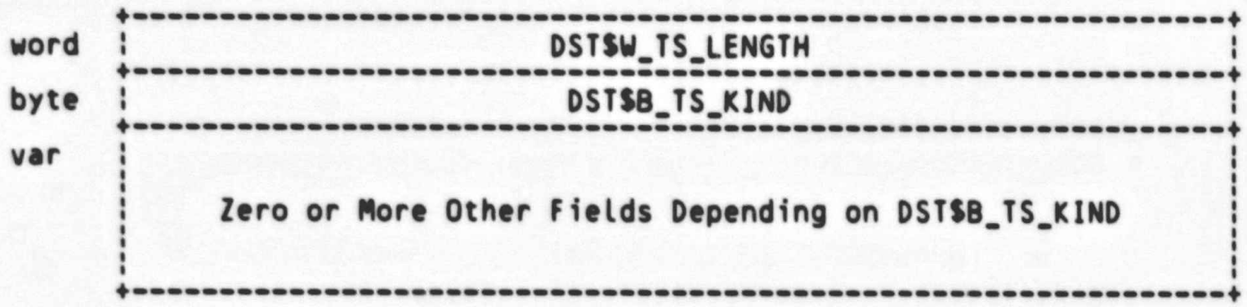
Define the fields of the Type Specification DST record.

```
FIELD DST$TYPSPEC_FIELDS =
        SET
        DST$B_TYPSPEC_NAME      = [ 2, B_ ],    ! The count byte for the Counted
                                                !       ASCII Type Name
        DST$A_TYPSPEC_TS_ADDR   = [ 3, A_ ]     ! The location of the DST Type
                                                !       Specification
        TES;
```

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

A DST Type Specification specifies the data type of some data symbol. DST Type Specifications constitute the most general form of data type description available in the Debug Symbol Table. They are found in only one kind of DST record, namely the Type Specification DST record. However, some Type Specifications contain nested Type Specifications, which permits quite complex type descriptions. For example, the parent type for a Subrange data type is given by a nested Type Specification within the Subrange Type Specification.

This is the general format of all DST Type Specifications:



A data symbol whose data type must be described by a DST Type Specification is described by a Separate Type Specification DST record. This DST record is immediately followed by a Type Specification DST record which contains the DST Type Specification for the symbol's data type.

To conserve DST space when several symbols have the same data type, the Type Specification that follows the Separate Type Specification DST record may be an Indirect Type Specification. The Indirect Type Specification then contains a DST pointer to the actual Type Specification DST record for the symbol's type. Only a single copy of this actual Type Specification is then needed. Multiple symbols of the same Record or Enumeration type must also use Separate Type Specification DST records followed by Type Specification DST records containing Indirect Type Specifications. In this case, the Indirect Type Specifications point to the Record Begin or Enumeration Type Begin DST record for the record or enumeration type being specified.

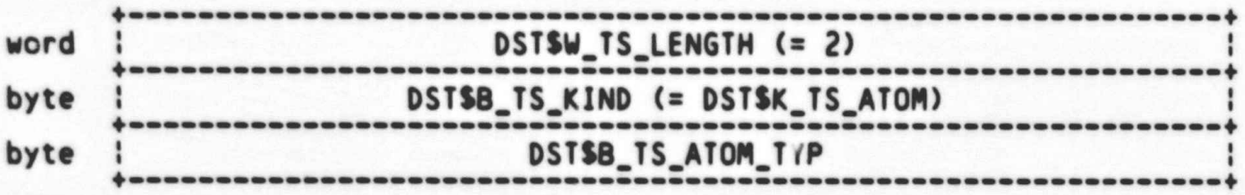
In fact, the only Type Specification that can refer to a record or enumeration type is the Indirect Type Specification. (The Novel Length Type Specification can too but is not normally used this way.) This Type Specification is thus used within other Type Specifications when record or enumeration types must be specified. For example, when the element type of an array is a record or enumeration type, it is specified by an Indirect Type Specification within the Array Type Specification. Similarly, if the target of a typed pointer is a record or enumeration type object, the target type is specified by an Indirect Type Specification

```
!       within the Typed Pointer Type Specification.
!
!
!
! Define all the fields that can appear in the various Type Specifications.
! Also define the declaration macro.
!
FIELD DST$TYPE_SPEC_FIELDS =
        SET
        DST$W_TS_LENGTH          = [ 0, W_ ],   ! The byte length of the Type
                                                !      Specification not includ-
                                                !      ing this length field
        DST$B_TS_KIND            = [ 2, B_ ],   ! The Type Specification kind
        DST$B_TS_ATOM_TYP        = [ 3, B_ ],   ! The Atomic data type code
        DST$A_TS_DSC_VSPEC_ADDR  = [ 3, A_ ],   ! The VAX descriptor Value Spec
        DST$L_TS_IND_PTR         = [ 3, L_ ],   ! Indirect Type Spec DST pointer
        DST$A_TS_TPTR_TSPEC_ADDR= [ 3, A_ ],    ! Typed Pointer parent type Type
                                                !      Specification location
        DST$B_TS_PIC_DLENG       = [ 3, B_ ],   ! The byte length of data objects
                                                !      of this picture type
        DST$B_TS_PIC_LANG        = [ 4, B_ ],   ! The DST language code for this
                                                !      picture data type
        DST$B_TS_PIC_PLENG       = [ 5, B_ ],   ! The length of the picture
                                                !      string in this Type Spec
        DST$A_TS_PIC_ADDR        = [ 6, A_ ],   ! The location where the picture
                                                !      is encoded in Type Spec
        DST$B_TS_ARRAY_DIM       = [ 3, B_ ],   ! The number of array dimensions
        DST$A_TS_ARRAY_FLAGS_ADDR=[ 4, A_ ],    ! The location of the array flags
                                                !      that indicate Type Specs
                                                !      for the subscript types
        DST$L_TS_SET_LENG        = [ 3, L_ ],   ! The length in bits of data
                                                !      objects of this Set type
        DST$A_TS_SET_PAR_TSPEC_ADDR = [ 7, A_ ],! The location of the Set's
                                                !      parent type Type Spec
        DST$L_TS_SUBR_LENG       = [ 3, L_ ],   ! The length in bits of objects
                                                !      of this subrange type
        DST$A_TS_SUBR_PAR_TSPEC_ADDR= [ 7, A_ ],! Location of the parent type
                                                !      Type Specification within
                                                !      the Subrange Type Spec
        DST$B_TS_FILE_LANG       = [ 3, B_ ],   ! Language code for file type
        DST$A_TS_FILE_RCRD_TYP   = [ 4, A_ ],   ! Location of Type Spec giving
                                                !      element type for file
        DST$A_TS_AREA_BYTE_LEN   = [ 3, A_ ],   ! Length in bytes of PL/I "area"
        DST$A_TS_OFFSET_VALSPEC  = [ 3, A_ ],   ! Location of Value Spec giving
                                                !      base address of PL/I area
        DST$L_TS_NOV_LENG        = [ 3, L_ ],   ! The "novel" length in bits of
                                                !      objects of this data type
        DST$L_TS_NOV_LENG_PAR_TSPEC = [ 7, L_ ],! DST pointer to parent type for
                                                !      this "novel length" type
        DST$L_TS_SELF_LENG       = [ 3, L_ ]    ! Table length for this array of
                                                !      PL/I Self-Relative Labels
        TES;

MACRO
        DST$TYPE_SPEC = BLOCK[,BYTE] FIELD(DST$TYPE_SPEC_FIELDS) %;
```

```
! The following are the values that may appear in the DST$B_TS_KIND field.
!
LITERAL
        DST$K_TS_DTYPE_LOWEST   = 1,    ! ---Lowest Type Spec kind
        DST$K_TS_ATOM           = 1,    ! Atomic Type Spec
        DST$K_TS_DSC            = 2,    ! VAX Standard Desciptor Type Spec
        DST$K_TS_IND            = 3,    ! Indirect Type Spec
        DST$K_TS_TPTR           = 4,    ! Typed Pointer Type Spec
        DST$K_TS_PTR            = 5,    ! Pointer Type Spec
        DST$K_TS_PIC            = 6,    ! Pictured Type Spec
        DST$K_TS_ARRAY          = 7,    ! Array Type Spec
        DST$K_TS_SET            = 8,    ! Set Type Spec
        DST$K_TS_SUBRANGE       = 9,    ! Subrange Type Spec
        !                       = 10,   ! Unused--available for future use
        DST$K_TS_FILE           = 11,   ! File Type Spec
        DST$K_TS_AREA           = 12,   ! Area Type Spec (PL/I)
        DST$K_TS_OFFSET         = 13,   ! Offset Type Spec (PL/I)
        DST$K_TS_NOV_LENG       = 14,   ! Novel Length Type Spec
        DST$K_TS_IND_TSPEC      = 15,   ! DEBUG internally generated pointer to
                                        !       Type Spec (cannot appear in DST)
        DST$K_TS_SELF_REL_LABEL = 16,   ! Self-Relative Label Type Spec (PL/I)
        DST$K_TS_RFA            = 17,   ! Record File Address Type Spec (BASIC)
        DST$K_TS_TASK           = 18,   ! Task Type Spec (ADA)
        DST$K_TS_DTYPE_HIGHEST  = 18;   ! ---Highest Type Spec kind

! The following set of literals give the lengths in bytes of those Type
! Specifications which have a fixed length.
!
LITERAL
        DST$K_TS_ATOM_LENG      = 4,    ! Atomic Type Spec length
        DST$K_TS_IND_LENG       = 7,    ! Indirect Type Spec length
        DST$K_TS_PTR_LENG       = 3,    ! Pointer Type Spec length
        DST$K_TS_FILE_LENG      = 4,    ! File Type Spec length
        DST$K_TS_AREA_LENG      = 3,    ! Area Type Spec length
        DST$K_TS_OFFSET_LENG    = 7,    ! Offset Type Spec length
        DST$K_TS_NOV_LENG_LENG  = 11,   ! Novel Length Type Spec length
        DST$K_TS_TASK_LENG      = 3;    ! Task Type Spec length
```

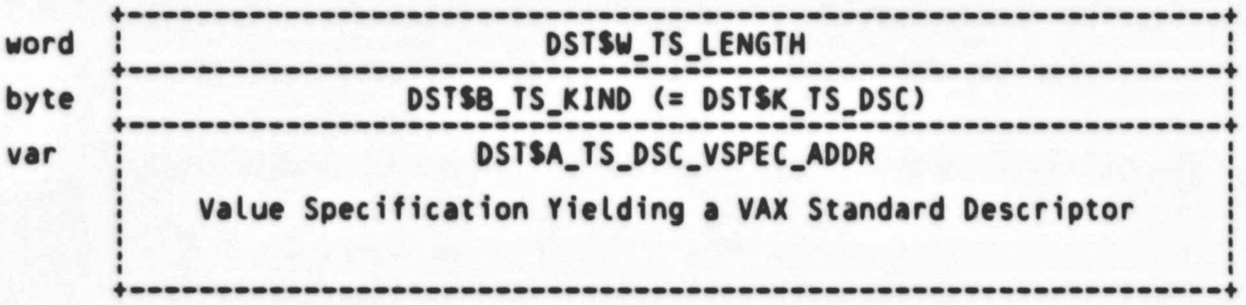
ATOMIC TYPE SPECIFICATIONS

The Atomic Type Specification is used to describe an atomic VAX standard data type. This Type Specification consists of the standard Type Specification header followed by a single byte containing the VAX standard data type code (one of the DSC$K_DTYPE_x codes). The Atomic Type Specification has the following format:



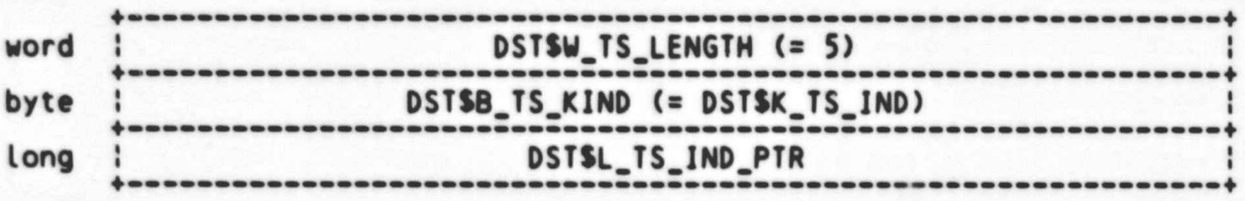
DESCRIPTOR TYPE SPECIFICATIONS

The Descriptor Type Specification is used for VAX Standard Data Types that can be described by VAX Standard Descriptors but cannot be described by an atomic type code. Packed decimal, which requires a digit length and a scale factor, and ASCII text, which requires a string length, are examples of such data types. The Descriptor Type Specification contains a Value Specification which must produce a VAX Standard Descriptor. This is the format:



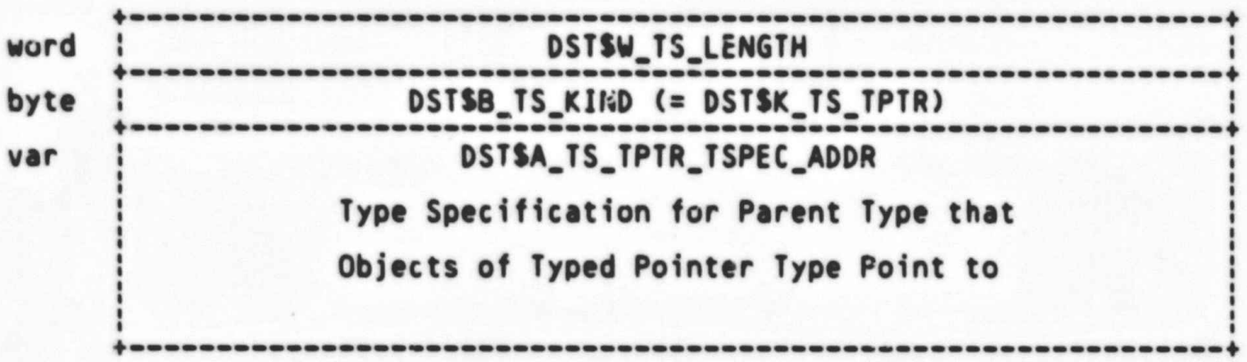
INDIRECT TYPE SPECIFICATIONS

The Indirect Type Specification is used when the actual Type Specification desired is found in another DST record. This Type Specification contains a DST pointer which points to that other DST record. The DST pointer contains the byte offset relative to the start of the whole DST of the DST record that gives the actual type information. The pointed-to DST record must be one of three kinds of DST records: a Type Specification DST record, a Record Begin DST record, or an Enumeration Type Begin DST record. The Indirect Type Specification is the only Type Specification that can refer to a record or enumeration type; those types are too complex (potentially) to be referred to any other way. This is the format of the Indirect Type Specification:



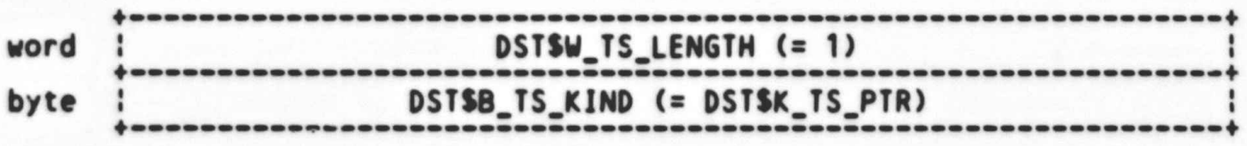
TYPED POINTER TYPE SPECIFICATIONS

The Typed Pointer Type Specification describes a typed pointer data type, meaning a pointer to a specific other data type. Pointer-to-integer, as found in PASCAL and other languages, is an example of a typed pointer type. In this example, integer is the "parent type". This Type Specification contains an embedded Type Specification which specifies the parent type for the typed pointer type. This is the format of the Type Pointer Type Specification:



POINTER TYPE SPECIFICATIONS

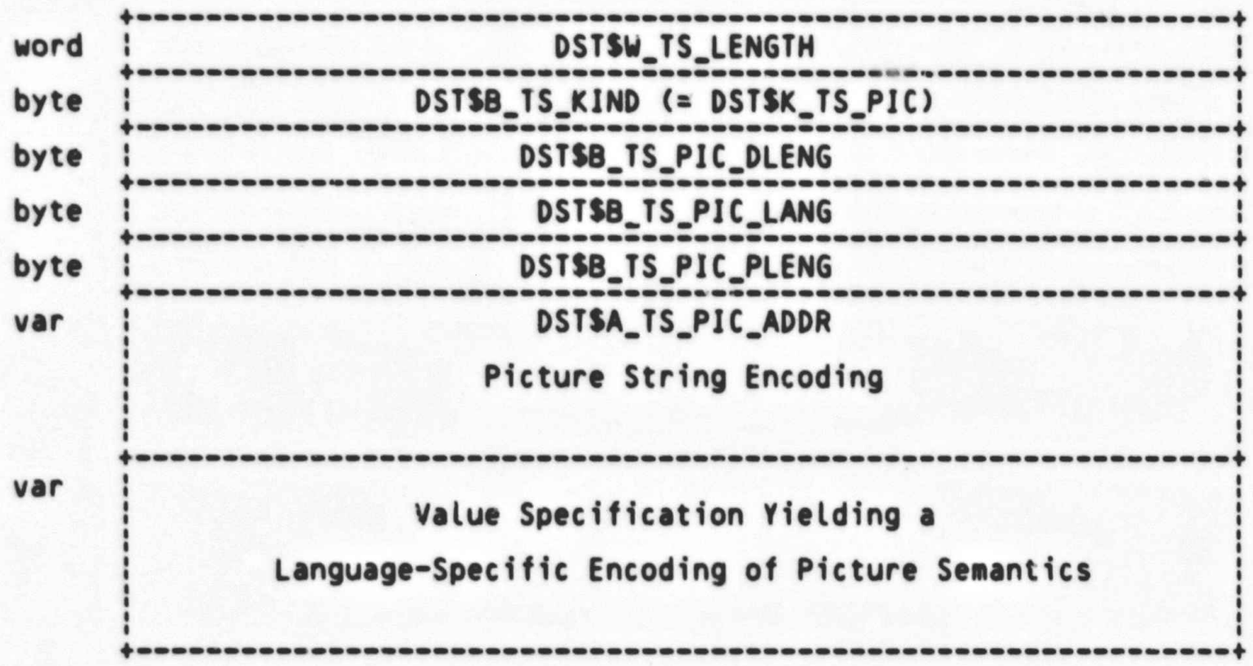
The Pointer Type Specification is used for pointer types which are not typed, meaning that the type of object that the pointer points to is not known at compile-time. PL/I pointers are examples of this kind of pointer type. Since there is no known parent type, none is specified in this Type Specification. The Pointer Type Specification thus has the simplest possible format:



PICTURE TYPE SPECIFICATIONS

The Picture Type Specification is used for picture data types as found in COBOL and PL/I. Because the exact semantics of picture data types vary between languages, this Type Specification contains the language code associated with this specific picture type. It also contains the byte length of objects of the picture type, an encoding of the picture, and a language-specific picture encoding (usually the EDITPC pattern string). The actual data objects of the picture data type are assumed to be represented as ASCII character strings.

This is the format of the Picture Type Specification:



The DST$B_TS_PIC_DLENG field contains the length in bytes of each data object of this picture type. DEBUG assumes that picture objects are represented internally as ASCII character strings.

The language code in the DST$B_TS_PIC_LANG field is the same as that used in the Module Begin DST record.

The DST$B_TS_PIC_PLENG field gives the byte length of the picture encoding in the DST$A_TS_PIC_ADDR field. The picture encoding in the DST$A_TS_PIC_ADDR field consists of a sequence of words. The high-order byte of each word contains an unsigned repetition factor and the low-order byte contains the ASCII representation of the repeated picture character. Hence the picture S999.99 is represented by this

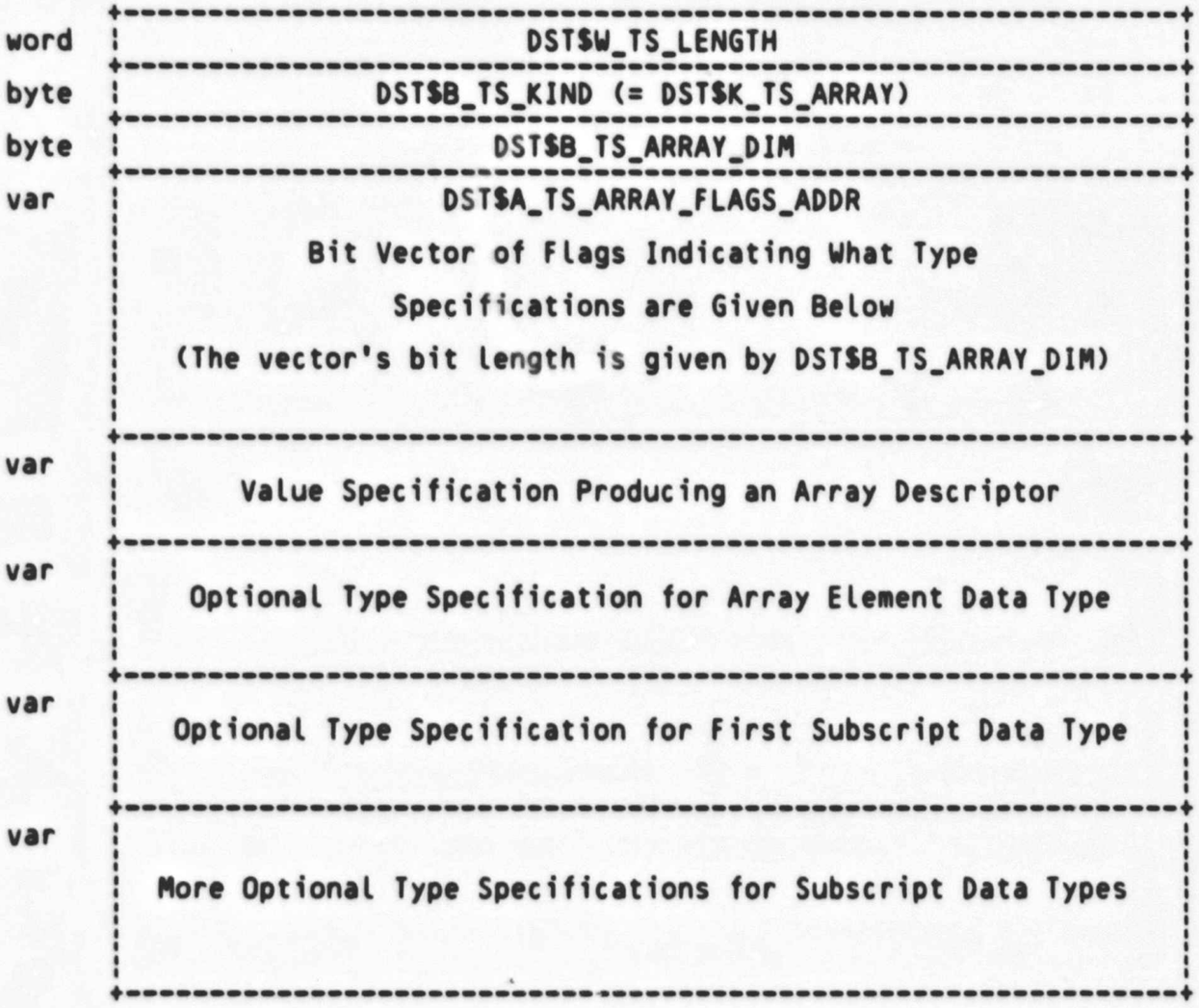
sequence of byte values: "S", 1, "9", 3, ".", 1, "9", 2. (The same picture can be written as "S(3)9.(2)9".)

The optional Value Specification at the end of the Picture Type Specification yields the address of the EDITPC pattern string that performs the encoding associated with this picture type. DEBUG uses this pattern string with the EDITPC instruction when doing DEPOSITS into objects of this picture type. If the Value Specification is omitted, DEBUG can only deposit character strings into such objects since it does not know how to encode numeric values.

ARRAY TYPE SPECIFICATIONS

The Array Type Specification specifies an Array data type. This specification can be quite complex because it not only specifies the shape of each array of this type, but also specifies the corresponding element data type and all subscript data types. The element type and the types of the subscripts are given by additional Type Specifications nested within the Array Type Specification.

This is the format of the Array Type Specification:



Here the DST$B_TS_ARRAY_DIM field gives the number of dimensions of this

array type. Next, DST$A_TS_ARRAY_FLAGS_ADDR gives the location of a bit-vector which indicates what nested Type Specifications are found later in this Array Type Specification. If bit 0 is set, a nested Type Specification is included for the array element type (the cell type). After that, if bit n is set, a nested Type Specification for the n-th subscript type is included in this Array Type Specification. If a bit in the bit-vector is zero (not set), the corresponding Type Specification is omitted from the Array Type Specification. If the element type specification is omitted, the element type is assumed to be given by the array descriptor's DTYPE field. If a subscript type specification is omitted, the subscript type is assumed to be longword integer (DTYPE_L). (Subscript Type Specifications are mainly needed for enumeration type subscripts as allowed in PASCAL.)

The number of bits in the bit-vector is DST$B_TS_ARRAY_DIM plus one more for the element type. The whole DST$A_TS_ARRAY_FLAGS_ADDR field is of course rounded up to the nearest byte boundary.

The array descriptor Value Specification that follows the bit-vector field produces a VAX Standard Descriptor for the array. (The descriptor class must be DSCK_CLASS_A, DSCK_CLASS_NCA, or DSC$K_CLASS_UBA.) This array descriptor gives the strides (or multipliers) and the lower and upper bounds for all of the array dimensions. It also gives the element data type, including its scale factor, digit count, or other type information as appropriate. However, the descriptor's element type can be overridden by an element Type Specification as noted above; in this case the DSC$B_DTYPE field of the descriptor should be zero.

The Array Type Specification is normally only used in two situations. First, it is used if the array type does not have a compile-time-constant descriptor (for example, if it has variable array bounds) and no run-time descriptor exists in the user's address space. Second, it is used if the array type cannot be described a VAX Standard Descriptor, either because the element type cannot be described by a VAX Standard Descriptor or because the subscript types are not integers. (Element types such as records, enumeration types, and typed pointers cannot be described by VAX Standard Descriptors.) If neither of these situations pertains, there are simpler ways of describing array types in the DST using Standard Data or Descriptor Format DST records.

SET TYPE SPECIFICATIONS

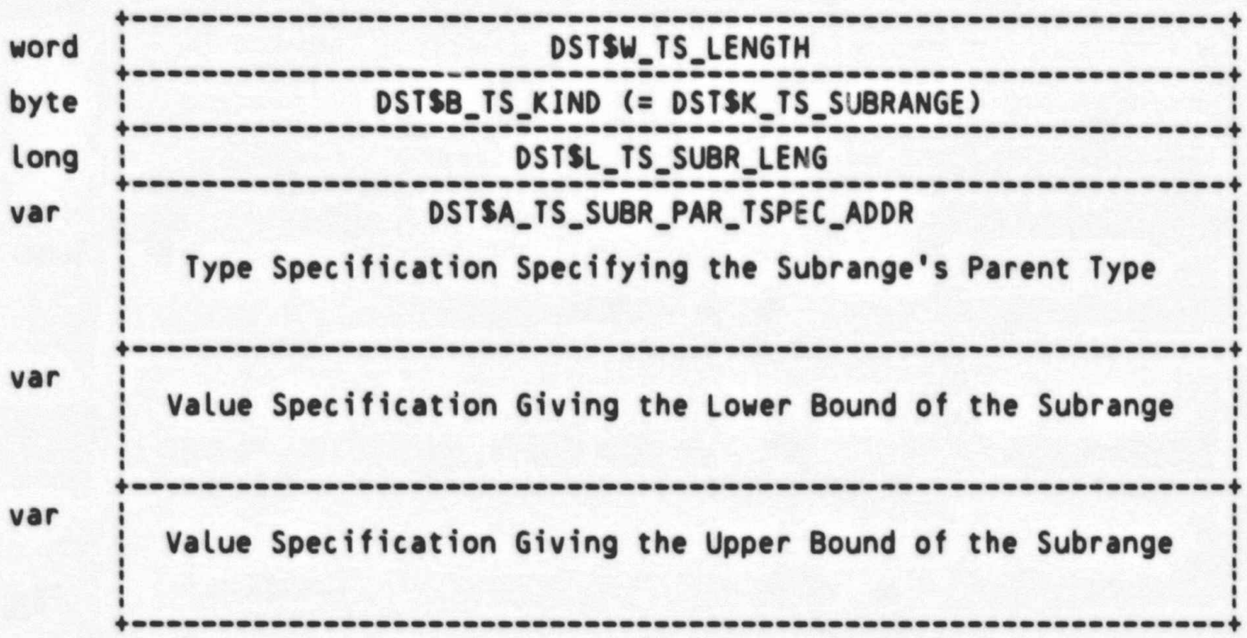
The Set Type Specification specifies a Set data type as in PASCAL. A Set type always has a parent data type. For the set-of-integers type, for example, integer is the parent type. The parent type must be either integer, some enumeration type, or a subrange of those types. DEBUG assumes that the Set type is represented internally as a bit-string where a given bit is set if and only if the corresponding integer or enumeration type element is a member of the set. The n-th bit of the bit-string (starting at bit 0) is assumed to correspond to the n-th element of the parent type. The length of the bit-string is part of the Set type and is specified in the Set Type Specification.

This is the format of the Set Type Specification:

Here the DST$L_TS_SET_LENG field gives the bit length of an object of the Set data type. DST$A_TS_SET_PAR_TSPEC_ADDR marks the location of an embedded DST Type Specification for the parent type of the Set type. Typically this is an Atomic Type Specification for type integer, an Indirect Type Specification that points to an Enumeration Type Begin DST record, or a Subrange Type Specification.

SUBRANGE TYPE SPECIFICATIONS

The Subrange Type Specification describes a Subrange data type, meaning a subrange of some ordinal type such as integer or an enumeration type. This Type Specification specifies the parent type (the original ordinal type) and the lower and upper bounds of the subrange. It also gives the bit length of objects of the Subrange type. This is the format of the Subrange Type Specification:



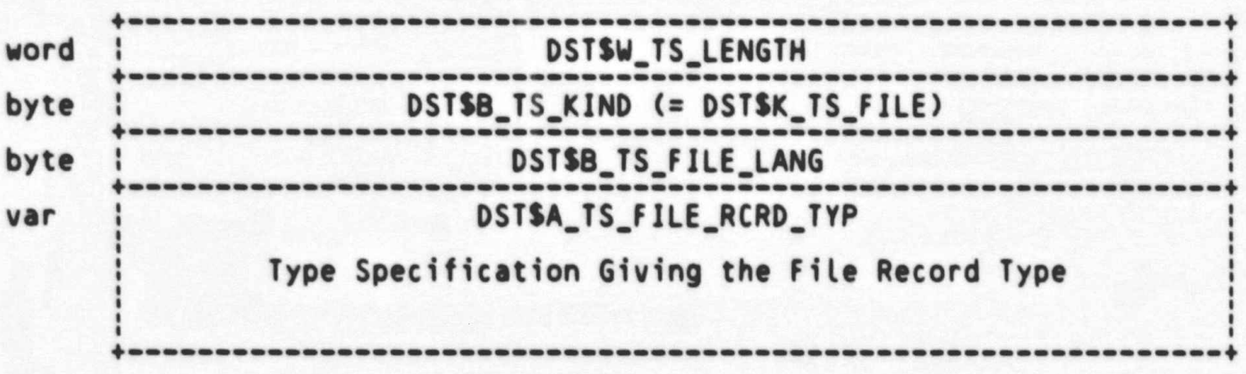
Here the DST$L_TS_SUBR_LENG field gives the length in bits of objects of the Subrange data type. DST$A_TS_SUBR_PAR_TSPEC_ADDR marks the location of a DST Type Specification for the parent type of the subrange. Typically this is an Atomic Type Specification for type integer or an Indirect Type Specification pointing to an Enumeration Type Begin DST record.

The two Value Specifications in this Type Specification specify the lower and upper bounds of the subrange. These bounds values must be values of the parent type.

FILE TYPE SPECIFICATIONS

The File Type Specification specifies a File data type as found in PASCAL or PL/I, for example. Since the interpretation of File types varies from language to language, the language code for this File type is included in the Type Specification. Optionally, a file record Type Specification can be included specifying the type of a record in this file type. A PASCAL File-of-Reals, for instance, would have Real (F-Floating) as its file record type.

This is the format of the File Type Specification:

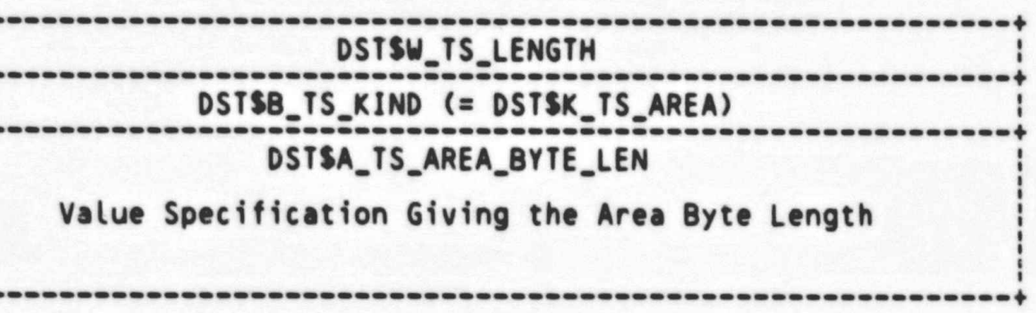


Here the DST$B_TS_FILE_LANG field contains the language code for this file. The same language codes are used as in the Module Begin DST record. DST$A_TS_FILE_RCRD_TYP is the location of a DST Type Specification for the record type, if applicable. This Type Specification is optional; if omitted, file-of-characters is assumed.

AREA TYPE SPECIFICATIONS

NOTE: THIS TYPE SPECIFICATION IS NOT SUPPORTED BY DEBUG V4.0.

The Area Type Specification describes a PL/I "area" type. PL/I areas are regions of memory whose base addresses are determined at run-time. Areas are always used in conjunction with PL/I Offsets (see below). This is the format of the Area Type Specification:

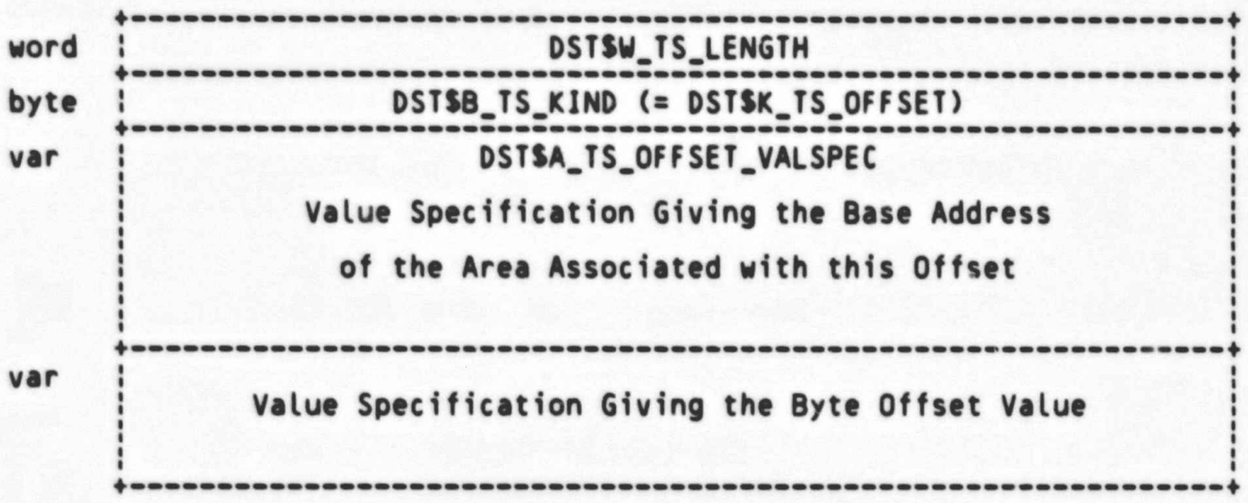


Here the DST$A_TS_AREA_BYTE_LEN Value Specification specifies the byte length of the PL/I Area.

OFFSET TYPE SPECIFICATIONS

NOTE: THIS TYPE SPECIFICATION IS NOT SUPPORTED BY DEBUG V4.0.

The Offset Type Specification describes a PL/I "offset" type. PL/I offsets are offsets relative to the start of a PL/I "area" (see above), a dynamically allocated region of memory. The Offset Type Specification specifies the base address of the associated area and the byte offset value of this offset type. This is the format:

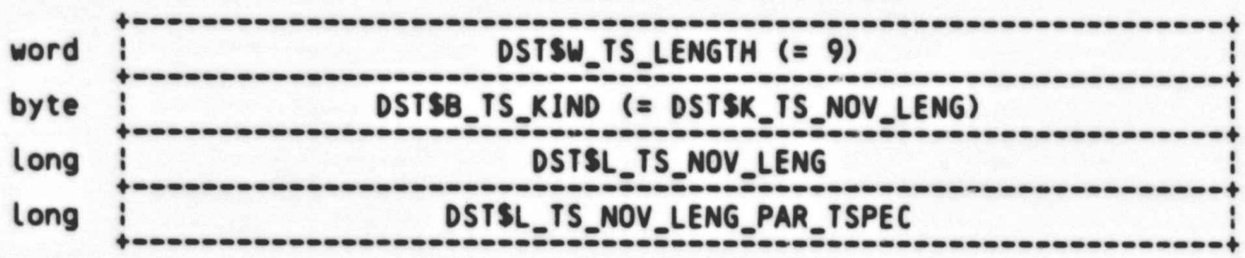


Here the DST$A_TS_OFFSET_VALSPEC Value Specification produces the base address of the associated area and the second Value Specification gives the byte offset value into the area.

NOVEL LENGTH TYPE SPECIFICATIONS

The Novel Length Type Specification is used to specify any data type that is identical to a parent data type except that the objects of this new type have a different length (a "novel" or atypical length). This Type Specification is used for the components of PACKED records in PASCAL, for example. A boolean component of a packed record consists of a single bit (the novel length) while all other booleans consist of a byte (the normal length). To describe the packed boolean type, a Novel Length Type Specification is used which specifies the novel length and points to the DST description of the parent type, namely the normal boolean type. DEBUG accessed objects of a Novel-Length type by expanding them to the normal length for that type.

This is the format of the Novel Length Type Specification:



Here the DST$L_TS_NOV_LENG field contains the "novel" length of this data type. The DST$L_TS_NOV_LENG_PAR_TSPEC field is a DST pointer which contains the byte offset relative to the start of the whole DST of the DST record that specifies the parent type. The pointed-to DST record must be a Type Specification DST record, a Record Begin DST record, or an Enumeration Type Begin DST record. (Typically it is a Type Specification DST record containing an Atomic Type Specification for type integer or boolean or an Enumeration Type Begin DST record.)

SELF-RELATIVE LABEL TYPE SPECIFICATIONS

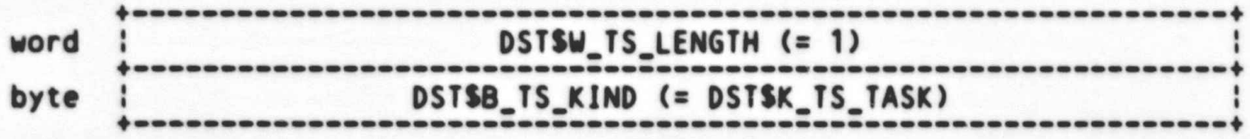
The Self-Relative Label Type Specification specifies the type of a PL/I "self-relative" label. Such a label is actually a label array, meaning that it must be indexed by an integer value to yield a specific label value. The internal representation consists of an array of longwords where each array element contains a label value relative to the start of the array. Making the element values relative to the start of the array ensures that the label array is Position-Independent (PIC).

This is the format of the Self-Relative Label Type Specification:

TASK TYPE SPECIFICATIONS

NOTE: THIS TYPE SPECIFICATION IS NOT SUPPORTED BY DEBUG V4.0.

The Task Type Specification specifies the data type of task objects as found in ADA. Objects of the Task data type are assumed to have longword values understood by the ADA multi-tasking kernel. Since no additional information is associated with the Task data type, the Task Type Specification has the minimal format:



END OF TYPE SPECIFICATION DESCRIPTION.

ENUMERATION TYPE DST RECORDS

Enumeration types, as found in PASCAL and C, are represented in the DST by three kinds of DST records. The Enumeration Type Begin DST record describes the type itself, giving the bit length of objects of that type and the name of the type (e.g., COLOR). This record is followed by some number of Enumeration Type Element DST records, one for each element, or literal, in the type (e.g., RED, BLUE, and GREEN). Each Enumeration Type Element DST record gives the name and numeric value of one literal of the enumeration type. The whole type description is then terminated by an Enumeration Type End DST record.

The Enumeration Type Begin and Enumeration Type End DST records thus bracket the list of elements of the type, much like other Begin-End pairs in the DST. The Enumeration Type Element DST records within those brackets do not have to be in numeric order of their values, although it is desirable if they are. For languages like ADA, where the numeric values of the elements need not go up sequentially with the logical element positions, the Enumeration Type DST Elements do have to be order of their logical positions, however. No other kinds of DST records (except Continuation DST records) may appear between the Enumeration Type Begin and the Enumeration Type End DST records.

THE ENUMERATION TYPE BEGIN DST RECORD

The Enumeration Type Begin DST record specifies the name of an enumeration type and the bit length of objects of that type. It also serves as the opening bracket for a list of Enumeration Type Element DST records, and must be matched by a closing Enumeration Type End DST record. This is record's format:

byte	DST$B_LENGTH
byte	DST$B_TYPE (= DST$K_ENUMBEG)
byte	DST$B_ENUMBEG_LENG
byte	DST$B_ENUMBEG_NAME
var	The Name of the Enumeration Type in ASCII (The name's length is given by DST$B_ENUMBEG_NAME)

Define the fields of the Enumeration Type Begin DST record.

```
FIELD DST$ENUMBEG_FIELDS =
        SET
        DST$B_ENUMBEG_LENG      = [ 2, B_ ],    ! Bit length of data objects of
                                                !       this enumeration type
        DST$B_ENUMBEG_NAME      = [ 3, B_ ]     ! Count byte for the Counted
                                                !       ASCII Type Name
        TES;
```

THE ENUMERATION TYPE ELEMENT DST RECORD

The Enumeration Type Element DST record specifies the name and value of one element (one literal) of an enumeration type. It may only appear between an Enumeration Type Begin and an Enumeration Type End DST record. The underlying representation of enumeration types is assumed to be unsigned integer. The DST$B_VFLAGS field in this record has its normal interpretation (see the Standard Data DST record for the details). Hence the DST$V_VALKIND field will have the value DST$K_VALKIND_LITERAL and the DST$L_VALUE field will have the appropriate integer value in this case.

This is the format of the Enumeration Type Element DST record:

byte	DST$B_LENGTH
byte	DST$B_TYPE (= DST$K_ENUMELT)
byte	DST$B_VFLAGS
long	DST$L_VALUE
byte	DST$B_NAME
var	The Name of the Enumeration Literal in ASCII (The name's length is given by DST$B_NAME)

THE ENUMERATION TYPE END DST RECORD

The Enumeration Type End DST record terminates the description of an enumeration type. This is the record's format:

byte	DST$B_LENGTH
byte	DST$B_TYPE (= DST$K_ENUMEND)

RECORD STRUCTURE DST RECORDS

Record structures, or simply records, refer to the aggregates of non-homogeneous components found in many languages. In some languages, such constructs are called "records" (in PASCAL and COBOL, for example) and in others they are called "structures" (in PL/I, for example). Here we will call them "records". What all records have in common is that they consist of a set of named components, each corresponding to some field in the record structure. The components can in general be of any data types supported by the language.

In the Debug Symbol Table, a record is represented by a Record Begin DST record followed by some number of data object DST records, one for each record component, followed by a Record End DST record. Any data object DST record within a Record-Begin/Record-End pair is taken to denote a component of that enclosing record specification. Other DST records may also appear between the Record-Begin/Record-End pair, such as Type Specification and other DST records that specify the data types of the components. However, only data DST records denote record components.

Nested records are defined by record components which are themselves records. The type of a record component which is itself a record is defined by another Record-Begin/Record-End pair of DST records. This additional record definition may appear inside the original record definition, but does not have to do so--an Indirect Type Specification pointing to a record definition outside the original record definition is also legal. Conversely, a record definition inside another record definition does not define a nested record unless some component of the outer record actually references the inner record definition. In short, the DST can only describe one level of record components at a time, but any component can be of any arbitrary data type including another record type.

The Record Begin DST record is unusual in that it can define both a data type and a data object. If the DST$B_VFLAGS field has the special value DST$K_VFLAGS_NOVAL, then the Record Begin DST record defines an abstract data type. Any object of this data type must be represented by a Separate Type Specification DST record which immediately precedes either the Record Begin DST record or a Type Specification DST record that contains an Indirect Type Specification that points to the Record Begin DST record. In this case, the name in the Record Begin record is taken to be the name of the data type, not of any object of that type.

However, if the DST$B_VFLAGS field does not contain DST$K_VFLAGS_NOVAL, then the Record Begin DST record defines both a data type and a data object of that type. This form can be used for languages such as COBOL which do not have named data types. In this case, the DST$B_VFLAGS and DST$L_VALUE fields specify the address of the record object in the same way as in the Standard Data DST record. It is still legal to have Indirect Type Specifications pointing to this Record Begin DST record, using it strictly as a type definition.

Some languages, such as PASCAL, allow record variants. (In ADA, the

same concept is called "discriminated" records.) An object of a record type with variants contains some set of components found in all objects of that type plus some set of components that vary from one record variant to the next. Which of the varying components are actually present in a given record may be determined by the value of a "tag variable" which is a fixed component of the record. Variants may also be nested so that variants have variants.

In the DST, record variants are described by Variant Set Begin DST records, Variant Value DST records, and Variant Set End DST records. The Variant Set Begin DST record marks the beginning of a set of record variants, where each variant consists of some set of record components. The Variant Set Begin DST record indicates which record component constitutes the tag variable that discriminates between the variants in the set. This tag variable must be a component of the same record and must precede the Variant Begin DST record in the DST. The Variant Begin DST record also gives the bit size of the variant, if known at compile-time.

The Variant Value DST record marks the beginning of a single record variant. It also specifies all tag variable values or value ranges that indicate the presence of this variant in a given record object. All record components (indicated by data DST records) after this Variant Value DST record and before the next Variant Value or Variant Set End DST record are taken to be components in this variant.

The Variant Set End DST record marks the end of some set of variants within a record specification. It also terminates the last variant within the set.

A record type with variants is thus specified as follows. First a Record Begin DST record marks the beginning of the record specification. After that come data DST records that denote all fixed components of the record type. Then comes a Variant Set Begin DST record that marks the beginning of a set of variant definitions and identifies the tag variable (if any) for that variant set. Immediately thereafter comes the first Variant Value DST record. It marks the start of the first variant and identifies the values or value ranges of the tag variable that correspond to this specific variant.

After the first Variant Value DST record come the data DST records for the record components in this particular variant. Next comes the Variant Value DST record for the next variant, along with its component DST records, and so on for each variant in the variant set. After the last component DST record for the last variant in the set comes a Variant Set End DST record. It is followed by the DST records for any additional record components, possibly including additional variant set definitions. Then comes the the Record End DST record.

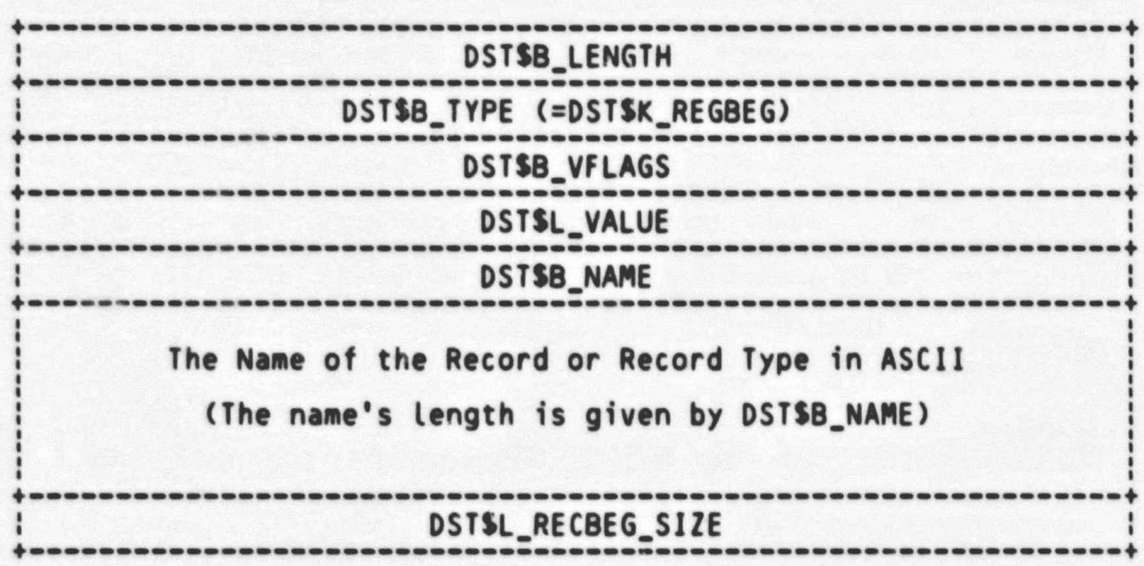
Variant sets may be nested inside variant sets. Such nesting is indicated in the DST by the corresponding proper nesting of Variant Set Begin and Variant Set End DST records.

THE RECORD BEGIN DST RECORD

The Record Begin DST record marks the beginning of a record type definition in the DST. It must be followed by the DST records for the components of that record and by a matching Record End DST record. The Record Begin DST record has essentially the same format as the Standard Data DST record, but with two exceptions. First, an extra longword gives the bit length of the record type and second, the DBG$B_VFLAGS field may have the special value DST$K_VFLAGS_NOVAL to indicate that this is strictly a type definition, not also the definition of a record object. If a normal value specification is used, a record object is being declared as well as a record type. In this case, a Trailing Value Specification may be included at the end of the DST record if necessary to describe the record's address.

The bit size of objects of this record type is also given in the DST record. This size should be included if the size is known at compile-time. If it is not known at compile-time, it should be specified as zero.

This is the format of the Record Begin DST record:




```
!
! Define the fields of the Record Begin DST record.  Also declare the macro
! that defines the trailer part of the DST record.
!
FIELD DST$RECBEG_TRAILER_FIELDS =
        SET
        DST$L_RECBEG_SIZE = [ 0 , L_ ]  ! The bit size of data objects of this
                                        !       record type (or 0 if unknown)
        TES;
```

```
MACRO
        DST$RECBEG_TRLR = BLOCK[,BYTE] FIELD(DST$RECBEG_TRAILER_FIELDS) %;
```

THE RECORD END DST RECORD

The Record End DST record marks the end of a record type definition in the DST. In effect, it terminates the scope set up by the matching Record Begin DST record. This is the record's format:

THE VARIANT SET BEGIN DST RECORD

The Variant Set Begin DST record marks the beginning of the DST description of a set of record variants. This DST record also identifies the tag variable that discriminates between the variants in the variant set. The tag variable is identified by a pointer to the DST record for the tag variable. This DST pointer consists of a byte address relative to the start of the DST. The size in bits of this variant set, meaning the size of the largest variant in the set, is also included. If this size is not known at compile-time, it should be set to zero.

This is the format of the Variant Set Begin DST record:

byte	DST$B_LENGTH
byte	DST$B_TYPE (= DST$K_VARBEG)
byte	DST$B_VFLAGS
long	DST$L_VALUE
byte	DST$B_NAME
var	The Name of the Variant Set in ASCII (The name's length is given by DST$B_NAME) (This name is normally null)
long	DST$L_VARBEG_SIZE
long	DST$L_VARBEG_TAG_PTR

```
! Define the fields of the Variant Set Begin DST record.  Also define the
! declaration macro for the trailer part of the record.

FIELD DST$VARBEG_TRAILER_FIELDS =
        SET
        DST$L_VARBEG_SIZE         = [ 0, L_ ],  ! Size in bits of variant part
                                                !      of record (or zero)
        DST$L_VARBEG_TAG_PTR      = [ 4, L_ ]   ! Pointer to TAG field DST
                                                !      record relative to the
                                                !      start of the DST
        TES;

MACRO
```

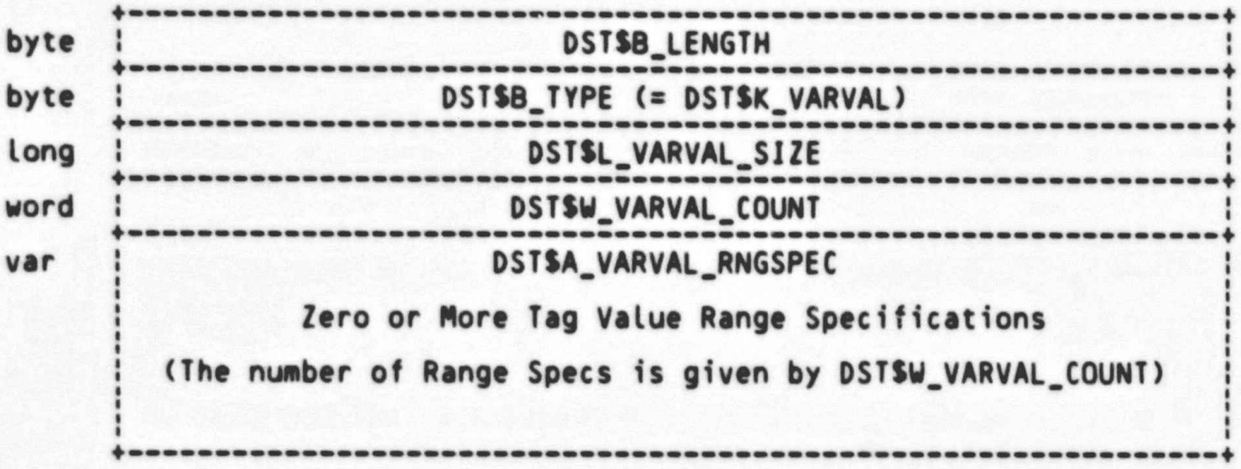
```
        DST$VARBEG_TRAILER = BLOCK[,BYTE] FIELD(DST$VARBEG_TRAILER_FIELDS) %;
```

THE VARIANT VALUE DST RECORD

The Variant Value DST record marks the beginning of a new record variant within a variant set. It also marks the end of the previous variant (if any). It is always found between a Variant Set Begin and a Variant Set End DST record. Since the Variant Set Begin DST record has already specified the tag variable, the Variant Value DST record only specifies the tag value or values that correspond to the present variant. It also specifies the size in bits of this variant if known at compile-time (otherwise zero is specified). The Variant Value DST record is followed by the data DST records (including nested variants if appropriate) which constitute the components of this specific variant.

A variant may have many tag values or tag value ranges. This DST record thus specifies a set of tag value ranges. The way these ranges are specified is described in detail on the following page.

This is the format of the Variant Value DST record:

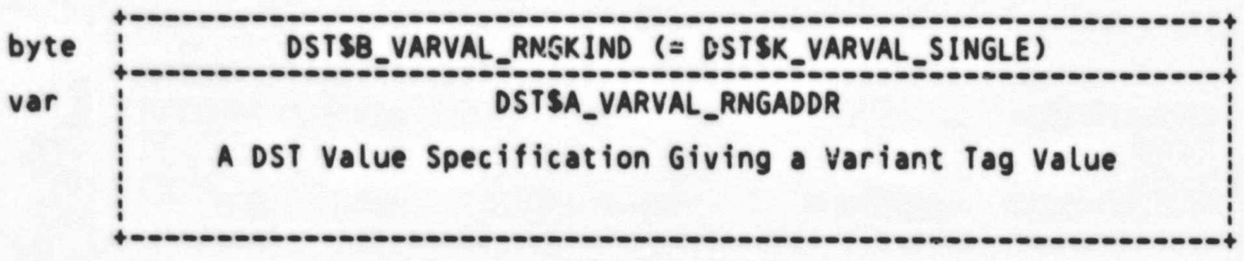


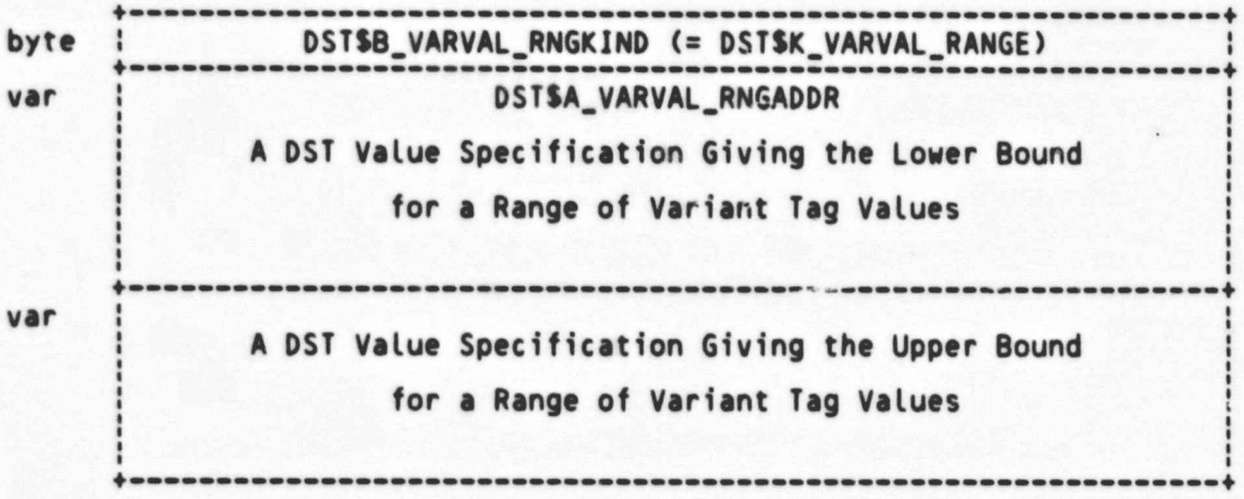
Define the fields of the Variant Value DST record.

```
FIELD DST$VARVAL_FIELDS =
        SET
        DST$L_VARVAL_SIZE       = [ 2, L_ ],    ! Bit size of this variant part
        DST$W_VARVAL_COUNT      = [ 6, W_ ],    ! The number of tag value ranges
                                                !       which follow
        DST$A_VARVAL_RNGSPEC    = [ 8, A_ ]     ! Location where the tag value
                                                !       range specs start
        TES;
```

TAG VALUE RANGE SPECIFICATIONS

Each Tag Value Range Specification in a Variant Value DST record consists of a byte specifying the kind of the range specification followed by one or two Value Specifications. If one Value Specification is given, that gives the tag value--the range consists of that one value. If two Value Specifications are given, they specify the lowest and highest values in the tag value range. The illustrations below show the two possible formats of Tag Value Range Specifications:





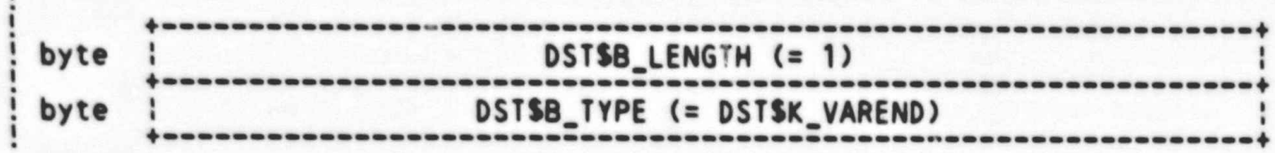
Define the fields of the Tag Value Range Specification.

```
FIELD DST$VARVAL_RNG_FIELDS =
        SET
        DST$B_VARVAL_RNGKIND = [ 0, B_ ],       ! Tag Value Range Spec kind
        DST$A_VALVAL_RNGADDR = [ 1, A_ ]        ! Location of first Value
                                                !       Specification
        TES;
```

```
! Define the possible values of the DST$B_VARVAL_RNGKIND field.
!
LITERAL
        DST$K_VARVAL_SINGLE     = 1,    ! The range consists of a single value
        DST$K_VARVAL_RANGE      = 2;    ! The range is given by a lower and an
                                        !       upper bound (two value specs).
```

THE VARIANT SET END DST RECORD

The Variant Set End DST record marks the end of record variant set; it terminates a set of variants which have the same tag variable. This is the format of the Variant Set End DST record:



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

BLISS data objects are represented by several different kinds of DST records. Ordinary scalar objects, such as simple integers, are represented by the Standard Data DST record or its variants. However, the more specialized BLISS data types such as Vectors, Bitvectors, Blocks, and Blockvectors, are represented by a special DST record called the BLISS Special Cases DST record. Pointers to such objects (e.g., REF VECTOR) are also represented by this DST record. In addition, BLISS field names are represented by their own kind of DST record, the BLISS Field DST record. Both of these record kinds are described in this section.

The BLISS Special Cases DST record and the BLISS Field DST record are supported for BLISS only. They should not be generated by compilers for any other language.

THE BLISS SPECIAL CASES DST RECORD

The BLISS Special Cases DST record is used to describe a number of data objects whose data types are specific to the BLISS language only. This includes such objects as BLISS Vectors, Bitvectors, Blocks, and Blockvectors and pointers to these objects (REF VECTOR, REF BLOCK, and so on). This DST record should not be generated for any language other than BLISS.

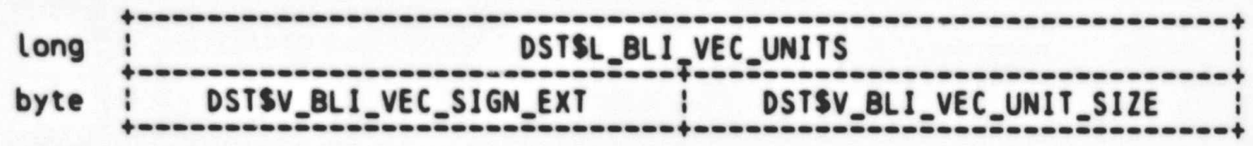
This DST records consists of four parts: The DST header fields, the fields in the set DST$BLI_FIELD, a variable-length set of fields, and the fields in the set DST$BLI_TRAIL_FIELDS. The variable-length set of fields can be empty, consist of the fields in DST$BLI_VEC_FIELDS, the fields in DST$BLI_BITVEC_FIELDS, the fields in DST$BLI_BLOCK_FIELDS, or the fields in DST$BLI_BLKVEC_FIELDS. Which set of fields appears in the variable-length part depends on the value of BLI$V_BLI_STRUC, which indicates which type of symbol is being defined.

This is thus the format of the BLISS Special Cases DST record:

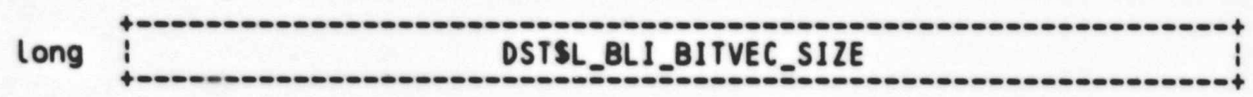
byte	DST$B_LENGTH		
byte	DST$B_TYPE (= DST$K_BLI)		
byte	DST$B_BLI_LNG		
byte	DST$B_BLI_FORMAL		
byte	DST$B_BLI_VFLAGS		
byte	BLI_REF	Unused--Must Be Zero	DST$V_BLI_STRUC
var	DST$A_BLI_SYM_ATTR Variable-Length Portion of DST Record		
long	DST$L_BLI_VALUE		
byte	DST$B_BLI_NAME		
var	The BLISS Symbol Name in ASCII (The name's length is given by DST$B_BLI_NAME)		
long	DST$L_BLI_SIZE		

The variable-length portion of the DST record can have several forms as discussed above. One possibility is that it is absent altogether. This occurs if the DST$V_BLI_STRUC field contains DST$K_BLI_NOSTRUC.

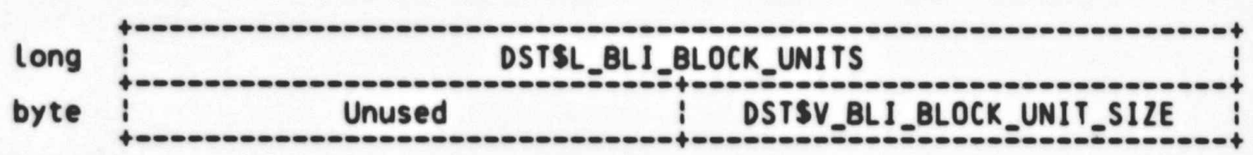
However, if DST$V_BLI_STRUC has the value DST$K_BLI_VEC, the variable-length portion of the DST record has the following format:



If DST$V_BLI_STRUC has the value DST$K_BLI_BITVEC, the variable-length portion of the DST record has the following format:



If DST$V_BLI_STRUC has the value DST$K_BLI_BLOCK, the variable-length portion of the DST record has the following format:



If DST$V_BLI_STRUC has the value DST$K_BLI_BLKVEC, the variable-length portion of the DST record has the following format:

Define the fields in the header portion of the BLISS Special Cases DST Record.

```
FIELD DST$BLI_FIELDS =
        SET
        DST$B_BLI_LNG           = [ 2, B_ ],    ! Length in bytes of the set of
                                                !      fields between this one
                                                !      and TRAIL_FIELDS
                                                ! between 3 and 72
        DST$A_BLI_TRLR1         = [ 3, A_ ],    ! The first trailer is at this
                                                !      location + DST$B_BLI_LNG
        DST$B_BLI_FORMAL        = [ 3, B_ ],    ! Flag set if this symbol is a
                                                !      routine formal parameter
        DST$B_BLI_VFLAGS        = [ 4, B_ ],    ! Value access information
        DST$B_BLI_SYM_TYPE      = [ 5, B_ ],    ! The type of the BLISS symbol
                                                !      as described by the fol-
                                                !      lowing sub-fields:
        DST$V_BLI_STRUC         = [ 5, V_(0,3) ], ! The structure of this symbol
        ! Unused                = [ 5, V_(3,4) ], ! This field Must Be Zero
        DST$V_BLI_REF           = [ 5, V_(7,1) ], ! Flag set if this is a REF
                                                !      (1 = REF, 0 = no REF)
        DST$A_BLI_SYM_ATTR      = [ 6, A_  ]    ! Address of variable length
                                                !      attribute segment in
                                                !      this DST record
        TES;

! These are the possible values of the DST$B_BLI_STRUC field.
!
LITERAL
        DST$K_BLI_NOSTRUC       = 0,    ! Not a BLISS structure
        DST$K_BLI_VEC           = 1,    ! BLISS Vector
        DST$K_BLI_BITVEC        = 2,    ! BLISS Bitvector
        DST$K_BLI_BLOCK         = 3,    ! BLISS Block
        DST$K_BLI_BLKVEC        = 4;    ! BLISS Blockvector

! Define the fields in the variable-length part of the BLISS Special Cases
! DST record when the value of the BLI$V_BLI_STRUC field is DST$K_BLI_VEC.
! This field describes a BLISS Vector.
!
FIELD DST$BLI_VEC_FIELDS =
        SET
        DST$L_BLI_VEC_UNITS     = [ 6, L_ ],    ! Number of elements allocated
                                                !      in the vector
        DST$V_BLI_VEC_UNIT_SIZE = [ 10, V_(0,4) ], ! The vector element unit
                                                !      size: 1 = byte, 2 =
                                                !      word, and 4 = longword
        DST$V_BLI_VEC_SIGN_EXT  = [ 10, V_(4,4)] ! Sign extension flag:
                                                !      1 = sign extension
                                                !      0 = no sign extension
        TES;

! Define the fields in the variable-length part of the BLISS Special Cases
! DST record when the value of the BLI$V_BLI_STRUC field is DST$K_BLI_BITVEC.
! This field describes a BLISS Bitvector.
!
FIELD DST$BLI_BITVEC_FIELDS =
```

```
        SET
        DST$L_BLI_BITVEC_SIZE = [ 6, L_ ] ! The number of bits in the bitvector
        TES;


! Define the fields in the variable-length part of the BLISS Special Cases
! DST record when the value of the BLI$V_BLI_STRUC field is DST$K_BLI_BLOCK.
! These fields describe a BLISS Block.
!
FIELD DST$BLI_BLOCK_FIELDS =
        SET
        DST$L_BLI_BLOCK_UNITS     = [ 6, L_ ],  ! The number of units allocated
                                                !      in the block

        DST$V_BLI_BLOCK_UNIT_SIZE = [ 10, V_(0,4) ] ! The unit size of the
                                                !      block: 1 = byte, 2 =
                                                !      word, and 4 = longword
        TES;


! Define the fields in the variable-length part of the BLISS Special Cases
! DST record when the value of the BLI$V_BLI_STRUC field is DST$K_BLI_BLKVEC.
! These fields describe a BLISS Blockvector.
!
FIELD DST$BLI_BLKVEC_FIELDS =
        SET
        DST$L_BLI_BLKVEC_BLOCKS     = [ 6, L_ ], ! The number of blocks in the
                                                 !      blockvector
        DST$L_BLI_BLKVEC_UNITS      = [ 10, L_ ],! The number of units per block
        DST$B_BLI_BLKVEC_UNIT_SIZE = [ 14, B_ ] ! The block unit size: 1 = byte,
                                                 !      2 = word, 4 = longword
        TES;


! Define the fields in the first trailer portion of the BLISS Special Cases
! DST record.  Also define the declaration macro.
!
FIELD DST$BLI_TRAIL1_FIELDS =
        SET
        DST$L_BLI_VALUE = [ 0, L_ ],     ! Value longword, interpreted
                                         !      according to contents of
                                         !      DST$B_BLI_VFLAGS
        DST$B_BLI_NAME  = [ 4, B_ ],     ! Count byte of the symbol name
                                         !      Counted ASCII string
        DST$A_BLI_TRLR2 = [ 5, A_ ]      ! The second trailer starts at this
                                         !      location + DST$B_BLI_NAME
        TES;

MACRO
        DST$BLI_TRAILER1 = BLOCK[,BYTE] FIELD(DST$BLI_TRAIL1_FIELDS) %;

! Define the fields in the second trailer portion of the BLISS Special Cases
! DST record.  Also define the declaration macro.
!
FIELD DST$BLI_TRAIL2_FIELDS =
```

```
        SET
        DST$L_BLI_SIZE  = [ 0, L_ ]     ! Size of the Bliss data item in bytes
        TES;

MACRO
        DST$BLI_TRAILER2 = BLOCK[,BYTE] FIELD(DST$BLI_TRAIL2_FIELDS) %;
```

THE BLISS FIELD DST RECORD

The BLISS Field DST record describes a BLISS field name. BLISS field names are declared in FIELD declarations in BLISS. Each BLISS field name is bound to an n-tuple of numbers. Usually the n-tuple is a four-tuple and the numbers represent a byte or longword offset, the bit offset within that byte or longword, the bit length of the field being described, and a sign-extension flag. DEBUG supports references to such fields in BLISS Blocks and Blockvectors. However, a BLISS field can be any n-tuple. If n is not 4, the field name can only be used in EXAMINE commands, but not in Block or Blockvector references.

The BLISS Field DST record should not be generated for any language other than BLISS. This is the format of the record:

byte	DST$B_LENGTH
byte	DST$B_TYPE (= DST$K_BLIFLD)
byte	DST$B_BLIFLD_UNUSED
long	DST$L_BLIFLD_COMPS
byte	DST$B_BLIFLD_NAME
var	The Name of the BLISS Field in ASCII (The name's length is given by DST$B_BLIFLD_NAME)
var	A Vector of Longwords Containing the Integer Values of the Components of the BLISS Field Definition (The number of values is given by DST$B_BLIFLD_COMPS)

Define the fields of the BLISS Field DST record.

```
FIELD DST$BLIFLD_FIELDS =
        SET
        DST$B_BLIFLD_UNUSED     = [ 2, B_ ],    ! Unused--Must Be Zero
        DST$L_BLIFLD_COMPS      = [ 3, L_ ],    ! The number of components
        DST$B_BLIFLD_NAME       = [ 7, B_ ]     ! The count byte of the field
                                                !       name Counted ASCII string

        TES;
```

L A B E L D S T R E C O R D S

Labels are represented by two different DST records. A label, in the sense used here, is a symbol bound to an instruction address. Labels do not include routine, lexical block, and entry point symbols, however. A label can be represented by either a Label DST record or a Label-or-Literal DST record. The Label-or-Literal DST record is intended only for language MACRO, it appears. (The history on the origin and intent of this record is unclear, however.) All other languages should use the Label DST record for labels.

THE LABEL DST RECORD

The Label DST record specifies the name and address of a label in the the current module. A label in this sense is always bound to an instruction address, not a data address. This is the DST record normally used for labels in high-level languages. The DST$L_VALUE field of this record contains the code address to which the label is bound.

This is the format of the Label DST record:

byte	DST$B_LENGTH
byte	DST$B_TYPE (= DST$K_LABEL)
byte	Unused--Must Be Zero
long	DST$L_VALUE
byte	DST$B_NAME
var	The Label Name in ASCII (The name's length is given by DST$B_NAME)

THE LABEL-OR-LITERAL DST RECORD

The Label-or-Literal DST record specifies the name and address of a label (meaning a code location) or the name and value of an integer literal (a named constant). It is not entirely clear why this DST record exists since labels can be described by Label DST records and integer literals can be described with Standard Data DST records. Most likely this DST record was intended for language MACRO where there is little distinction between labels and literals; one is relocatable and the other is not, but that is about all. If DST$V_VALKIND has the value DST$K_VALKIND_ADDR, the symbol is a label and if it has the value DST$K_VALKIND_LITERAL, the symbol is a literal. The address of the label or the value of the literal is found in the DST$L_VALUE field. It is recommended that high-level languages avoid this DST record and use the Label DST record or the Standard Data DST record instead.

This is the format of the Label-or-Literal DST record:

byte	DST$B_LENGTH	
byte	DST$B_TYPE (= DST$K_LBLORLIT)	
byte	Unused--Must Be Zero	DST$V_VALKIND
long	DST$L_VALUE	
byte	DST$B_NAME	
var	The Label or Literal Name in ASCII (The name's length is given by DST$B_NAME)	

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

The Entry Point DST record describes an ENTRY name in the FORTRAN or PL/I sense. In other words, it describes a secondary entry point to the routine within which this DST record is nested. This record should never be generated for the main entry point to a routine since that entry point is already described by the Routine Begin DST record. An entry point described by the Entry Point DST record is always assumed to be called through the CALLS/CALLG instructions (not JSB/BSB). The DST$L_VALUE field contains the address of the entry point.

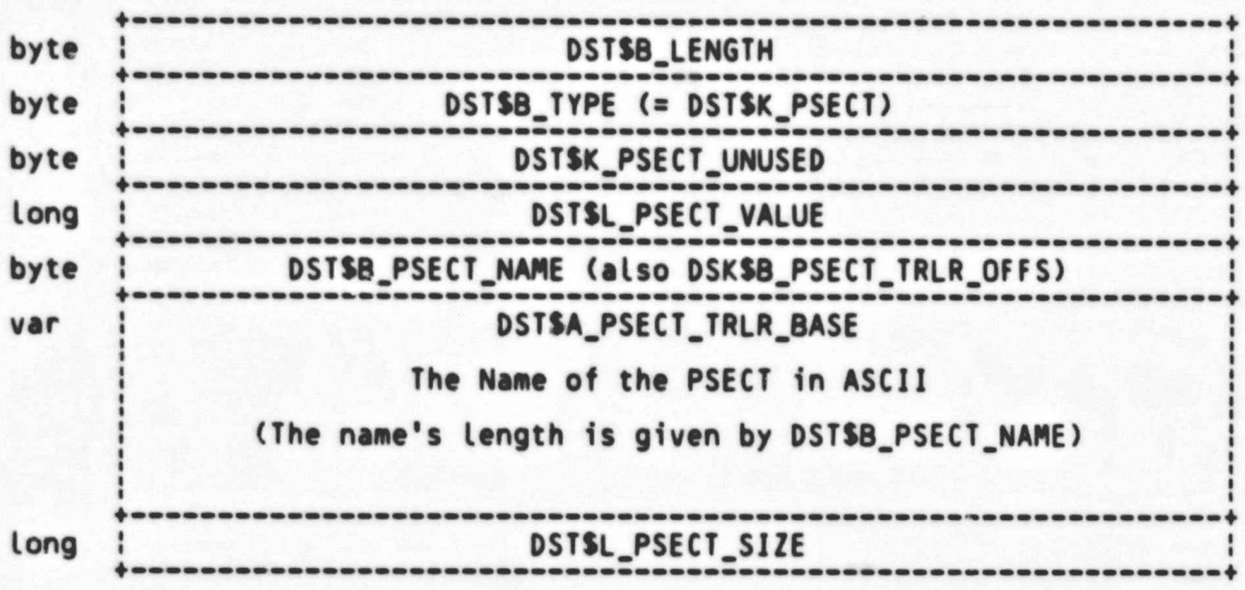
This is the format of the Entry Point DST record:

byte	DST$B_LENGTH
byte	DST$B_TYPE (= DST$K_ENTRY)
byte	Unused--Must Be Zero
long	DST$L_VALUE
byte	DST$B_NAME
var	The Entry Point Name in ASCII (The name's length is given by DST$B_NAME)

T H E P S E C T D S T R E C O R D

The PSECT DST record specifies the name, address, and length of a PSECT, where a PSECT is a Program Section in the linker sense. PSECT DST records are only used for language MACRO where it is possible to generate code or data at the beginning of a PSECT without having any other label on that code. DEBUG ignores PSECT DST records for all other languages since high-level languages have other code and data labels that are more appropriate.

This is the format of the PSECT DST record:

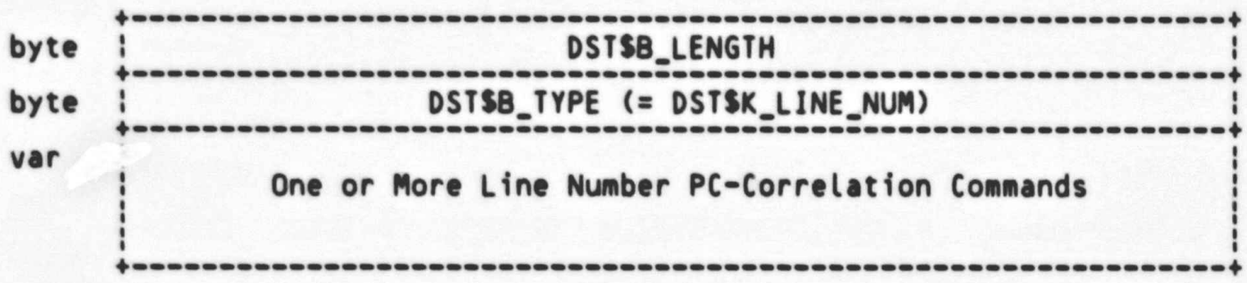



```
! Define the fields of the PSECT DST record.

FIELD DST$PSECT_FIELDS =
        SET
        DST$B_PSECT_UNUSED      = [ 2, B_ ],    ! Unused--Must Be Zero
        DST$L_PSECT_VALUE       = [ 3, L_ ],    ! Start address of the PSECT
        DST$B_PSECT_NAME        = [ 7, B_ ],    ! The coutn byte in the PSECT
                                                !     name COunted ASCII string
        DST$B_PSECT_TRLR_OFFS   = [ 7, B_ ],    ! Byte offset to the PSECT DST
                                                !     record trailer fields
        DST$A_PSECT_TRLR_BASE   = [ 8, A_ ]     ! Base address for offset to
                                                !     DST record trailer fields
        TES;

! Define the PSECT DST record trailer fields.  Also define the declaration
! macro.
```

```
!
FIELD DST$PSECT_TRAILER_FIELDS =
        SET
        DST$L_PSECT_SIZE = [ 0, L_ ]    ! Number of bytes in the PSECT
        TES;

MACRO
        DST$PSECT_TRAILER = BLOCK[,BYTE] FIELD(DST$PSECT_TRAILER_FIELDS) %;

! Note that the address of the PSECT DST record tailer is computed as follows:
!
!       DST_RECORD[DST$A_PSECT_TRLR_BASE] + .DST_RECORD[DST$B_PSECT_TRLR_OFFS]
```

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

The Line Number PC-Correlation DST record specifies the correlation between listing line numbers, as assigned by the compiler, and PC addresses. It thus the means whereby the compiler tells DEBUG where the generated object code for each source line starts and how long it is in bytes. This is the format of the Line Number PC-Correlation DST record:



After the two-byte header, each Line Number PC-Correlation DST record contains a sequence of Line Number PC-Correlation commands. Each such command sets or manipulates one or more state variables used by DEBUG in the interpretation of these commands. The main state variables are the current line number and the current PC address, but there are several others as well. The exact semantics of the various commands are described in the sections that follow.

Line Number PC-Correlation DST records are associated with the module within which they appear. The must thus appear between the Module Begin and the Module End DST records for the current module. There are no further restrictions on where they may appear, however. In particular, they need not be nested within the routines or lexical blocks that they describe. It is thus legal to generate all Line Number PC-Correlation DST records for a module after the last Routine End DST record, for instance. These records can also be interspersed between Routine and Block Begin and End records in any way convenient for the compiler implementer. However it is done, DEBUG treats them as belonging to the module as a whole.

The Line Number PC-Correlation information may be spread over as many DST records as necessary. No Line Number PC-Correlation command may be broken across record boundaries, but otherwise the Line Number PC-Correlation DST records within a module are considered to constitute a single command stream. The Continuation DST record may not be used to continue Line Number PC-Correlation DST records.

```
! Define the fields of the Line Number PC-Correlation DST record.
!
FIELD DST$LINE_NUM_FIELDS =
        SET
        DST$A_LINE_NUM_DATA = [ 2, A_ ] ! Start address of PC-correlation data
        TES;


!       LINE NUMBER PC-CORRELATION COMMANDS
!
!       Each PC-Correlation command consists of a command byte possibly fol-
!       lowed by a parameter byte, word, or longword.  The presence, size, and
!       meaning of the parameter field is determined by the command byte.  This
!       illustration summarizes the structure of one command:
!
!
!        +-------------------------------------------------------------+
! byte   |                         COMMAND_BYTE                        |
!        +-------------------------------------------------------------+
! var    |                                                             |
!        |                  Zero or One Parameter Field                |
!        |                                                             |
!        |                   (Byte, Word, or Longword)                 |
!        |                                                             |
!        +-------------------------------------------------------------+
!
!
!       The command byte contains a command code.  If this command code is
!       negative, this is a Delta-PC command.  A Delta-PC command specifies
!       by how many bytes to increment the PC to get to the start of the
!       next line (see detailed description below).  This byte count is en-
!       coded directly in the command byte: If the command code is negative,
!       its negative is the PC increment.  The Delta-PC command has no param-
!       eter field.  If the command code is positive, it specifies some other
!       command as described below.  In this case, there may be a parameter
!       field, depending on the command code.
!
!
! Define the command codes allowed in Line Number PC-Correlation commands.
! If the command code is zero or negative, the command is a one-byte Delta-PC
! command.  Here we define the command-code range for the Delta-PC command.
!
LITERAL
        DST$K_DELTA_PC_LOW      = -128, ! The lower bound on Delta-PC commands
        DST$K_DELTA_PC_HIGH     = 0;    ! The upper bound on Delta-PC commands

! Define the PC-correlation command codes other than the Delta-PC command.
! These command codes are always positive.
!
LITERAL
```

```
        DST$K_DELTA_PC_W        = 1,    ! Delta-PC Word command
        DST$K_DELTA_PC_L        = 17,   ! Delta-PC Longword command
        DST$K_INCR_LINUM        = 2,    ! Increment Line Number Byte command
        DST$K_INCR_LINUM_W      = 3,    ! Increment Line Number Word command
        DST$K_INCR_LINUM_L      = 18,   ! Increment Line Number Longword command
        DST$K_SET_LINUM_INCR    = 4,    ! Set Line Number Increment Byte command
        DST$K_SET_LINUM_INCR_W  = 5,    ! Set Line Number Increment Word command
        DST$K_RESET_LINUM_INCR  = 6,    ! Reset Line Number Increment command
        DST$K_BEG_STMT_MODE     = 7,    ! Begin Statement Mode command
        DST$K_END_STMT_MODE     = 8,    ! End Statement Mode command
        DST$K_SET_STMTNUM       = 13,   ! Set Statement Number Byte command
        DST$K_SET_LINUM_B       = 19,   ! Set Line Number Byte command
        DST$K_SET_LINUM         = 9,    ! Set Line Number Word command
        DST$K_SET_LINUM_L       = 20,   ! Set Line Number Longword command
        DST$K_SET_PC            = 10,   ! Set Relative PC Byte command
        DST$K_SET_PC_W          = 11,   ! Set Relative PC Word command
        DST$K_SET_PC_L          = 12,   ! Set Relative PC Longword command
        DST$K_SET_ABS_PC        = 16,   ! Set Absolute PC Longword command
        DST$K_TERM              = 14,   ! Terminate Line Byte command
        DST$K_TERM_W            = 15,   ! Terminate Line Word command
        DST$K_TERM_L            = 21,   ! Terminate Line Longword command
        !                               ! -------------------------
        DST$K_PCCOR_LOW         = -128, ! Smallest value allowed in the first
                                        !       byte of a PC-correlation command
        DST$K_PCCOR_HIGH        = 21;   ! Largest value allowed in the first
                                        !       byte of a PC-correlation command


!       The parameter field, if present, contains an unsigned byte, unsigned
!       word, or longword value.  The possible PC-Correlation command formats
!       thus look as follows:

!
!       +-------------------------------------------------------------+
!  byte !                          COMMAND_BYTE                       !
!       +-------------------------------------------------------------+
!
!       +-------------------------------------------------------------+
!  byte !                          COMMAND_BYTE                       !
!       +-------------------------------------------------------------+
!  byte !                NEXT_UNS_BYTE (Unsigned Byte Value)          !
!       +-------------------------------------------------------------+
!
!       +-------------------------------------------------------------+
!  byte !                          COMMAND_BYTE                       !
!       +-------------------------------------------------------------+
!  word !                NEXT_UNS_WORD (Unsigned Word Value)          !
!       +-------------------------------------------------------------+
!
!       +-------------------------------------------------------------+
!  byte !                          COMMAND_BYTE                       !
!       +-------------------------------------------------------------+
!  long !                  NEXT_UNS_LONG (Longword Value)             !
```

+----------+

PC-CORRELATION COMMAND SEMANTICS

The individual commands are described separately below. To clarify what these commands actually do, each is followed by a formal semantic description using BLISS-like pseudo-code. This description show what the command does to a number of state variables used by DEBUG when interpreting these commands. The state variables are the following:

```
CURRENT_LINE -- The current line number.
CURRENT_STMT -- The current statement number.
CURRENT_INCR -- The current line number increment.
CURRENT_STMT_MODE -- The statement mode flag; set to TRUE when
                     statement mode is set, set to FALSE otherwise;
START_PC     -- The start address of the lowest-address routine
                in the current module;
CURRENT_PC   -- The current PC value (code address).
CURRENT_MARK -- The line-open/line-closed flag; set to LINE_OPEN
                when line numbers are being defined and set to
                LINE_CLOSED when a routine has been terminated
                and new lines are not being defined.
```

The initial values of these state variables when the PC-Correlation commands for a given module are interpreted are as follows:

```
CURRENT_LINE = 0;
CURRENT_STMT = 1;
CURRENT_INCR = 1;
CURRENT_STMT_MODE = FALSE;
START_PC = Start address of the lowest-address
                routine in the current module;
CURRENT_PC = START_PC;
CURRENT_MARK = LINE_CLOSED;
```

The sections below describe the format and semantics of each of the individual PC-Correlation commands.

THE DELTA-PC COMMAND

This command defines a correlation between a line number and a PC value. The current line number is incremented by the current increment value (normally 1) and the current PC value is incremented by the negative of the command byte. The resulting line number then has the resulting PC value. In other words, both the line number and the PC value are incremented before the correlation is established. The PC increment value (the negative of the command code) thus specifies how many bytes to go forward to get to the start of the line being defined. These are the formal semantics of the command:

```
        IF CURRENT_STMT_MODE
        THEN
            CURRENT_STMT = CURRENT_STMT + 1
        ELSE
            CURRENT_LINE = CURRENT_LINE + CURRENT_INCR;

        CURRENT_PC = CURRENT_PC - PC_COMMAND[COMMAND_BYTE];
        CURRENT_MARK = LINE_OPEN;
```

The value of CURRENT_PC now contains the start address of the listing line specified by the values of CURRENT_LINE and CURRENT_STMT. Note that line-open mode is now set.

THE DST$K_DELTA_PC_W COMMAND

This command is like the normal Delta-PC command except that the PC increment value is given in an unsigned word following the command code. These are the semantics:

```
        IF CURRENT_STMT_MODE
        THEN
            CURRENT_STMT = CURRENT_STMT + 1
        ELSE
            CURRENT_LINE = CURRENT_LINE + CURRENT_INCR;

        CURRENT_MARK = LINE_OPEN;
        CURRENT_PC = CURRENT_PC + PC_COMMAND[NEXT_UNS_WORD];
```

The value of CURRENT_PC now contains the start address of the listing line specified by the values of CURRENT_LINE and CURRENT_STMT. Note that line-open mode is now set.

THE DST$K_DELTA_PC_L COMMAND

This command is like the normal Delta-PC command except that the PC increment value is given in an unsigned longword following the command code. These are the semantics:

```
        IF CURRENT_STMT_MODE
        THEN
            CURRENT_STMT = CURRENT_STMT + 1
        ELSE
            CURRENT_LINE = CURRENT_LINE + CURRENT_INCR;

        CURRENT_MARK = LINE_OPEN;
```

```
        CURRENT_PC = CURRENT_PC + PC_COMMAND[NEXT_UNS_LONG];
```

The value of CURRENT_PC now contains the start address of the listing line specified by the values of CURRENT_LINE and CURRENT_STMT. Note that line-open mode is now set.

THE DST$K_INCR_LINUM COMMAND

This command increments the current line number by the value given in the unsigned byte following the command code. If statement mode is set, the current statement is reset to 1 as well. These are the formal semantics of the command:

```
        CURRENT_LINE = CURRENT_LINE + PC_COMMAND[NEXT_UNS_BYTE];
        IF CURRENT_STMT_MODE THEN CURRENT_STMT = 1;
```

THE DST$K_INCR_LINUM_W COMMAND

This command increments the current line number by the value given in the unsigned word following the command code. If statement mode is set, the current statement is reset to 1 as well. These are the formal semantics of the command:

```
        CURRENT_LINE = CURRENT_LINE + PC_COMMAND[NEXT_UNS_WORD];
        IF CURRENT_STMT_MODE THEN CURRENT_STMT = 1;
```

THE DST$K_INCR_LINUM_L COMMAND

This command increments the current line number by the value given in the unsigned longword following the command code. If statement mode is set, the current statement is reset to 1 as well. These are the formal semantics of the command:

```
        CURRENT_LINE = CURRENT_LINE + PC_COMMAND[NEXT_UNS_LONG];
        IF CURRENT_STMT_MODE THEN CURRENT_STMT = 1;
```

THE DST$K_SET_LINUM_INCR COMMAND

This command set the current line number increment value to the value specified in the unsigned byte following the command code. If state-

ment mode is set, the current statement number is set to 1. These are the formal semantics of the command:

```
CURRENT_INCR = PC_COMMAND[NEXT_UNS_BYTE];
IF CURRENT_STMT_MODE THEN CURRENT_STMT = 1;
```

THE DST$K_SET_LINUM_INCR_W COMMAND

This command set the current line number increment value to the value specified in the unsigned word following the command code. If statement mode is set, the current statement number is set to 1. These are the formal semantics of the command:

```
CURRENT_INCR = PC_COMMAND[NEXT_UNS_WORD];
IF CURRENT_STMT_MODE THEN CURRENT_STMT = 1;
```

THE DST$K_RESET_LINUM_INCR COMMAND

This command resets the current line number increment value to 1. If statement mode is set, the current statement number is set to 1 as well. These are the semantics:

```
CURRENT_INCR = 1;
IF CURRENT_STMT_MODE THEN CURRENT_STMT = 1;
```

THE DST$K_BEG_STMT_MODE COMMAND

This command sets statement mode, meaning that subsequent Delta-PC commands will increment the current statement number within the current line and not the current line itself. This command is only allowed in the line-open state. Statement mode can optionally be used by languages that have multiple statements per line. This command also set the current statement number to 1. These are the semantics:

```
IF CURRENT_MARK NEQ LINE_OPEN THEN SIGNAL(Invalid DST Record);
CURRENT_STMT_MODE = TRUE;
CURRENT_STMT = 1;
```

THE DST$K_END_STMT_MODE COMMAND

This command clears statement mode so that that subsequent Delta-PC commands will again increment the current line number, not the statement number. The command also set the current statement number to 1. These are the semantics:

```
CURRENT_STMT_MODE = FALSE;
CURRENT_STMT = 1;
```

THE DST$K_SET_LINUM_B COMMAND

This command sets the current line number to the value specified in the unsigned byte that follows the command code. These are the semantics:

```
CURRENT_LINE = PC_COMMAND[NEXT_UNS_BYTE];
```

THE DST$K_SET_LINUM COMMAND

This command sets the current line number to the value specified in the unsigned word that follows the command code. These are the semantics:

```
CURRENT_LINE = PC_COMMAND[NEXT_UNS_WORD];
```

THE DST$K_SET_LINUM_L COMMAND

This command sets the current line number to the value specified in the longword that follows the command code. These are the semantics:

```
CURRENT_LINE = PC_COMMAND[NEXT_UNS_LONG];
```

THE DST$K_SET_STMTNUM COMMAND

This command sets the current statement number to the value specified in the unsigned word that follows the command code. The command should only be used when statement mode is set. These are the semantics:

```
CURRENT_STMT = PC_COMMAND[NEXT_UNS_WORD];
```

THE DST$K_SET_PC COMMAND

This command sets the current PC value to be the value specified in the unsigned byte following the command code added to the start address of the lowest-address routine in the current module. This command is only allowed in the line-closed state. These are the formal semantics:

```
IF CURRENT_MARK NEQ LINE_CLOSED THEN SIGNAL(Invalid DST Record);
CURRENT_PC = START_PC + PC_COMMAND[NEXT_UNS_BYTE];
```

THE DST$K_SET_PC_W COMMAND

This command sets the current PC value to be the value specified in the unsigned word following the command code added to the start address of the lowest-address routine in the current module. This command is only allowed in the line-closed state. These are the formal semantics:

```
IF CURRENT_MARK NEQ LINE_CLOSED THEN SIGNAL(Invalid DST Record);
CURRENT_PC = START_PC + PC_COMMAND[NEXT_UNS_WORD];
```

THE DST$K_SET_PC_L COMMAND

This command sets the current PC value to be the value specified in the longword following the command code added to the start address of the lowest-address routine in the current module. This command is only allowed in the line-closed state. These are the formal semantics:

```
IF CURRENT_MARK NEQ LINE_CLOSED THEN SIGNAL(Invalid DST Record);
CURRENT_PC = START_PC + PC_COMMAND[NEXT_UNS_LONG];
```

THE DST$K_SET_ABS_PC COMMAND

This command sets the current PC value to be the absolute address specified in the longword following the command code. This command is only allowed in the line-closed state. These are the formal semantics:

```
IF CURRENT_MARK NEQ LINE_CLOSED THEN SIGNAL(Invalid DST Record);
CURRENT_PC = PC_COMMAND[NEXT_UNS_LONG];
```

THE DST$K_TERM COMMAND

This command terminates the PC-Correlation command sequence for the current routine or other program unit and specifies the number of bytes in the last line specified by a Delta-PC command. Since the Delta-PC command specifies how many bytes precede the line being defined, the Terminate command is needed to say how many bytes are in that line (i.e., how many bytes will increment the PC to the first byte past the current program unit). The number of bytes in the last line is specified by the unsigned byte following the command code. This command also sets the line-closed state. These are the semantics of the command:

```
CURRENT_PC = CURRENT_PC + PC_COMMAND[NEXT_UNS_BYTE];
CURRENT_MARK = LINE_CLOSED;
```

THE DST$K_TERM_W COMMAND

This command terminates the PC-Correlation command sequence for the current routine or other program unit and specifies the number of bytes in the last line of that program unit. It is a variant of the DST$K_TERM command described above. The number of bytes in the last line is specified by the unsigned word following the command code. This command also sets the line-closed state. These are the semantics of the command:

```
CURRENT_PC = CURRENT_PC + PC_COMMAND[NEXT_UNS_WORD];
CURRENT_MARK = LINE_CLOSED;
```

THE DST$K_TERM_L COMMAND

This command terminates the PC-Correlation command sequence for the current routine or other program unit and specifies the number of bytes in the last line of that program unit. It is a variant of the DST$K_TERM command described above. The number of bytes in the last line is specified by the longword following the command code. This command also sets the line-closed state. These are the semantics of the command:

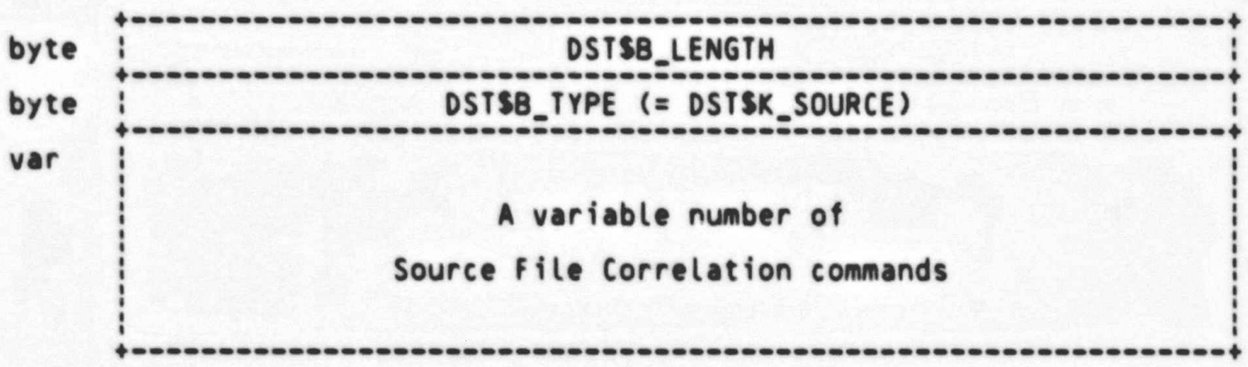
```
CURRENT_PC = CURRENT_PC + PC_COMMAND[NEXT_UNS_LONG];
CURRENT_MARK = LINE_CLOSED;
```

END OF LINE NUMBER PC-CORRELATION DST RECORD DESCRIPTION.

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

The Source File Correlation DST record is used to specify the correlation between listing line numbers on the one hand and source files and source file record numbers on the other. These records enable DEBUG to display source lines during the debugging session.

The Source File Correlation DST record has the following format:



After the length and DST type bytes, the record consists of a sequence of Source File Correlation commands. These commands specify what source files contributed source lines to this module and how the module's listing line numbers are lined up with the source files and record numbers within those source files. The available commands are described individually below.

If the Source File Correlation commands needed to fully describe the current module will not fit in a single Source Line Correlation DST record, they can be spread over any number of such DST records. These records will be processed sequentially, in the order that they appear, until there are no more such records for the current module.

The purpose of the Source File Correlation commands is to allow DEBUG to construct a table of correlations between line numbers and source records. A "line number" in this context means the listing line number. This is the line number which is printed in the program listing and is output to the PC-Correlation DST records by the compiler. (PC-Correlation DST records correlate listing line numbers with Program Counter values.) A corresponding source line is identified by two things: a source file and a record number within that source file.

The semantics of the Source File Correlation commands can be understood in terms of manipulating three state variables and issuing one command. The three state variables are:

LINE_NUM -- The current listing line number.

```
                SRC_FILE -- The File ID of the current source file,
                            i.e. a small integer uniquely defining
                            the source file.
                SRC_REC  -- The record number (in the RMS sense) in
                            the current source file of the current
                            source line.

        LINE_NUM is assumed to have an initial value of 1 while SRC_FILE and
        SRC_REC are initially undefined.  The one command is:

                DEFINE(LINE_NUM, SRC_FILE, SRC_REC)

        This command declares that line number LINE_NUM is associated with the
        source line at record number SRC_REC in the file specified by SRC_FILE.

        Given this, the compiler must output a sequence of Source File Correla-
        tion commands which cause LINE_NUM, SRC_FILE, and SRC_REC to be set up
        appropriately and which cause the proper DEFINE operations to be issued
        to allow DEBUG to generate the correct line number to source record
        correlation table.  (DEBUG may not actually generate the full table,
        but it must be able to generate any part of such a table it needs.)
        The semantics of each Source File Correlation command is described
        below in terms of these state variables and commands.

        Line numbers must be DEFINEd in sequential order, from lowest line
        number to highest line number, in the Source File Correlation commands
        for one module.  The source records these line numbers correlate with
        may be in any order, of course.

        It should be clear from what follows that the source for one module may
        come from many source files.  This can be caused by plus-lists on the
        compiler command (e.g., $FORTRAN/DEBUG A+B+C) and by INCLUDE statements
        in the source.  Also, source lines may come from modules within source
        libraries as well as from independent source files.

        Form feeds in source files, or more precisely source file records which
        contain nothing but a single form feed (CNTL-L) character, are counted
        as individual sources lines in some languages but are ignored (not as-
        signed line numbers) in other languages.  DEBUG will handle either con-
        vention, but DEBUG's default behavior is that form feed records are
        ignored in sources files.  They are not displayed and they do not count
        toward the source file record number of subsequent source records.  To
        make DEBUG count such records, the DST$K_SRC_FORMFEED command must be
        used.


! Define the location of the first command in the DST record.

FIELD DST$SOURCE_FIELDS =
        SET
        DST$A_SRC_FIRST_CMD = [ 2, A_ ] ! Location of first command in record
        TES;

! Define the command codes for all the Source File Correlation commands.
```

```
!
LITERAL
        DST$K_SRC_MIN_CMD       =  1,   ! Minimum command code for CASE ranges
        DST$K_SRC_DECLFILE      =  1,   ! Declare a source file for this module
        DST$K_SRC_SETFILE       =  2,   ! Set the current source file (word)
        DST$K_SRC_SETREC_L      =  3,   ! Set source record number (longword)
        DST$K_SRC_SETREC_W      =  4,   ! Set source record number (word)
        DST$K_SRC_SETLNUM_L     =  5,   ! Set listing line number (longword)
        DST$K_SRC_SETLNUM_W     =  6,   ! Set listing line number (word)
        DST$K_SRC_INCRLNUM_B    =  7,   ! Increment listing line number (byte)
        !                       =  8,   ! Unused--available for future use
        !                       =  9,   ! Unused--available for future use
        DST$K_SRC_DEFLINES_W    = 10,   ! Define N separate lines (word)
        DST$K_SRC_DEFLINES_B    = 11,   ! Define N separate lines (byte)
        !                       = 12,   ! Unused--available for future use
        !                       = 13,   ! Unused--available for future use
        !                       = 14,   ! Unused--available for future use
        !                       = 15,   ! Unused--available for future use
        DST$K_SRC_FORMFEED      = 16,   ! Count Form-Feeds as source records
        DST$K_SRC_MAX_CMD       = 16;   ! Maximum command code for CASE ranges

! Define the fields of the Source Line Correlation commands.  Also define the
! corresponding declaration macros.
!
FIELD DST$SRC_COMMAND_FIELDS =
        SET
        !
        ! Field common to all Source File Correlation commands.
        !
        DST$B_SRC_COMMAND       = [ 0, B_ ],    ! Command code
        !
        ! The fields of the Declare Source File command.
        !
        DST$B_SRC_DF_LENGTH     = [ 1, B_ ],    ! Length of this command
        DST$B_SRC_DF_FLAGS      = [ 2, B_ ],    ! Flag bits--reserved (MBZ)
        DST$W_SRC_DF_FILEID     = [ 3, W_ ],    ! Source file's File ID
        DST$Q_SRC_DF_RMS_CDT    = [ 5, A_ ],    ! Creation date and time or mod-
                                                !    ule insertion date and time
        DST$L_SRC_DF_RMS_EBK    = [ 13, L_ ],   ! End-of-File block number
        DST$W_SRC_DF_RMS_FFB    = [ 17, W_ ],   ! First Free Byte in EOF block
        DST$B_SRC_DF_RMS_RFO    = [ 19, B_ ],   ! Record and File Organization
        DST$B_SRC_DF_FILENAME   = [ 20, B_ ],   ! Source file name counted ASCII
        DST$A_SRC_DF_FILENAME   = [ 21, A_ ],   !      (count byte, string addr)
        !
        ! Fields used to access information in all other commands.
        !
        DST$L_SRC_UNSLONG       = [ 1, L_ ],    ! Unsigned longword parameter
        DST$W_SRC_UNSWORD       = [ 1, W_ ],    ! Unsigned word parameter
        DST$B_SRC_UNSBYTE       = [ 1, B_ ]     ! Unsigned byte parameter
        TES;

! Declare trailer field in the Declare Source File command.
!
FIELD DST$SRC_DECLFILE_TRLR_FIELDS =
```

```
        SET
        DST$B_SRC_DF_LIBMODNAME = [ 0, B_ ],    ! Module name counted ASCII
        DST$A_SRC_DF_LIBMODNAME = [ 1, A_ ]     !       (count byte, string addr)
        TES;

! Declaration macros for Source File Correlation command and trailer blocks.
!
MACRO
        DST$SRC_COMMAND = BLOCK[,BYTE] FIELD(DST$SRC_COMMAND_FIELDS) %,
        DST$SRC_CMDTRLR = BLOCK[,BYTE] FIELD(DST$SRC_DECLFILE_TRLR_FIELDS) %;
```

DECLARE SOURCE FILE (DST$K_SRC_DECLFILE)

This command declares a source file which contributes source lines to the current module. It declares the name of the file, its creation date and time, and various other attributes. The command also assigns a one-word "file ID" to this source file. This is the format of the Declare Source File command:

byte	DBG$B_SRC_COMMAND (= DST$K_SRC_DECLFILE)
byte	DST$B_SRC_DF_LENGTH
byte	DST$B_SRC_DF_FLAGS
word	DST$W_SRC_DF_FILEID
quad	DST$Q_SRC_DF_RMS_CDT
long	DST$L_SRC_DF_RMS_EBK
word	DST$W_SRC_DF_RMS_FFB
byte	DST$B_SRC_DF_RMS_RFO
var	DST$B_SRC_DF_FILENAME
var	DST$B_SRC_DF_LIBMODNAME

The fields in this command are the following:

DST$B_SRC_DF_LENGTH - The length of this command, i.e. the number of bytes remaining in the command after this field.

DST$B_SRC_DF_FLAGS - Bit flags. This field is reserved for future use. At present this field Must Be Zero.

DST$W_SRC_DF_FILEID - The one-word "File ID" of this source file. This File ID, which can later be used in the Set File command, is simply a unique number which the compiler assigns to each source file which contributes source lines to the current module. Each source file thus has a number (the File ID) and is identified by that number in the Set File (DST$K_SRC_SETFILE) command.

DST$Q_SRC_DF_RMS_CDT - The creation date and time of this source file. This quadword quantity should be retrieved with a $XABDAT extended attribute block from RMS via the $OPEN or $DISPLAY system service. The creation date and time should be taken from the XAB$Q_CDT field of the XAB.

If the source file is a module in a source library, this field should contain the module's Insertion Date and Time in the lib-

rary. This value should be retrieved with the LBR$SET_MODULE Librarian call. The library file's creation date is not used.

DST$L_SRC_DF_RMS_EBK - The End-of-File block number for this source file. This longword quantity should be retrieved with a $XABFHC extended attibute block from RMS via the $OPEN or $DISPLAY system service. The End-of-File block number should be taken from the XAB$L_EBK field of the XAB.

This field should be zero for modules in source libraries.

DST$W_SRC_DF_RMS_FFB - The first free byte of the End-of-File block for this source file. This word quantity should be retrieved with a $XABFHC extended attribute block from RMS via the $OPEN or $DISPLAY system service. The first free byte value should be taken from the XAB$W_FFB field of the XAB.

This field should be zero for modules in source libraries.

DST$B_SRC_DF_RMS_RFO - The file organization and record format of this source file. This byte value should be retrieved with a $XABFHC extended attribute block from RMS via the $OPEN or $DISPLAY system service. The file organization and record format should be taken from the XAB$B_RFO field of the XAB.

This field should be zero for modules in source libraries.

DST$B_SRC_DF_FILENAME - The full filename of the source file. This is the fully specified filename, complete with device name and version number, in which all wild cards and logical names have been resolved. This string should be retrieved with a $NAM block from RMS via the $OPEN or $SEARCH system service. The desired string is the "Resultant String" specified by the NAML_RSA, NAMB_RSS, and NAM$B_RSL fields of the $NAM block. Here the file name is represented as a Counted ASCII string (a one-byte character count followed by the name string).

DST$B_SRC_DF_LIBMODNAME - The source library module name (if applicable) or the null string. If the source file is actually a module in a source library, the DST$B_SRC_DF_FILENAME field gives the filename of the source library and the DST$B_SRC_DF_LIBMODNAME field gives the name of the source module within that library. If the source file does not come from a source library, this field (DST$B_SRC_DF_LIBMODNAME) contains the null (zero-length) string. This field is represented as a Counted ASCII string.

SET SOURCE FILE (DST$K_SRC_SETFILE)

This command sets the current source file to the file denoted by the one-word file ID given in the command. The set file is then the file from which further source lines are taken when the corresponding listing lines are defined. This is the format of the command:

byte	DBG$B_SRC_COMMAND (= DST$K_SRC_SETFILE)
word	DST$W_SRC_UNSWORD: The File ID of the desired source file

The semantics of this command is:

```
SRC_FILE := file ID from command
SRC_REC  := set to current source record for this
            source file
```

SET SOURCE RECORD NUMBER LONG (DST$K_SRC_SETREC_L)

This command sets the current source file record number to the longword value specified in the command. Its format is:

byte	DBG$B_SRC_COMMAND (= DST$K_SRC_SETREC_L)
long	DST$L_SRC_UNSLONG: The desired new source record number

The semantics of this command is:

```
SRC_REC := longword value from command
```

SET SOURCE RECORD NUMBER WORD (DST$K_SRC_SETREC_W)

This command set the current source file record number to the word value specified in the command. It is thus a more compact form of the DST$K_SRC_SETREC_L command. Its format is:

byte	DBG$B_SRC_COMMAND (= DST$K_SRC_SETREC_W)
word	DST$W_SRC_UNSWORD: The desired new source record number

The semantics of this command is:

SRC_REC := word value from command

SET LINE NUMBER LONG (DST$K_SRC_SETLNUM_L)

This command set the current listing line number to a longword value specified in the command. Its format is:

byte	DBG$B_SRC_COMMAND (= DST$K_SRC_SETLNUM_L)
long	DST$L_SRC_UNSLONG: The desired listing line number

The semantics of this command is:

LINE_NUM := longword value in commmand

SET LINE NUMBER WORD (DST$K_SRC_SETLNUM_W)

This command sets the current listing line number to a one-word value specified in the command. Its format is:

byte	DBG$B_SRC_COMMAND (= DST$K_SRC_SETLNUM_W)
word	DST$W_SRC_UNSWORD: The desired listing line number

The semantics of this command is:

LINE_NUM := word value in command

INCREMENT LINE NUMBER BYTE (DST$K_SRC_INCRLNUM_B)

This command increments the current listing line number by a one-byte value specified in the command. Its format is:

byte	DBG$B_SRC_COMMAND (= DST$K_SRC_INCRLNUM_B)
byte	DST$B_SRC_UNSBYTE: The desired listing line number increment

The semantics of this command is:

LINE_NUM := LINE_NUM + byte value in command

COUNT FORM-FEEDS AS SOURCE RECORDS (DST$K_SRC_FORMFEED)

This command specifies that DEBUG should count source records which consists of nothing but a Form-Feed character (CNTL-L) as being distinct, numbered source records. In some languages, such records are not considered to be source lines; instead they are regarded as control information. The compiler then does not assign line numbers to them and DEBUG ignores them completely--they are not displayed as part of the source and they do not contribute to the source record numbering of source files. However, if the DST$K_SRC_FORMFEED command is specified in the Source File Correlation DST Record for a module, then such records count as normal records; they can be displayed and they are assigned source file record numbers.

If used, this command must appear before any commands that actually define source lines. Making it the first command in the first Source File Correlation Record for the module is a good choice.

byte	DBG$B_SRC_COMMAND (= DST$K_SRC_FORMFEED)

The semantics of this command is to set a mode flag which says to count Form-Feed records as normal records. The default behavior is to ignore Form-Feed records.

DEFINE N LINES WORD (DST$K_SRC_DEFLINES_W)

This command defines the source file and source record numbers for a specified number of listing line numbers. The specified number is given by a one-word count in the command. The command format is:

byte	DBG$B_SRC_COMMAND (= DST$K_SRC_DEFLINES_W)
word	DST$W_SRC_UNSWORD: The number of lines to define

The semantics of this command is:

```
DO the number of times specified in the command:
        BEGIN
        DEFINE(LINE_NUM, SRC_FILE, SRC_REC);
        LINE_NUM := LINE_NUM + 1;
        SRC_REC := SRC_REC + 1;
        END;
```

DEFINE N LINES BYTE (DST$K_SRC_DEFLINES_B)

This command defines the source file and source record number for a specified number of listing line numbers. The specified number is given by a one-byte count in the command. This is thus a more compact form of the DST$K_SRC_DEFLINES_W command. Its format is:

byte	DBG$B_SRC_COMMAND (= DST$K_SRC_DEFLINES_B)
byte	DST$B_SRC_UNSBYTE: The number of lines to define

The semantics of this command is:

```
DO the number of times specified in the command:
        BEGIN
        DEFINE(LINE_NUM, SRC_FILE, SRC_REC);
        LINE_NUM := LINE_NUM + 1;
        SRC_REC := SRC_REC + 1;
        END;
```

END OF SOURCE FILE CORRELATION DST RECORD DESCRIPTION.

THE DEFINITION LINE NUMBER DST RECORD

NOTE: THIS DST RECORD IS NOT SUPPORTED BY DEBUG V4.0.

The Definition Line Number DST record specifies the listing line number at which a data symbol or other object is defined or declared. The intent is to make use of this information in future DEBUG commands so that a user can see the declaration source line for a specified symbol. The Definition Line Number DST record must immediately follow the data DST record of the data object whose line of definition is being specified.

This is the format of the Definition Line Number DST record:

byte	DST$B_LENGTH (= 6)
byte	DST$B_TYPE = (DST$K_DEF_LNUM)
byte	Unused (Must Be Zero)
long	DST$L_DEF_LNUM_LINE

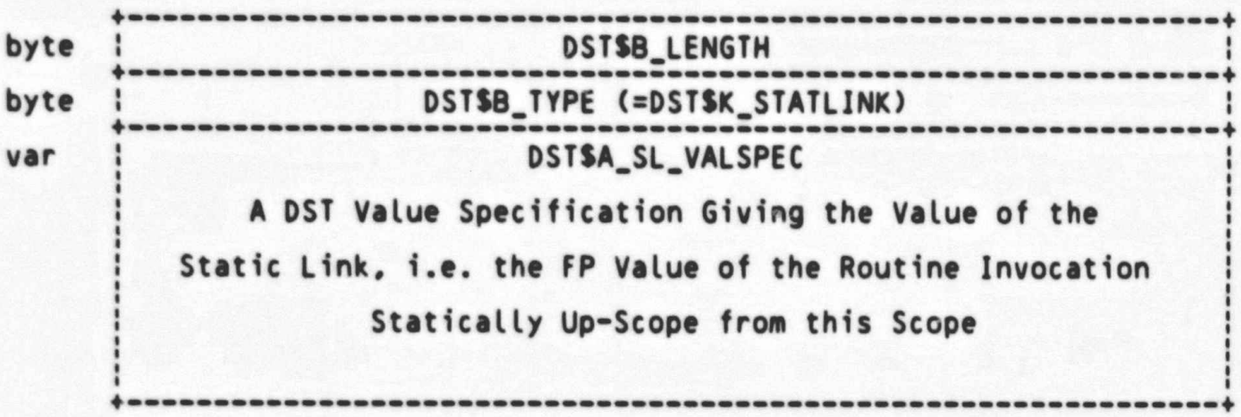
Define the fields of the Definition Line Number DST record. The unused byte in the DST record is reserved for future use.

```
FIELD DST$DEF_LNUM_FIELDS =
        SET
        DST$L_DEF_LNUM_LINE = [ 3, L_ ] ! The definition line number
        TES;
```

T H E S T A T I C L I N K D S T R E C O R D

The Static Link DST record specifies the "Static Link" for a routine. The Static Link is a pointer to the VAX call frame for the proper up-scope invocation of the outer routine within which the present invocation of the present routine is nested. The Static Link is thus used when DEBUG does up-level addressing in response to user commands. A Static Link DST Record is always associated with the inner-most routine within whose Routine-Begin and Routine-End records it is nested. The Static Link DST Record is optional--it need not be used by languages or for routines which do not keep track of static links in their run-time environments. In fact, the Static Link DST record only makes a difference for recursive routines that pass routines as parameters, a fairly obscure situation.

This is the format of the Static Link DST record:




```
! Define the fields of the Static Link DST record.

FIELD DST$STATLINK_FIELDS =
        SET
        DST$A_SL_VALSPEC        = [ 2, A_ ]     ! Location of Value Spec giving
                                                !       the up-scope FP value

        TES;
```

T H E P R O L O G D S T R E C O R D

The Prolog DST record tells DEBUG where to put routine breakpoints. It is used for routines that have prolog code that must be executed before data objects can be freely examined or otherwise accessed from DEBUG. Such prolog code typically sets up stack locations and descriptors for formal parameters or other data objects. By putting routine breakpoints on the first instruction after the prolog code, as specified in the Prolog DST record, DEBUG ensures that all local storage and formal parameters are accessible to the user.

Prolog DST records are optional. If omitted for some routine, DEBUG simply uses the routine start address for routine breakpoints or tracepoints requested by the user. If specified, the Prolog DST record is counted as belonging with the nearest Routine Begin or Entry Point DST record before it, not counting nested routines. Placing the Prolog DST record immediately after the Routine Begin or Entry Point DST record with which it is associated is good practice.

This is the format of the Prolog DST record:

byte	DST$B_LENGTH (=5)
byte	DST$B_TYPE (= DST$K_PROLOG)
long	DST$L_PROLOG_BKPT_ADDR

```
! Define the fields of the Prolog DST record.

FIELD DST$PROLOG_FIELDS =
        SET
        DST$L_PROLOG_BKPT_ADDR  = [ 2, L_ ]     ! The routine breakpoint address
        TES;
```

```
!          T H E   V E R S I O N   N U M B E R   D S T   R E C O R D
!
!
!       The Version Number DST record gives the version number of the compiler
!       that compiled the current module.  The Version Number DST Record must
!       be nested within the Module Begin and Module End DST Records for the
!       module in question.  DEBUG ignores this record except in special cases
!       when it is necessary to distinguish between old and new versions of the
!       compiler that generated a given object module.
!
!       This is the format of the Version Number DST record:
!
!
!       +---------------------------------------------------------------+
! byte  |                        DST$B_LENGTH (= 3)                     |
!       +---------------------------------------------------------------+
! byte  |                  DST$B_TYPE (= DST$K_VERSION)                 |
!       +---------------------------------------------------------------+
! byte  |                        DST$B_VERSION_MAJOR                    |
!       +---------------------------------------------------------------+
! byte  |                        DST$B_VERSION_MINOR                    |
!       +---------------------------------------------------------------+
!
!
! Define the fields of the Version Number DST record.
!
FIELD DST$VERSION_FIELDS =
        SET
        DST$B_VERSION_MAJOR     = [ 2, B_ ],    ! The major version number
        DST$B_VERSION_MINOR     = [ 3, B_ ]     ! The minor version number
        TES;
```

THE COBOL GLOBAL ATTRIBUTE DST RECORD

The COBOL Global Attribute DST record indicates that the symbol whose DST record immediately follows has the COBOL "global" attribute. This attribute specifies that the symbol is visible in nested COBOL scopes (routines) within the scope (routine) in which the symbol is declared. Without this attribute, a symbol is only visible in its scope of declaration but not within any nested scopes. In this regard, COBOL differs from most other languages. DEBUG thus needs to know this attribute in order to implement the COBOL scope rules correctly.

The COBOL Global Attribute DST record is only generated by the COBOL compiler. If it precedes the DST record for some symbol, that symbol is deemed to have the COBOL global attribute; if it omitted, the symbol is deemed not to have the global attribute. DEBUG ignores this attribute for all other languages.

This is the format of the COBOL Global Attribute DST record:

T H E O V E R L O A D E D S Y M B O L D S T R E C O R D

NOTE: THIS DST RECORD IS NOT SUPPORTED BY DEBUG V4.0.

The Overloaded Symbol DST record is used to indicate that a given symbol name is overloaded. The record indicates which other symbols in the DST are possible resolutions to the overloading. It is used by the ADA compiler.

In ADA, it is possible to have more than one routine of the same name in the same scope. If the routine name is R, DEBUG disambiguates the individual instances of the overloaded routine name with the invented names R__1, R__2, R__3, and so on. DEBUG requires the ADA compiler to generate normal DST records for these routines, using the invented names. DEBUG also requires the ADA compiler to generate the Overloaded Symbol DST record with the original overloaded name "R" in order to inform DEBUG of the overloading.

After the length and type fields, this record contains a Counted ASCII string with the name of the overloaded symbol. Following the Counted ASCII string, there is a word field containing a count of the number of overloaded instances of the name in this scope. Next there is a vector of pointers, one for each instance, pointing to the DST records for the instances of the overloaded symbol. These DST pointers consist of byte offsets relative to the start of the whole DST.

This is the format of the Overloaded Symbol DST record:

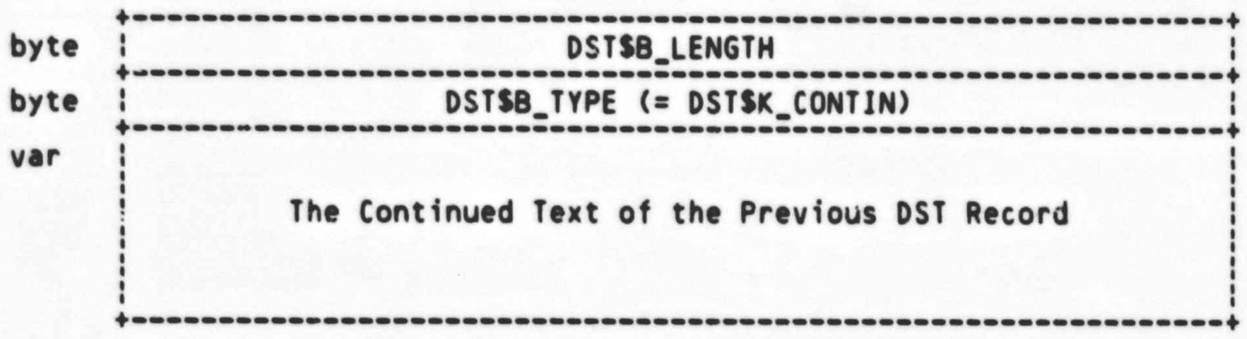
byte	DST$B_LENGTH
byte	DST$B_TYPE (= DST$K_OVERLOAD)
byte	DST$B_OL_NAME
var	The Overloaded Symbol Name in ASCII (The name's length is given by DST$B_OL_NAME)
word	DST$W_OL_COUNT
var	DST$A_OL_VECTOR A Vector of Longword Pointers to the DST Records of the Symbols with Invented Names that Constitute the Instances of this Overloading

```
!       +-----------------------------------------------------------------+
!
!
! Define the fields of the Overloaded Symbol DST record.
!
FIELD DST$OVERLOAD_FIELDS =
        SET
        DST$B_OL_NAME    = [ 2, B_ ],    ! Count byte of the overloaded symbol
                                         !      name Counted ASCII string
        DST$A_OL_TRAILER= [ 3, A_ ]      ! The trailer fields start at this
                                         !      location + .DST$B_OL_NAME
        TES;

! Define the fields of the Overloaded Symbol DST record trailer portion.  Also
! define the corresponding declaration macro.
!
FIELD DST$OVERLOAD_TRLR_FIELDS =
        SET
        DST$W_OL_COUNT  = [ 0, W_ ],     ! Number of instances in this scope
        DST$A_OL_VECTOR = [ 2, A_ ]      ! Vector of DST pointers to instances
                                         !      of overloaded symbol
        TES;

MACRO
        DST$OVERLOAD_TRLR = BLOCK[,BYTE] FIELD(DST$OVERLOAD_TRLR_FIELDS) %;

! This is a short BLISS example of how the trailer fields are accessed:
!
!
        LOCAL
            DSTPTR: REF DST$RECORD,      ! Pointer to DST record
            OVERLOAD_COUNT,              ! The number of overloadings
            OVERLOAD_TRAILER:            ! Pointer to DST record trailer
                REF DST$OVERLOAD_TRLR,   !
            OVERLOAD_VECTOR:             ! Vector of DST-record pointers to the
                REF VECTOR[,LONG];       !     instances of this overloading

        ! Here we assume that DSTPTR points to the Overloaded Symbol DST record.
        !
        OVERLOAD_TRAILER = DSTPTR[DST$A_OL_TRALER] + .DSTPTR[DST$B_OL_NAME];
        OVERLOAD_COUNT = .OVERLOAD_TRAILER[DST$B_OL_COUNT];
        OVERLOAD_VECTOR = OVERLOAD_TRAILER[DST$A_OL_VECTOR];
```

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

When the text of a Debug Symbol Table record is longer than 255 bytes, it is no longer possible to hold that text in a single DST record since the DST$B_LENGTH field cannot hold a value larger than 255. In this case it is necessary to generate the original DST record followed by as many Continuation DST records as necessary to hold the full text. The original DST record then holds at least 100 and at most 255 bytes of text. Each Continuation DST record consists of the standard two-byte header followed by the continued text of the original DST record.

This is the format of the Continuation DST record:



DEBUG reconstitutes a continued DST record by concatenating the text of the first DST record with the text portions of its Continuation DST records. In effect, the first two bytes of each Continuation DST record are stripped out. Any further interpretation of the DST text is then done on the concatenated copy.

Certain kinds of DST records are not allowed to be continued with Continuation DST records. These records are Module Begin, Routine Begin, Block Begin, Label, Label-or-Literal, Entry Point, PSECT, Line Number PC-Correlation, and Source File Correlation DST records. In addition, DST records with fixed sizes, such as Module End and Routine End DST records, are not allowed to be continued. Line Number PC-Correlation and Source File Correlation DST records cannot be continued with Continuation DST records, but one can have multiple such records in one module; they can thus be continued, but through a different mechanism. The records that really need to be continued, such as Standard Data DST records and their variants (Descriptor Format and Trailing Value Specification Format records), Separate Type Specification DST records, and Type Specification DST records, can all be continued using the Continuation DST record mechanism.

Define the fields of the Continuation DST record.

```
FIELD DST$CONTIN_FIELDS =
        SET
        DST$A_CONTIN    =  [ 2, A_ ]    ! Address of continuation text
        TES;
```

O B S O L E T E D S T R E C O R D S

There are several obsolete DST records. These are records that were at one time generated by compilers, but are no longer used by any current version of any Digital compiler. Some of these records were not properly thought out and were abandoned when it was realized that their intended uses could not be implemented. Others were at one time used and useful, but were generated by now-obsolete compilers. Such records are not generated by current compiler versions, and the capabilities they provided are now provided more general mechanisms in other DST records.

None of the obsolete DST records should be generated by any future compilers, and their use will not necessarily be supported by DEBUG.

THE GLOBAL-IS-NEXT DST RECORD

The Global-is-Next DST record is now obsolete. It consisted of just the DST$B_LENGTH byte and the DST$B_TYPE byte. DST$K_GLOBNXT was the type code. The purpose of this record was never properly thought out and no support for it was ever implemented. It should not be generated by any future compilers or compiler versions.

THE EXTERNAL-IS-NEXT DST RECORD

The External-is-Next DST record is now obsolete. It consisted of just the DST$B_LENGTH byte and the DST$B_TYPE byte. DST$K_EXTRNXT was the type code. The purpose of this record was never properly thought out and no support for it was ever implemented. It should not be generated by any future compilers or compiler versions.

THE THREADED-CODE PC-CORRELATION DST RECORD

This DST record is identical in format to the Line Number PC-Correlation DST record except that the record type code is DST$K_LINE_NUM_REL_R11. It was used by an obsolete COBOL compiler according to legend (the memories are a bit hazy by now). The idea was that the threaded code generated by this compiler consisted of a vector of longwords where each longword contained the address of a run-time support routine to call. Register R11 pointed to the beginning of this vector. The code generated for a source line thus consisted of some number of longwords with addresses to call (or perhaps jump to--the exact details are lost in the mists of time). The line number PC-correlation information passed to DEBUG consisted of line numbers correlated with byte-offsets relative to R11 (i.e., to the start of the threaded code). Breakpoints

were placed on a specified line by looking up the corresponding offset relative to R11 and then storing an address within DEBUG into that location. When the location was reached, DEBUG was entered. DEBUG could then convert the "PC", i.e. the threaded-code location, back to a line number to announce the breakpoint. It is not clear how, or even whether, tracing, stepping, and watchpoints were implemented.

The Threaded-Code PC-Correlation DST record is no longer supported by DEBUG and should not be generated by any current or future compilers.

THE COBOL HACK DST RECORD

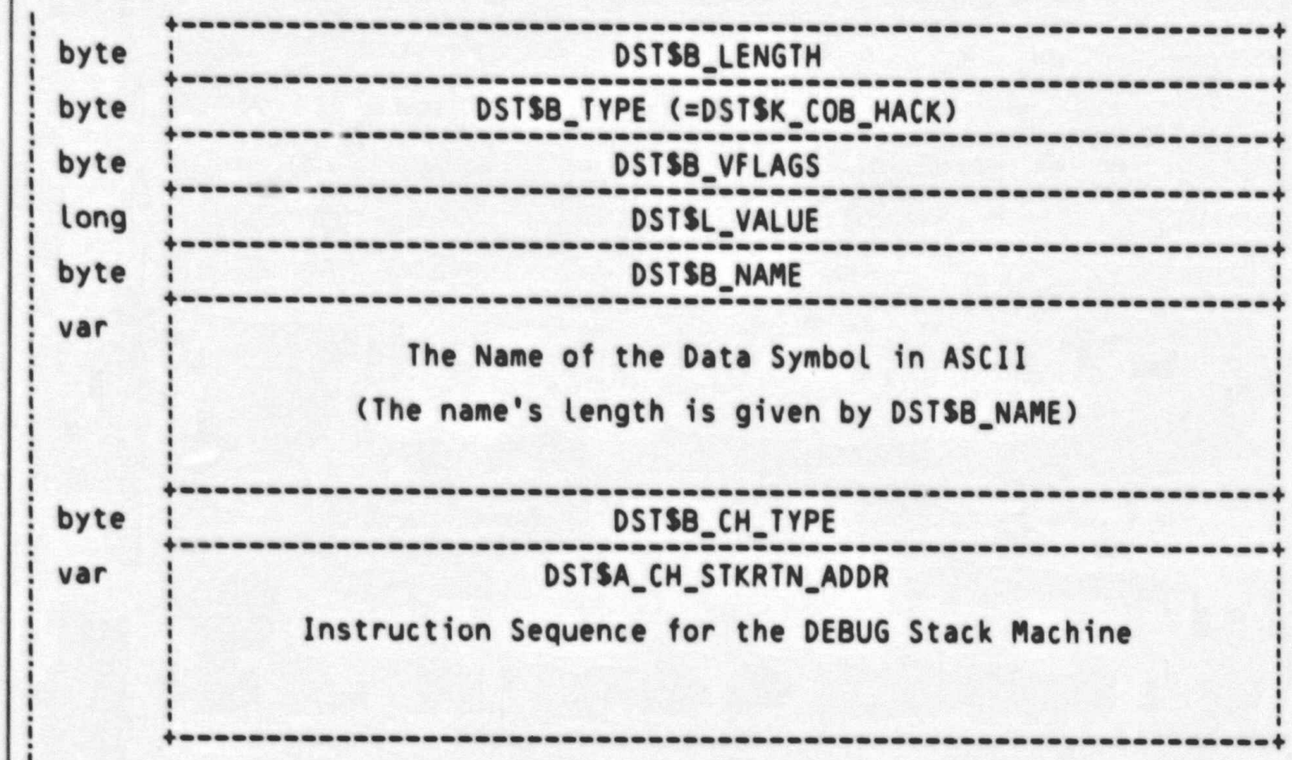
The COBOL Hack DST record was at one time used to support formal arguments to COBOL procedures. It has now been superceded by the more general Value Specification mechanism, and is thus obsolete. It is no longer generated by the COBOL compiler, and it should not be generated by any current or future compilers. Future versions of DEBUG may not support it.

The fields of this record consist of the fields of the Standard Data DST record followed by a type field that specifies the data type and then a sequence of commands for the DEBUG stack machine. (See the section on Value Specifications for details on the DEBUG stack machine.) The result of interpreting the stack machine routine is the address of the object described by this record. The DST$B_VFLAGS and DST$L_VALUE fields are zero unless the object has a descriptor. In this latter case they specify the location of the descriptor. The result of the stack machine routine is placed in the DSC$A_POINTER field of the descriptor before it is used. In addition, if it is an array descriptor, the DSC$A_A0 field is added to the result of the stack machine routine and the result is placed in the DSC$A_A0 field before the descriptor is used.

The type field following the name field contains the VAX Standard Type Code of the object being described here. If the object also has a descriptor, its DSC$B_DTYPE field must agree with this code.

The stack machine commands used in this context are those described in the section entitled "The DEBUG Stack Machine" in the chapter on DST Value Specifications.

This is the format of the COBOL Hack DST record:




```
! Define the fields of the Cobol Hack DST record.  Also define the declaration
! macro for the trailer fields.

FIELD DST$COB_HACK_FIELDS =
        SET
        DST$A_COBHACK_TRLR      = [ 8, A_ ]     ! Location of trailer fields
        TES;

FIELD DST$CH_TRLR_FIELDS =
        SET
        DST$B_CH_TYPE           = [ 0, B_ ],    ! VAX standard data type
        DST$A_CH_STKRTN_ADDR    = [ 1, A_ ]     ! Start of stack routine code
        TES;

MACRO
        DST$CH_TRLR = BLOCK[,BYTE] FIELD(DST$CH_TRLR_FIELDS) %;
```

```
!       VALUE SPECIFICATION DST RECORDS
!
!       The Value Specification DST record contains nothing but a DST Value
!       Specification.  However, there appears to be no use for this record
!       since all DST Value Specifications that are actually used appear in
!       other DST records.  This record was probably designed with some use
!       in mind, but was then abandoned when better ways of addressing the
!       original need were devised.  DEBUG ignores this DST record, and it
!       is believed that no compilers actually generate it.  This DST record
!       should not be generated by any future compilers.
!
!       This is the format of the Value Specification DST record:
!
!
!       +---------------------------------------------------------------+
!  byte |                        DST$B_LENGTH                           |
!       +---------------------------------------------------------------+
!  byte |                DST$B_TYPE (= DST$K_VALSPEC)                   |
!       +---------------------------------------------------------------+
!  var  |                                                               |
!       |                  A DST Value Specification                    |
!       |                                                               |
!       +---------------------------------------------------------------+
!
! Define the fields of the Value Specification DST record.
!
FIELD DST$VALSPEC_FIELDS =
        SET
        DST$A_VS_VALSPEC_ADDR = [ 2, A_ ]       ! The start location of the
                                                !       Value Specification
        TES;
```

```
!
!                D S T   R E C O R D   D E C L A R A T I O N   M A C R O
!
!
!
!       This macro allows BLISS symbols which are declared DST$RECORD or
!       REF DST$RECORD to be qualified by all the field names present in
!       the various DST record formats.  It is anticipated that users will
!       declare separate symbols for field sets which describe trailing
!       fields in DST records; a pointer to the PSECT DST record trailer,
!       for example, would be declared to be a REF DST$PSECT_TRAILER.
!       Separate macros are supplied above for all such trailer fields.
!
! Define the declaration macro for all DST records.
!
MACRO
    DST$RECORD = BLOCK [256,BYTE] FIELD(
        DST$HEADER_FIELDS,
        DST$STD_FIELDS,
        DST$DSC_FIELDS,
        DST$TVS_FIELDS,
        DST$MODBEG_FIELDS,
        DST$RTNBEG_FIELDS,
        DST$RTNEND_FIELDS,
        DST$BLKBEG_FIELDS,
        DST$BLKEND_FIELDS,
        DST$VERSION_FIELDS,
        DST$STATLINK_FIELDS,
        DST$PROLOG_FIELDS,
        DST$BLI_FIELDS,
        DST$BLI_VEC_FIELDS,
        DST$BLI_BITVEC_FIELDS,
        DST$BLI_BLOCK_FIELDS,
        DST$BLI_BLKVEC_FIELDS,
        DST$VARVAL_FIELDS,
        DST$ENUMBEG_FIELDS,
        DST$PSECT_FIELDS,
        DST$LINE_NUM_FIELDS,
        DST$SOURCE_FIELDS,
        DST$DEF_LNUM_FIELDS,
        DST$CONTIN_FIELDS,
        DST$COB_HACK_FIELDS,
        DST$BLIFLD_FIELDS,
        DST$TYPSPEC_FIELDS,
        DST$VALSPEC_FIELDS,
        DST$CH_TRLR_FIELDS,
        DST$OVERLOAD_FIELDS)%;


! END OF DSTRECRDS.REQ.
```

TRACEMSG
MDL

TBKLIB
REQ

TRACE

TRACE
MAP

STRUCDEF
REQ

TBKPROLOG
REQ

TBKDST
REQ

STRUCDEF
LIS