Guide to the VSI Structure Definition Language

Operating System and Version: VSI OpenVMS x86-64 Version 9.2-3 or higher

Software Version: VSI SDL Version 3.8

VMS Software, Inc. (VSI)
Boston, Massachusetts, USA

Guide to the VSI Structure Definition Language

Legal Notice

Table of Contents

Preface

This manual describes the VSI Structure Definition Language (VSI SDL) and the VSI SDL translator for use on OpenVMS operating systems.

VSI SDL source code can be translated to output files in one or more target OpenVMS languages. VSI SDL is suitable for systems and application programming environments that use executable programs consisting of modules written in one or multiple OpenVMS programming languages.

This manual describes support in SDL for both Ada and PL/I output. While these compilers exist on older OpenVMS targets, there are no VSI supported Ada or PL/I compilers on OpenVMS x86-64.

1. Intended Audience

This manual is intended for users who are familiar with one or more programming languages and who are currently involved in the design and development of multilanguage programming applications; however, users are not required to have previous experience with VSI SDL in order to use this manual.

2. Document Structure

This manual contains the following chapters and appendixes.

- *Chapter 1, "Overview"* provides a brief overview of VSI SDL and the translation process.
- *Chapter 2, "Processing a VSI SDL Source File"* describes how to create, edit, and process a VSI SDL source file using the VSI Language-Sensitive Editor (LSE) templates and the SDL command options.
- *Chapter 3, "VSI SDL Language Elements"* describes the VSI SDL language elements that compose VSI SDL declarations.
- *Chapter 4, "VSI SDL Declarations"* describes the function and format of VSI SDL declarations.
- *Appendix A, "VSI SDL Diagnostic Messages"* provides a list and descriptions of VSI SDL diagnostic messages.
- *Appendix B, "VSI SDL Language Translation Summaries"* shows translation summaries for all output languages supported by VSI SDL.

3. Conventions

The following conventions are used in this document.

Convention	Meaning
{ STRUCTURE } { UNION }	Stacked items within braces indicate that you must select one of the items.
[ ]	Simple square brackets indicate that the enclosed item(s) are optional.
[COMMON]	Stacked items within brackets indicate that only one item may be selected.

Convention	Meaning
[GLOBAL]	
MODULE name;	Names shown in uppercase letters in examples and format descriptions are VSI SDL keywords that must be entered as shown. Names and syntactic elements shown in lowercase letters represent user-specified names and identifiers.
arg,...	A comma followed by an ellipsis means that the preceding item may be repeated one or more times, with commas separating two or more items.
. . .	A vertical ellipsis in an example or figure indicates that not all the statements or elements are shown.
common storage	Boldface words in text are used to introduce or define a new term, or to refer to a term used in a code example.
$ LSEDIT USER.SDL	In interactive examples, user input is underlined.

4. VSI Encourages Your Comments

You may send comments or suggestions regarding this manual or any VSI document by sending electronic mail to the following Internet address: <docinfo@vmssoftware.com>. Users who have VSI OpenVMS support contracts through VSI can contact <support@vmssoftware.com> for help with this product.

5. OpenVMS Documentation

The full VSI OpenVMS documentation set can be found on the VMS Software Documentation webpage at https://docs.vmssoftware.com.

Chapter 1. Overview

The VSI Structure Definition Language (VSI SDL) is used to write source statements that describe data structures and that can be translated to source statements in other languages. You can include the resulting output files in a corresponding target language program for subsequent compilation.

Because VSI SDL is compiler – and language-independent, it is particularly useful for maintaining multilanguage implementations. For example, you can create and later modify a single VSI SDL source file that can be translated to multilanguage output files; any number of these output files can then be included in one or several multilanguage programming applications.

VSI SDL supports the following OpenVMS languages:

- Ada
- VSI BASIC
- VSI BLISS
- VSI C/C++
- VSI Datatrieve
- VSI DCL
- VSI Fortran
- VSI MACRO
- VSI Pascal
- PL/I

1.1. The VSI SDL Translation Process

The translation of a VSI SDL source file occurs when you issue the SDL command (see *Section 2.1, "Processing a VSI SDL Source File"*). The SDL command activates the VSI SDL "front-end" translator (SDL$MAIN.EXE), which is stored in the directory SYS$SYSTEM. The front end parses the VSI SDL source code and, if you specify the **/LANGUAGE** qualifier, passes the parse tree to one or more of the VSI SDL "back-end" translators. The following table is a list of the back ends that are stored in the directory SYS$SHARE.

Language	**Back End**
Ada	SDL$ADA.EXE
VSI BASIC	SDL$BASIC.EXE
VSI BLISS	SDL$BLISS.EXE SDL$BLISSF.EXE[1]
VSI C/C++	SDL$CC.EXE
VSI Datatrieve	SDL$DTR.EXE
VSI DCL	SDL$DCL.EXE

Language	Back End
VSI Fortran	SDL$FORTRAN.EXE
VSI MACRO	SDL$MACRO.EXE
VSI Pascal	SDL$PASCAL.EXE
PL/I	SDL$PLI.EXE

[1]The BLISSF back end generates VSI BLISS FIELDSETS for VSI SDL aggregates.

Each back end translates only those VSI SDL declarations that can (or need) be expressed in that language. VSI SDL declarations, described in detail in *Chapter 4, "VSI SDL Declarations"*, translate to the following types of data items:

- Scalar or dimensioned scalar data items (ITEM declarations)
- Nonscalar data items (AGGREGATE and subaggregate declarations)
- Named constants (CONSTANT declarations)
- External entries (ENTRY declarations)

There are 4 ways to use SDL:

1. The front end (SDL$MAIN.EXE) parses the statements contained in the VSI SDL source file TEST.SDL as a result of issuing the SDL command in one of the following four ways:

   a. `$ SDL/LIST/PARSE/LANGUAGE=BAS TEST.SDL`

      If you specify the **/LANGUAGE** qualifier (for example, **/LANGUAGE=BAS**), the front end parses the source code and passes the parse tree to the back end for the specified language (for example, SDL$BASIC.EXE). (The function of the **/LIST** qualifier is described at the end of step 1. The **/PARSE** qualifier is shown here for completeness, but it is the default for this command and does not need to be specified.)

   b. `$ SDL/PARSE=TEST/LIST TEST.SDL`

      If you do not specify the **/LANGUAGE** qualifier, but specify instead the **/PARSE** qualifier and an output file name, the front end parses the source code and writes the parse tree to an intermediate file with a default file type of .SDI.

   c. `$ SDL/PARSE=TEST/LIST/LANGUAGE=BAS TEST.SDL`

      If you specify the **/PARSE** qualifier with an output file name and the **/LANGUAGE** qualifier, the front end parses the source code, writes the parse tree to the specified intermediate file (for example, TEST), and passes the parse tree to the specified back end.

   d. `$ SDL/PARSE/LIST TEST.SDL`

      If you specify the SDL command with the optional **/PARSE** and **/LIST** qualifiers and do not specify an intermediate file name, the front end parses the source code (checking for errors), but does not generate an intermediate file.

   The front end produces a .LIS file as a result of specifying the **/LIST** qualifier on all the SDL commands described in steps 1a. through 1d. The .LIS file contains a line-numbered copy of your source code and describes any errors or warnings encountered during the translation process.

2. If you specify the **/PARSE** qualifier and an output file name, as in step 1b. or step 1c., you may later decide to translate the intermediate (already parsed) code to one or more language output files (for example, TEST.BAS) by entering the following command:

```
$ SDL/NOPARSE/LANGUAGE=BAS TEST.SDI
```

> **Note**
>
> In cases where VSI SDL translation must be performed at a customer site, the intermediate .SDI file is included in the software VSI sends to customers.

3. If you enter the command in step 2, the alternate front end (SDL$NPARSE.EXE) passes the parse tree (in TEST.SDI) to the specified back end (SDL$BASIC.EXE).

4. The specified back end (SDL$BASIC.EXE) produces an output file with the default file type of .BAS.

You can specify any or all of the language options on the **/LANGUAGE** qualifier, and VSI SDL produces separate output files for each language specified.

1.2. Description of a Sample VSI SDL Source File

Example 1.1, "Sample VSI SDL Source File" is a typical VSI SDL source file, and the key following the example describes each of the numbered language elements. (*Chapter 3, "VSI SDL Language Elements"* describes all the language elements in detail.)

Example 1.1. Sample VSI SDL Source File

```
MODULE opr_descriptor IDENT "Version 2.0";❶
/* define constants and node structure for operators❷
#max_args = 10;❸
CONSTANT (fixed_binary,floating,char,untyped) EQUALS 1 INCREMENT 1;❹
AGGREGATE operator STRUCTURE ❺
    PREFIX "opr_";❻
    flink ADDRESS;❼
    blink ADDRESS;
    opcount WORD;
    optype CHARACTER LENGTH 1;
    id WORD;
    operands LONGWORD DIMENSION 0:#max_args-1;❽
END operator;
#opsize = .;❾
CONSTANT opr_node_size EQUALS #opsize / 2;
ITEM current_node_ptr ❿ ADDRESS ⓫ GLOBAL;⓬
END_MODULE opr_descriptor;⓭
```

Key to Example 1–1:

❶ All VSI SDL declarations are grouped within modules, and you must assign a name to each module. The IDENT keyword precedes any commented information that you may want to add to describe the MODULE declaration.

❷ Output comments begin with a slash and an asterisk (/*) and are written to the language output file unless the **/NOCOMMENTS** qualifier is specified.

❸ Local symbols begin with a pound sign (#) and are not written to the output file. Local symbols may be used to express values in declarations. For example, the symbol `#max_args` is used in the declaration of the array `operands`.

❹ CONSTANT declarations produce declarations of named constants. When the INCREMENT option and an increment value are specified, VSI SDL automatically increments the initial value for each of the declared output constants. In the example, `fixed_binary` will be assigned the value 1, `floating` will be assigned the value 2, and so on.

❺ AGGREGATE declarations define data structures and their members.

❻ When the PREFIX option and a prefix are specified, VSI SDL concatenates the prefix and a data type code to the declared aggregate member names in the language output files. Compare these aggregate member name declarations with the C output shown in *Example 1.2, "C Output File for the Sample VSI SDL Source File"*.

❼ Aggregate members are declared using reserved VSI SDL data type keywords.

❽ An aggregate member or a scalar data item can be declared to be an array by specifying the DIMENSION option. In this example, the array `operands` has 10 elements, with subscripts 0 through 9.

❽ The period (.) represents the current byte offset within an AGGREGATE declaration. The local symbol assignment `#opsize = .;` captures the size of the constant portion of the structure `operator`. The value of this local symbol is then used in the declaration of the constant `opr_node_size`.

❾ ITEM declarations, such as the declaration of `current_node_ptr`, define scalar data items.

❿ The ADDRESS keyword specifies a data type that is an address, or pointer.

⓫ The GLOBAL keyword specifies global storage to override the default language storage class for an ITEM or an AGGREGATE declaration.

⓬ The END_MODULE keyword ends a MODULE declaration; you may optionally specify the MODULE name after the END_MODULE keyword.

Example 1.2, "C Output File for the Sample VSI SDL Source File" shows the C output file that results from translation of the VSI SDL source file shown in *Chapter 3, "VSI SDL Language Elements"*.

Chapter 3, "VSI SDL Language Elements" describes all the VSI SDL language elements, and *Chapter 4, "VSI SDL Declarations"* describes the function and format of each of the VSI SDL declarations.

Example 1.2. C Output File for the Sample VSI SDL Source File

```
/
****************************************************************************************
/* Created: 29-Apr-2026 08:34:02 by OpenVMS SDL V3.8-1     */
/* Source:  29-APR-2026 08:33:09 EXAMPLE_1P1.SDL;2 */
/
****************************************************************************************
/*** MODULE opr_descriptor IDENT Version 2.0 ***/
#pragma __member_alignment __save
#pragma __nomember_alignment
/* define constants and node structure for operators
 */
```

```
#define fixed_binary 1
#define floating 2
#define char 3
#define untyped 4
struct operator {
    int *opr_a_flink;
    int *opr_a_blink;
    short int opr_w_opcount;
    char opr_t_optype;
    short int opr_w_id;
    int opr_l_operands [10];
    } ;
#define opr_node_size 26

#pragma __extern_model __save
#pragma __extern_model __strict_refdef
extern int *current_node_ptr;
#pragma __extern_model __restore


#pragma __member_alignment __restore
```

Chapter 2. Processing a VSI SDL Source File

This chapter describes how to process your source file using the SDL command and all its qualifiers.

You can use any text editor to create your source file, including LSE, EDT and TPU/EVE. Prior LSE kits included SDL templates and placeholders to assist in writing SDL source files. Those templates are not provided in the current kit but they will return in the future.

2.1. Processing a VSI SDL Source File

The SDL command invokes the VSI SDL translator from DCL command level to produce output files for one or more target languages. The SDL command has the following format:

```
SDL[/qualifier[...]] file-spec[/qualifier[...]],...
```

Command Parameter

file-spec,...

Specifies one or more VSI SDL source files to be translated. A file specification must specify a file name; if it does not include a file type, VSI SDL uses the default file type .SDL. You can specify multiple input files, separated by commas. VSI SDL translates each source file individually and creates separate output files for each.

Wildcards are not allowed in the file specification.

Command Qualifiers

Command qualifiers may be specified following the SDL command, or they may be used to qualify individual file specifications. *Table 2.1, "SDL Command Qualifiers and Their Defaults"* lists all the optional SDL command qualifiers and their defaults.

Table 2.1. SDL Command Qualifiers and Their Defaults

Qualifier	Default
/ALIGNMENT	No alignment
/ALPHA_AXP	/ALPHA_AXP on all VSI systems
/[NO]B64	/NOB64
/ASM_DEVELOPMENT	/NOASM_DEVELOPMENT
/[NO]CHECK_ALIGNMENT	/NOCHECK_ALIGNMENT
/[NO]COMMENTS	/COMMENTS
/[NO]COPYRIGHT	/NOCOPYRIGHT
/[NO]C_DEVELOPMENT	/NOC_DEVELOPMENT
/[NO]DUMP[=file-spec]	/NODUMP

Qualifier	Default
/[NO]GLOBAL_DEFINITION	/NOGLOBAL_DEFINITION
/[NO]HEADER	/HEADER
/LANGUAGES=(language[=file-spec],...)	No languages
/[NO]LIST	/NOLIST (interactive) /LIST (batch)
/[NO]MEMBER_ALIGN	/NOMEMBER_ALIGN
/[NO]MODULE	/MODULE
/[NO]PARSE[=file-spec]	/PARSE
[/NO]PLI_DEVELOPMENT	/NOPLI_DEVELOPMENT
/[NO]SUBFIELDS	/NOSUBFIELDS
/[NO]SUPPRESS	/NOSUBFIELDS
/SYMBOLS	No symbols
/VAX	/VAX on VAX systems
/[NO]VMS_DEVELOPMENT	/NOVMS_DEVELOPMENT

/ALIGNMENT=value

The assumed alignment. Integer value greater than zero. If specified, diagnostic messages are emitted for data items that do not fall on the assumed alignment.

/ALPHA_AXP

Note the following:

- The size of certain data types (HARDWARE_ADDRESS, INTEGER_HW, HARDWARE_ INTEGER, POINTER_HW) is 8 bytes if **/ALPHA_AXP** is specified and 4 bytes otherwise.
- BITFIELDs can have 64 bits if **/ALPHA_AXP** is specified and only 32 bits otherwise.
- BASIC defines the data type HUGE as BASIC$HFLOAT_AXP if **/ALPHA_AXP** is specified and as HFLOAT otherwise and HUGE_COMPLEX as BASIC$H_FLOATING_COMPLEX_ AXP if **/ALPHA_AXP** is specified and as BASIC$H_FLOATING_COMPLEX otherwise.
- CC writes alignment pragmas to the output file only if **/ALPHA_AXP** is specified.
- CC generates 64-bit pointers only if **/ALPHA_AXP** is specified.
- CC generates QUADWORD data types as _ _int64 if **/ALPHA_AXP** is specified and as int[2] otherwise.

/[NO]B64

This qualifier is only valid for the languages BLISS and BLISSF.

The default extension for the output file is .R64 if **/B64** is specified and .R32 otherwise.

The word size used in BITFIELDs is 64 bits if **/B64** is specified and 32 bits otherwise.

The name used for conditional compilation changes from BLISS or BLISSF to BLISS64 or BLISSF64, respectively.

You cannot specify the qualifiers **/B64** and **/VAX** together.

/[NO]CHECK_ALIGNMENT

If specified, diagnostic messages are emitted for data items that do not fall on their natural alignment.

/[NO]COMMENTS

Controls whether output comments are included in the output file. For more compact target language representation, use the **/NOCOMMENTS** qualifier to save file space. The default is **/COMMENTS**.

/[NO]COPYRIGHT

Controls whether a standard VSI copyright header is produced in the output file. The **/COPYRIGHT** qualifier causes the VSI SDL translator to precede the output with a comment containing the standard copyright claim. The default is **/NOCOPYRIGHT**.

/[NO]C_DEVELOPMENT

Only used by the CC backend.

If an input file contains more than one module, C generates one .h file per module if either **/C_DEVELOPMENT** or **/VMS_DEVELOPMENT** is specified and one file containing all the modules otherwise.

C defines a macro with all lowercase characters to equal the same name in all uppercase characters for every entry node if either **/C_DEVELOPMENT** or **/VMS_DEVELOPMENT** is specified.

C creates function prototypes if either **/C_DEVELOPMENT** or **/VMS_DEVELOPMENT** is specified (although the ones generated with **/VMS_DEVELOPMENT** only contain `__unknown_ params`).

C generates most definitions twice if **/C_DEVELOPMENT** is specified, separated with `#ifdef __NEW_STARLET" ... "else" ... "endif"`. The `__NEW_STARLET` definitions contain complete function prototypes, the OLD definitions only `__unknown_params`. Also the definitions of structs and unions are different.

The `__member_alignment pragmas` are only generated if both **/ALPHA_AXP** and **/C_ DEVELOPMENT** are specified.

C generates `if !defined(__VAXC)` for special cases if **/C_DEVELOPMENT** or **/VMS_DEVELOPMENT** is specified, otherwise `ifdef __cplusplus` is generated.

C generates `#ifndef __<module-name>_LOADED ...` if **/C_DEVELOPMENT** or **/VMS_DEVELOPMENT** is specified.

C generates `__required_pointer_size` pragmas if either **/VMS_DEVELOPMENT** or both **/C_DEVELOPMENT** and **/ALPHA_AXP** are specified.

Together with the **/ALPHA_AXP** qualifier and the **/VMS_DEVELOPMENT** qualifier this qualifier influences the definition of certain data types and the generation of certain pragmas, e.g. the data type QUADWORD is defined as __int64 if **/ALPHA_AXP** is specified and **/VMS_DEVELOPMENT** and not **/C_DEVELOPMENT**, and as int [2] otherwise, and the HARDWARE_ADDRESS

and POINTER_HW data types are defined as __int64 if **/ALPHA_ AXP** and neither **/C_DEVELOPMENT** nor **/VMS_DEVELOPMENT** are specified and as int[2] if **/ALPHA_AXP** is not specified, and are not defined if **/ALPHA_AXP** and one of the qualifiers **/C_DEVELOPMENT** or **/VMS_DEVELOPMENT** is specified.

/[NO]DUMP [=file-spec]

Controls whether the VSI SDL intermediate code is displayed on your screen. If you specify the **/DUMP** qualifier with a file specification, the output is sent to the specified file. The default is **/NODUMP**.

/[NO]GLOBAL_DEFINITION

Controls whether an item or aggregate declared with the GLOBAL option generates a declaration indicating that the value of the global data item is defined in this module. In some languages, this qualifier has no effect; see the individual language translation summaries in *Appendix B, "VSI SDL Language Translation Summaries"* and online examples of language output files as a result of processing the VSI SDL source file EXAMPLE.SDL in SDL$EXAMPLES.

By default, global definitions are not generated; the GLOBAL option designates global data whose value is defined elsewhere.

/[NO]HEADER

Controls whether a header containing the date and the source file name is included at the beginning of the output file. The default is **/HEADER**.

/LANGUAGES={language[=file-spec]}
{(language[=file-spec],...)}

Specifies one or more of the language options listed in *Table 2.2, "VSI SDL Output Language Options and File Types"* for which the VSI SDL translator is to produce one or more source output files. By default, VSI SDL writes output files into separate files in the current default directory. The default file name for each output file is taken from the file name of the corresponding source file and the default target file type for each language name.

Table 2.2. VSI SDL Output Language Options and File Types

Language	Option	Target File Type
Ada	/LANGUAGES=ADA	.ADA
ASM (x86-64 assembler)	/LANGUAGES=ASM	.S
VSI BASIC	/LANGUAGES=BASIC	.BAS
VSI BLISS	/LANGUAGES=BLISS	.R32, .R64
	/LANGUAGES=BLISSF	.R32, .R64[1]
	/LANGUAGES=BLISS64	R64[b]
VSI C/C++	/LANGUAGES=CC	.H
VSI DCL	/LANGUAGES=DCL	.COM
VSI Datatrieve	/LANGUAGES=DTR	.DTR
VSI Fortran	/LANGUAGES=FORTRAN	.FOR

Language	Option	Target File Type
VSI MACRO	/LANGUAGES=MACRO	.MAR
VSI Pascal	/LANGUAGES=PASCAL	.PAS
PL/I	/LANGUAGES=PLI	.PLI
SDML	/LANGUAGES=SMDL	.SDML
DECTPU	/LANGUAGES=TPU	.TPU
UIL	/LANGUAGES=UIL	.UIL

[1]The BLISSF back end generates VSI BLISS FIELDSETS for VSI SDL aggregates.

[b]The BLISS64 back end generates the same code as the BLISS backend when the /B64 qualifier has been specified.

VSI SDL builds the language image name by concatenating the prefix SDL to the language option specified on the SDL command. For example, if you specify **/LANGUAGE=FORTRAN**, VSI SDL searches for and activates the SDL$FORTRAN.EXE back end. You may abbreviate most language options, but VSI SDL uses the first image it finds that matches the generated file specification. For example, if you wish to invoke the BLISSF back end instead of the VSI BLISS back end, you must specify **/LANGUAGE=BLISSF**.

The **/LANGUAGE** qualifier also allows you to override the default output file specification for one or more language output files. You can specify a language option followed by the destination file specification for that language. You must separate the language from the destination file specification with an equal sign (=) and separate specifications for different languages with commas.

/[NO]LIST [=file-spec]

Controls whether a listing file is produced.

If the SDL command is executed from interactive mode, **/NOLIST** is the default. If the SDL command is executed from batch mode, **/LIST** is the default.

The **/LIST** qualifier causes the VSI SDL translator to produce a listing file with numbered lines of source code and descriptions of any compilation errors. The listing file has the same name as the related source file and a file type of .LIS.

/[NO]MEMBER_ALIGN

Specifies that every item in aggregates should be aligned. This is the same as specifying ALIGN on all aggregates.

/[NO]MODULE

Controls whether an Ada package or a VSI Pascal module is generated in the output file. The default is **/MODULE**. The **/[NO]MODULE** qualifier affects only the VSI Pascal and Ada languages.

The VSI Pascal backend generates "%include"-able files if **/NOMODULE** is specified. Otherwise, the generated files are MODULEs which can be precompiled and used with the [INHERIT] source attribute.

/[NO]PARSE [=file-spec]

Controls whether VSI SDL reads or writes a VSI SDL intermediate file. A VSI SDL intermediate file contains VSI SDL source code that has already been parsed using the **SDL/PARSE** command. The default file type for the VSI SDL intermediate file is .SDI.

The intermediate file produced as a result of the **SDL/PARSE** command can later be used instead of VSI SDL source code as input to the back end. This is done using the **SDL/NOPARSE** command. When this qualifier appears in an SDL command line, the input file specification is assumed to be that of an SDL intermediate file with a default file type of .SDI. Examples 4 and 5 at the end of this section show the uses of the **/[NO]PARSE** qualifiers.

/[NO]PLI_DEVELOPMENT

Only used by the PLI backend.

This qualifier specifies the definition of certain data types, e.g. most integral data types are defined as fixed binary (n) if **/PLI_DEVELOPMENT** is specified and as bit(n) aligned otherwise.

In addition, the type names of certain parameters are changed, e.g. ASTADR to *entry value* and MASK_BYTE to bit (8) aligned.

Default parameters that are not optional are marked as optional if **/PLI_DEVELOPMENT** is specified

/[NO]SUBFIELDS

When this qualifier is given, the BLISSF backend handles subfields of members declared with a named type differently.

/SUPPRESS ={suppress-option}
{(suppress-option,...)}

The **/SUPPRESS** qualifier has the following format:

```
SDL /SUPPRESS=(PREFIXES, TAGS)
```

Note the following:

- Output in all languages in that compilation is affected. It is not possible to make the qualifier position-dependent (CDU constraint).
- Either PREFIXES or TAGS, or both, may be included in the list.
- If both prefixes and tags are suppressed, the connecting underscore is also suppressed.

/SYMBOLS={symbol=value}
{(symbol=value,...)}

It is possible to specify symbols and values which can be used in the IFSYMBOL statement (used with conditional compilation).

See IFSYMBOL.

/VAX

For modules, Fortran writes the following comment to the output file:

```
!DEC$ OPTIONS/ALIGN=(RECORDS=PACKED,COMMONS=PACKED)/NOWARN
```

CC generates double as return type for functions if the actual return type is quadword and **/VAX** but not **/VMS_DEVELOPMENT** is specified.

You cannot specify the qualifiers **/ALPHA_AXP** and **/VAX** together. You cannot specify the qualifiers **/B64** and **/VAX** together.

/[NO]VMS_DEVELOPMENT

Modifies the behavior of certain VSI SDL back ends so that the generated output files conform to specific VMS development standards.

Ada defines the data types QUADWORD and INTEGER_QUAD as INTEGER_64 if **/ALPHA_AXP** and **/VMS_DEVELOPMENT** is specified and as UNSIGEND_QUADWORD otherwise.

For BLISS and BLISSF, the **/VMS_DEVELOPMENT** qualifier causes ENTRY declarations to generate KEYWORD macros in the output (.R32) file and defines routines as `external routine` *`routine-name`* `: novalue` otherwise.

BLISS' and BLISSF's generation of certain literals for the sizes of structures or unions depends on this qualifier.

The following rules apply when the **/VMS_DEVELOPMENT** qualifier is specified for VSI BLISSF output:

- User fill fields are ignored.
- Nested structures are ignored.
- Macros are generated instead of fields for certain level one items.

CC handles user fill depending on **/VMS_DEVELOPMENT**.

CC handles references to aggregates within typedef'd aggregates different when **/VMS_DEVELOPMENT** is specified.

For MACRO, the **/VMS_DEVELOPMENT** qualifier causes the VMS macros $EQU, $DEF, $DEFINI, and $DEFEND to be generated in the output (.MAR) file.

For Pascal, the **/VMS_DEVELOPMENT** qualifier causes $TYPE to be appended to the names of structures and unions in the output (.PAS) file.

The following rules apply when the **/VMS_DEVELOPMENT** qualifier is specified for Pascal output:

- read node modules are prefixed with PASCAL$.
- The generation of certain fill fields is suppressed.
- $TYPE is appended to data types that have no `$` in their name.

  An error message is emitted if a complex data type is encountered.
- The type $DEFPTR is generated for user data types instead of the actual data type.

The following rules apply when the **/VMS_DEVELOPMENT** qualifier is specified for PL/I output:

- UNSIGNED BYTE, UNSIGNED WORD, and UNSIGNED LONGWORD yield FIXED

  BINARY(7), FIXED BINARY(15), and FIXED BINARY(31), respectively.

- Special VMS TYPENAME values are recognized.
- DEFAULT n for n <> 0 generates OPTIONAL.

The default is **/NOVMS_DEVELOPMENT**.

Example 2.1.

The following are examples and descriptions of the SDL command.

1. `$ SDL BLOCKNODE/LANGUAGE=(CC=C$:[C.CSRC])`

   VSI SDL translates the declarations in BLOCKNODE.SDL to C and writes the VSI C output to C$:[C.CSRC]BLOCKNODE.H.

2. `$ SDL/LANGUAGE=(MACRO,BLISS) IODEF,SSDEF`

   VSI SDL translates the declarations in IODEF.SDL and SSDEF.SDL and writes the output to IODEF.MAR, IODEF.R32, SSDEF.MAR, and SSDEF.R32.

3. `$ SDL/PARSE=INTER TEST`

   VSI SDL translates the declarations in the source file TEST.SDL and writes the output to a VSI SDL intermediate file called INTER.SDI.

4. `$ SDL/NOPARSE/LANG=(MACRO,BLISS) INTER`

   VSI SDL translates the declarations that have already been parsed by the VSI SDL front end in INTER.SDI and writes the output to INTER.MAR and INTER.R32.

Chapter 3. VSI SDL Language Elements

This chapter describes the function and syntax of the following VSI SDL language elements that compose the VSI SDL declarations described in *Chapter 4, "VSI SDL Declarations"*:

- User-specified names, which can be either local symbol names or source program identifiers
- Reserved VSI SDL keywords
- Expressions

The following can also be used within a MODULE declaration; although they are not VSI SDL language elements:

- Local and output comments
- INCLUDE statement
- READ statement
- Conditional compilation
- Text pass-through
- DECLARE statement

The space, tab, or carriage return character delimits the language elements, and a semicolon (;) terminates each declaration. In MODULE and AGGREGATE declarations, the semicolon also terminates separate parts of the declaration.

Appendix B, "VSI SDL Language Translation Summaries" shows language translation summaries of all the VSI SDL language elements.

3.1. Names

A VSI SDL name can be either a user-specified local symbol name that is not translated to the output file or a user-specified source program identifier that is translated to the output file. Names are composed of upper- and lowercase letters (A - Z, a - z), numbers (0 - 9), the dollar sign ($), and the underscore (_). Specifying a name is subject to the following rules:

1. Local symbol names must begin with a pound sign (#).
2. Source program identifiers must begin with an alphabetic character (A - Z, a - z), a dollar sign ($), or an underscore (_).
3. Source program identifiers that are reserved VSI SDL keywords or that contain invalid VSI

   SDL characters must be enclosed in quotation marks (" "). (For more information on reserved VSI SDL keywords, see *Section 3.2, "Keywords"*)

4. VSI SDL passes all source program identifiers to the output file in the same case in which they are defined.

3.1.1. Local Symbol Names

A local symbol name is known only within a VSI SDL source file and cannot be translated directly to the output file. A local symbol name can be assigned a value anywhere within a source file, but must begin with the pound sign (#). A local symbol is declared when it is first assigned a value. This value can be any valid expression (see *Section 3.3, "Expressions"*). If you reference a local symbol before assigning it a value, VSI SDL displays an error message and does not produce an output file. A local symbol assignment has the following syntax:

```
#local-name = expression;
```

#local-name

Is any valid VSI SDL name.

expression

Is any valid VSI SDL expression resulting in a longword integer value.

Signed integer longword data types are described in *Section 3.2.3.9, "Integer Data Types"*

The following are examples of local symbol assignments:

```
#max_args = 255;
#counter = #counter + 1;
```

The values of these local symbols may be referenced by subsequent declarations, as shown in the following example:

```
CONSTANT block_node_size EQUALS #max_args + #counter;
```

3.1.2. Source Program Identifiers

Source program identifiers (identifiers) specify declaration names, AGGREGATE member names, and ENTRY parameter names that are passed to the output file. Optional user-specified prefixes and tags can be appended to these identifiers. If a prefix is specified without a tag, VSI SDL concatenates a default tag (corresponding to the data type) to the identifier in the output file. (*Section 3.2.2.2, "PREFIX, MARKER, and TAG Keywords"* describes the PREFIX and TAG keyword options.) The identifier block_node_size in the following example names the CONSTANT declaration:

```
CONSTANT block_node_size EQUALS #max_args + #counter;
```

To avoid compilation errors, each reference to a particular VSI SDL identifier must be a case-sensitive match because an identifier is passed to the output file in the same case in which it appears in the source file. You can use reserved VSI SDL keywords and characters that are not valid in identifiers if you enclose them in quotation marks (" "). For example:

```
ITEM "length" LONGWORD;
```

This declaration produces the identifier length, which is a reserved VSI SDL keyword typically used to specify the length of a bit-string or character-string data type (see *Section 3.2.3.3, "BITFIELD Data Type"* and *Section 3.2.3.5, "CHARACTER Data Type"*).

3.2. Keywords

Reserved VSI SDL keywords are used to specify the following:

- Declarations
- Declaration modifiers
- Prefixes, markers, and tags
- Alignment
- Storage classes
- Arrays
- Data types

You can use keywords as identifiers if they are enclosed in quotation marks (" "). For example:

```
ITEM "length" LONGWORD;
```

This declaration produces the identifier length, which names this particular ITEM declaration. The length identifier is a reserved VSI SDL keyword that is typically used to specify the length of a bit-string or character-string data type (see *Section 3.2.3.3, "BITFIELD Data Type"* and *Section 3.2.3.5, "CHARACTER Data Type"*).

Reserved VSI SDL keywords can be entered in either upper- or lowercase letters, but they cannot be truncated.

The following sections describe the format and function of each of the reserved VSI SDL keywords.

3.2.1. Declaration Keywords

Table 3.1, "Keywords That Identify or End Declarations" alphabetically lists and defines the keywords for the declarations described in detail in *Chapter 4, "VSI SDL Declarations"*.

Table 3.1. Keywords That Identify or End Declarations

Keyword	Definition
AGGREGATE	Declaration that produces a structure or union body
CONSTANT	Declaration of a named constant
END	Delimiter for the end of an aggregate body
END_MODULE	Delimiter for the end of a module
ENTRY	Declaration of an entry
ITEM	Declaration of an item
MODULE	Declaration of a module
STRUCTURE	Declaration that is a type of aggregate or subaggregate
UNION	Declaration that is a type of aggregate or subaggregate

3.2.2. Declaration Modifier Keywords

Table 3.2, "Keywords Used in Declarations" lists and defines other keywords that are used in declarations. The prefix and tag, storage class, and array keywords have special functions that are described in detail in the sections following *Table 3.2, "Keywords Used in Declarations"*. All the other keywords defined in *Table 3.2, "Keywords Used in Declarations"* are described in greater detail in *Chapter 4, "VSI SDL Declarations"*.

Table 3.2. Keywords Used in Declarations

MODULE Declaration	
Keyword	**Description**
IDENT	Optional keyword used to pass information describing the MODULE declaration to the output file
ITEM Declaration	
Keyword	**Description**
ALIGN, NOALIGN,BASEALIGN	Optional keywords used to specify alignment; see *Section 3.2.2.3, "Alignment Keywords"*
COMMON and GLOBAL	Optional keywords used to specify common and global storage; see *Section 3.2.2.4, "Storage Class Keywords"*
DIMENSION	Optional keyword used to specify that the ITEM declaration is an array; see *Section 3.2.2.7, "DIMENSION Keyword"*
PREFIX	Optional keyword used to concatenate a user-defined prefix to ITEM names, AGGREGATE member names, and named constants; see *Section 3.2.2.2, "PREFIX, MARKER, and TAG Keywords"*
TAG	Optional keyword used to override the default OpenVMS code assigned to a name and to assign a user-defined tag instead; see *Section 3.2.2.2, "PREFIX, MARKER, and TAG Keywords"*
AGGREGATE Declaration	
Keyword	**Description**
COMMON, GLOBAL, BASED, TYPEDEF	Optional keywords used to specify common, global, or based storage, or a TYPEDEF; see *Section 3.2.2.4, "Storage Class Keywords"*
DIMENSION	Optional keyword used to specify that the AGGREGATE declaration is an array; see *Section 3.2.2.7, "DIMENSION Keyword"*
FILL	Optional keyword used to indicate whether the associated aggregate or member occurs only as a fill to force byte alignment on the following member or aggregate, respectively
MARKER	Optional keyword used to assign a user-defined prefix to the aggregate name; see *Section 3.2.2.2, "PREFIX, MARKER, and TAG Keywords"*
ORIGIN	Optional keyword used to define the beginning of an aggregate with respect to an aggregate member
PREFIX	Optional keyword used to concatenate a user-defined prefix to AGGREGATE member names, ITEM names, and named constants; see *Section 3.2.2.2, "PREFIX, MARKER, and TAG Keywords"*

TAG	Optional keyword used to override the default OpenVMS code assigned to a name and to assign a user-defined tag instead; see *Section 3.2.2.2, "PREFIX, MARKER, and TAG Keywords"*
CONSTANT Declaration	
Keyword	**Description**
COUNTER	Optional keyword that saves the last assigned value in a local symbol declaration for subsequent use
EQUALS	Required keyword used in assigning the value to the first named constant
STRING	Optional keyword specified immediately after EQUALS to indicate the definition of a string constant.
INCREMENT	Optional keyword used to specify constants with incremental values
PREFIX	Optional keyword used to concatenate a user-defined prefix to aggregate member names and named constants; see *Section 3.2.2.2, "PREFIX, MARKER, and TAG Keywords"*
TAG	Optional keyword used to override the default OpenVMS code assigned to a name and to assign a user-defined tag instead; see *Section 3.2.2.2, "PREFIX, MARKER, and TAG Keywords"*
TYPENAME	Optional keyword used only by the Ada and PL/I back ends to specify a user-defined data type name; see *Section 3.2.2.1, "User-Specified TYPENAME keyword"*
ENTRY Declaration	
Parameter-Passing Mechanism Keywords	**Description**
DESCRIPTOR	Optional parameter-passing mechanism keyword used to specify that a parameter must be passed BY DESCRIPTOR
REFERENCE	Optional parameter-passing mechanism keyword used to specify that a parameter must be passed BY REFERENCE; REFERENCE is the default
RTL_STR_DESC	Optional parameter-passing mechanism keyword used to specify that a parameter must be passed by any of the classes of string descriptors
VALUE	Optional parameter-passing mechanism keyword used to specify that the parameter must be passed BY immediate VALUE
Parameter Mode Keywords	**Description**
IN	Parameter description keyword used to indicate that a parameter is an input parameter; can be used with OUT to indicate that the parameter is both an input and an output parameter; IN is the default
OUT	Parameter description keyword used to indicate that a parameter is an output parameter; can be used with IN to indicate that the parameter is both an input and an output parameter
Other Parameter Modifier Keywords	**Description**

DEFAULT	Optional parameter description keyword used to specify a default parameter value
DIMENSION	Optional keyword used to specify that the parameter is an array; see *Section 3.2.2.7, "DIMENSION Keyword"*
LIST	Optional parameter description keyword used to indicate that the routine may be called with one or more parameters of the type being described
NAMED	Optional parameter description keyword used only by the Ada back end to name the parameter
OPTIONAL	Optional parameter description keyword used to specify that the parameter may or may not appear in the sequence of parameters using the entry point name
TYPENAME	Optional parameter description keyword used only by the Ada and PL/I back ends to specify a user-defined data type name; see *Section 3.2.2.1, "User-Specified TYPENAME keyword"*
Entry Return Value Keywords	**Description**
NAMED	Optional keyword used to specify the name of the parameter (in a Ada IMPORT_VALUED_PROCEDURE) into which the return value is returned
RETURNS	Optional keyword used to specify the data type returned by the external entry
TYPENAME	Optional keyword used only by the Ada and PL/I back ends to specify a user-defined name that is the data type returned by the external entry; see *Section 3.2.2.1, "User-Specified TYPENAME keyword"*
Entry Description Keywords	**Description**
ALIAS	Optional keyword used to indicate an alternate internal name that can be used to designate the entry point
LINKAGE	Optional keyword used only by the VSI MACRO back end to indicate that a special call macro will be used in the expansion of the entry macro
PARAMETER	Optional keyword used to describe the parameters of the external entry
VARIABLE	Optional keyword used to indicate that the entry point can be invoked with a variable number of parameters; see also the LIST parameter modifier keyword

3.2.2.1. User-Specified TYPENAME keyword

The TYPENAME keyword is used to specify a data type name that is not a VSI SDL data type. Depending on which back end is specified, this name may or may not override the VSI SDL data type. The Ada and PL/I back ends use these data type names as parameter data types, as return value data types, and as CONSTANT declaration data types. The TYPENAME keyword has the following syntax:

```
TYPENAME name
```

The Ada language translation (as a result of processing the VSI SDL source file EXAMPLE.SDL in SDL $EXAMPLES) shows an example of the TYPENAME keyword. The following is an example of the TYPENAME keyword on each of the parameters in an ENTRY declaration:

```
ENTRY SYS$FAO
      ALIAS $FAO
      PARAMETER (CHARACTER DESCRIPTOR NAMED CTRSTR IN TYPENAME CHARDESC,
             WORD UNSIGNED NAMED OUTLEN OUT DEFAULT 0 TYPENAME NUMBER,
             CHARACTER DESCRIPTOR NAMED OUTBUF OUT TYPENAME CHARDESC,
             LONGWORD VALUE NAMED P1 OPTIONAL LIST TYPENAME VARIES)
      RETURNS LONGWORD TYPENAME CONDVALU;
```

3.2.2.2. PREFIX, MARKER, and TAG Keywords

User-defined prefixes, markers, and tags are optional character strings that help to uniquely identify the names associated with a particular facility or system. When the **/SUPPRESS** qualifier is specified on the command line, the inclusion of prefixes and/or tags on output symbol names is suppressed. See *Section 2.1, "Processing a VSI SDL Source File"* for a description of the **/SUPPRESS** qualifier.

PREFIX Keyword

The PREFIX option may be specified on an AGGREGATE, subaggregate, CONSTANT, or ITEM declaration to cause VSI SDL to concatenate a user-specified prefix and the name specified in the declaration.

When you specify a prefix for an aggregate, VSI SDL concatenates the prefix and the name of each member or named constant declared within the aggregate. The name of the aggregate itself is not altered by the use of the PREFIX option. The PREFIX option has the following syntax:

```
PREFIX prefix-string
```

prefix-string

Is a 0- to 32-character string that can be any valid VSI SDL identifier.

If you specify the PREFIX option, VSI SDL constructs the identifier of each member by concatenating the prefix, a tag, an underscore, and the member name.

You can override a prefix that is currently in effect by specifying a new prefix for a particular aggregate member. If this member happens to be a subaggregate, the new prefix is applied to all the members of that subaggregate. Otherwise, if no prefix is specified for the subaggregate, all subaggregate members are assigned the same prefix as that specified on the level-1 aggregate.

MARKER Keyword

You can use the MARKER keyword to assign a user-defined prefix to the aggregate name. The MARKER option has the following syntax:

```
MARKER marker-string
```

marker-string

Is a 0- to 32-character string that can be any valid VSI SDL name that may or may not be enclosed in quotation marks (" ") and may be null.

TAG Keyword

The TAG option overrides the default tags that VSI SDL uses in forming identifiers. You can specify tags for CONSTANT, ITEM, and AGGREGATE declarations and aggregate members. The tag that you specify, however, affects only the outer-level identifier. For example, a tag you supply in an AGGREGATE declaration affects only the aggregate name; if you wish to change all the tags in an aggregate, you must do it on a member-by-member basis. The TAG option has the following syntax:

```
TAG tag-string
```

tag-string

Is a 0- to 32-character string specifying the tag to use in forming the name. If the TAG option is not specified, VSI SDL uses a default code based on the data type of the name.

If you specify the TAG option, VSI SDL appends the tag-string, which may be null (" "), and an underscore character (_) to the current prefix-string. A tag consisting of a single underscore character produces a single underscore character in any resulting identifier.

The following is an example of the PREFIX, TAG, and MARKER keywords:

```
AGGREGATE operator STRUCTURE MARKER doowop$ PREFIX beebop$ TAG shoo;
 flink ADDRESS;
 blink WORD;
 END;
```

Table 3.3, "SDL Command Qualifiers and Their Defaults" shows the default tags that VSI SDL uses when the TAG option is not specified on an aggregate member.

Table 3.3. SDL Command Qualifiers and Their Defaults

Data Type	Default Tag
CONSTANT	K
BYTE [UNSIGNED]	B
WORD [UNSIGNED]	W
LONGWORD [UNSIGNED]	L
QUADWORD	Q
OCTAWORD	O
F_FLOATING	F
D_FLOATING	D
G_FLOATING	G
H_FLOATING	H
F_FLOATING COMPLEX	FC
D_FLOATING COMPLEX	DC
G_FLOATING COMPLEX	GC
H_FLOATING COMPLEX	HC
DECIMAL	P

Data Type	Default Tag
BITFIELD	V for BITFIELD offset S for BITFIELD size[1] M for BITFIELD mask[1]
CHARACTER	T
ADDRESS	A
BOOLEAN	B
VOID	Z
INTEGER	IS
INTEGER_BYTE	IB
INTEGER_WORD	IW
INTEGER_LONG	IL
INTEGER_QUAD	IQ
INTEGER_HW	IH
POINTER_HW	PH
POINTER_LONG	PL
POINTER	PS
POINTER_QUAD	PQ
HARDWARE_ADDRESS	HA
HARDWARE_INTEGER	HI
STRUCTURE	R
UNION	R

[1]Identifiers with size and mask (if MASK is specified) tags are generated regardless of whether a PREFIX or TAG option is specified.

The following example shows the use of a user-specified prefix on an AGGREGATE declaration:

```
AGGREGATE operator STRUCTURE
     PREFIX opr$;
     id WORD;
     "typename" CHARACTER;
     CONSTANT (fixed_bin_,float_) EQUALS 0 INCREMENT 1;
     bits STRUCTURE;
     variable_size BITFIELD; size_units BITFIELD LENGTH 3;
     END bits;
END operator;
```

In the previous example, the member name `typename` is enclosed in quotation marks because it is a VSI SDL keyword.

The previous declaration produces the following names, with the prefix `opr$` and default tags, in the PL/I output file:

```
%replace opr$k_fixed_bin_ by 0;
```

```
%replace opr$k_float_ by 1;

%replace opr$s_operator by 4;
dcl 1 operator based,

    2 opr$w_id fixed binary(15),
    2 opr$t_typename character(1),
    2 opr$r_bits ,
       3 opr$v_variable_size bit(1),
       3 opr$v_size_units bit(3),
       3 opr$v_fill_0 bit(4);
```

The name opr$v_fill_0 in the previous list is the result of a BITFIELD declaration that VSI SDL supplied because the subaggregate did not end on a byte boundary. The name (opr$v_ fill_0) ensures that the next aggregate begins on a byte boundary. *Section 4.3.1.7, "Forcing Data Alignment"* describes data alignment in detail. VSI SDL uses default codes followed by an underscore (_) for the tag portion of a prefix when the TAG option is not specified. You can override the default OpenVMS codes by specifying a tag, which may be null (" "). For example:

```
CONSTANT (abc,def,ghi) EQUALS 0 INCREMENT 1
PREFIX new TAG "";
```

This declaration results in the names new_abc, new_def, and new_ghi.

3.2.2.3. Alignment Keywords

Both the ALIGN and BASEALIGN keywords can ensure that items are properly aligned. The BASEALIGN keyword takes an argument, which specifies the alignment, whereas the ALIGN keyword uses the natural alignment of the item.

- If the ALIGN keyword is included in the definition of an aggregate, every member will be aligned. Both i1 and i2 will be aligned.

  ```
  AGGREGATE MyStruct STRUCTURE ALIGN;
       c1 CHARACTER;
       i1 LONGWORD;
       c2 CHARACTER;
       i2 LONGWORD;
  END;
  ```

- If the ALIGN keyword is included in the definition of a member of an aggregate, this member will be aligned, even if the AGGREGATE itself does not have the ALIGN attribute. i2 will be aligned whereas i1 will not be aligned.

  ```
  AGGREGATE MyStruct STRUCTURE NOALIGN;
       c CHARACTER;
       i1 LONGWORD;
       i2 LONGWORD ALIGN;
  END;
  ```

- If the NOALIGN keyword is included in the definition of a member of an aggregate, no action will be taken to ensure that this member will be aligned, even if the AGGREGATE has the ALIGN attribute. i2 will be aligned, i1 will not be aligned.

  ```
  AGGREGATE MyStruct STRUCTURE ALIGN;
       c CHARACTER;
       i1 LONGWORD NOALIGN;
  ```

```
     i2 LONGWORD;
END;
```

- If the NOALIGN keyword is included in the definition of an aggregate, no action will be taken to ensure that any member of this aggregate will be aligned. Neither i2 nor i1 will be aligned.

```
AGGREGATE MyStruct STRUCTURE NOALIGN;
     c CHARACTER;
     i1 LONGWORD;
     i2 LONGWORD;
END;
```

- If the BASEALIGN keyword is included in the definition of a member of an aggregate, this member will be aligned according to the given alignment. In this case, i2 will have an offset of 256 (2^8),

```
AGGREGATE MyStruct STRUCTURE NOALIGN;
     c CHARACTER;
     i1 LONGWORD;
     i2 LONGWORD BASEALIGN(8);
END;
```

- If the BASEALIGN keyword is included in the definition of an aggregate or an item, the aggregate or item itself will be padded, so that in an array of elements of this aggregate or item type, all elements will be aligned according to the given alignment. The syntax is as follows:

```
AGGREGATE MyStruct STRUCTURE BASEALIGN (2);
     c CHARACTER;
     i1 LONGWORD;
     i2 LONGWORD;
END;
```

The size of the aggregate will be a multiple of 4 (2^2), in this case 12, and neither i1 nor i2 will be aligned.

In the following example, the item will have a size of 8 (2^3).

```
ITEM MyItem LONGWORD UNSIGNED BASEALIGN (3);
```

3.2.2.4. Storage Class Keywords

Storage class refers to the way in which the target language compiler allocates storage for scalar items and aggregates. In general, declarations produce a template describing data for which the compiler allocates storage dynamically at run time, rather than at compile time. This type of storage is the default and is specified using the BASED option in some languages, although the default storage class option is language-dependent. The PL/I back end generates the BASED storage class option, which has the following syntax:

```
BASED (pointer-name)
```

In languages that support the construct, you can use the BASED pointer-name option on an AGGREGATE declaration to bind a named pointer to that aggregate. In all target languages, the aggregate resulting from such a declaration has the default storage class (BASED).

A subaggregate always acquires the storage class of the aggregate to which it belongs.

The default storage class associated with any declaration is language-dependent. You can override the default storage class by specifying either of the following storage classes:

- **Common storage** is allocated in an external program section (Psect) with the OVR option and is shared by all routines that reference it. You declare data in common storage by using the COMMON option on an AGGREGATE or ITEM declaration. For example, the VSI MACRO back end places all declarations in an absolute program section by default and associates the names with the offset values.

- **Global storage** represents data in a global storage location whose value is defined elsewhere. You declare global data by using the GLOBAL option on an AGGREGATE or ITEM declaration.

  For some languages, the **/GLOBALDEF** qualifier on the SDL command affects the operation of the GLOBAL option. When this qualifier is used, GLOBAL generates a reference to global data whose value is defined in this module. You can see the effect of **/GLOBALDEF** for each output language by processing the VSI SDL source file EXAMPLE.SDL in SDL$EXAMPLES.

- **TYPEDEF** behaves like a storage class. In C, examples of storage classes are static, globaldef/ref, extern, etc. Syntactically, you can replace a 'static' in any declaration with 'typedef' and have it compile. Storage classes (including TYPEDEF) are mutually exclusive in an SDL declaration.

  Example of C type definition:

  ```
  typedef struct {int jg$l_i1; int jg$l_i2;} MyStruct ;
   static MyStruct foo ;
  ```

  is equivalent to:

  ```
  static struct {int jg$l_i1; int jg$l_i2;} foo ;
  ```

  or:

  ```
  static struct MyStruct {int jg$l_i1; int jg$l_i2;} foo ;
  ```

  or

  ```
  struct MyStruct {int jg$l_i1; int jg$l_i2;} ;
  static struct MyStruct foo ;
  ```

If COMMON and GLOBAL appear together in a declaration, a DUPCONATT (duplicate or conflicting attributes) error is given. The COMMON, GLOBAL and TYPEDEF storage classes are mutually exclusive.

3.2.2.5. SDL Storage Classes and Typedef Syntax

Explicit SDL Storage classes are COMMON and GLOBAL. To maintain orthogonal syntax, TYPEDEF is permitted wherever COMMON and GLOBAL are permitted.

The Storage class definition in SDL is included as an option on an ITEM or AGGREGATE declaration. This means that an AGGREGATE TYPEDEF for the preceding example would be:

```
AGGREGATE MyStruct STRUCTURE TYPEDEF PREFIX jg$ ;
     i1 LONGWORD ;
     i2 LONGWORD ;
END ;
```

3.2.2.6. Data Types

In most cases it is possible to refer to a user-defined Data Type where a standard built-in SDL Data Type (BYTE, LONGWORD etc.) can be referenced.

Reference to the type MyStruct defined in the preceding example is made as follows:

```
ITEM foo MyStruct ;
```

3.2.2.7. DIMENSION Keyword

You can apply a dimension to AGGREGATE (as well as members of aggregates and subaggregates) and ITEM declarations, which means that you can define an array of structures, a structure that contains one or more arrays, or an array of structures each of which contains one or more arrays. The DIMENSION option is valid when specified with any of the data types described in *Section 3.2.3, "Data Type Keywords"* and has the following syntax:

```
DIMENSION [lbound:]hbound
```

lbound

Is any valid VSI SDL expression giving the value of the lowest-numbered element of the array. If lbound is not specified, VSI SDL supplies a default lbound of 1.

hbound

Is any valid VSI SDL expression giving the number of elements in the array, or, if lbound is specified, the highest-numbered element.

The following is an example of the DIMENSION option specified on an AGGREGATE declaration:

```
AGGREGATE array_info STRUCTURE;
   bound STRUCTURE DIMENSION 8;
        lower LONGWORD;
        upper LONGWORD;
        multiplier LONGWORD;
        constant_lower BITFIELD LENGTH 1;
        constant_upper BITFIELD LENGTH 1;
        constant_multiplier BITFIELD LENGTH 1;
        reserved BITFIELD LENGTH 13;
   END bound;
END array_info;
```

The subaggregate bound has eight elements. Each element consists of the members upper, lower, multiplier, and so on. Because bound is an array, each of its members (upper, lower, multiplier, and so on) can also be considered an array of eight elements.

When you specify a single value after the DIMENSION keyword, as in the previous example, the specified back end assumes that the value represents the high bound value and supplies a default low bound value of one. You can override this default by specifying both a low bound and a high bound value as follows:

```
ITEM node_pointers DIMENSION 0:255 ADDRESS;
```

This declaration results in a declaration of the array `node_pointers`, whose low bound is 0 and whose high bound is 255. Only one dimension can be specified for an AGGREGATE or ITEM declaration. This restriction ensures that there is no interlanguage conflict in an array declaration.

You can see how VSI SDL translates the DIMENSION option for each output language by processing the VSI SDL source file EXAMPLE.SDL in SYS$EXAMPLES.

3.2.3. Data Type Keywords

Data type keywords specify the data types of scalar objects, which can be declared as members of aggregates or as individual items. Data type keywords are also used to describe the data types of parameters, as well as the return value of an ENTRY. They can also be used in an AGGREGATE declaration to generate an implicit union. The data type declaration also specifies, either implicitly or explicitly, the size of a member.

The following sections describe the data types and the keywords you use to specify them.

3.2.3.1. Pointer Data Types

The keywords ADDRESS, POINTER, POINTER_LONG, POINTER_QUAD, POINTER_HW, and HARDWARE_ADDRESS specify a data type that is an address, or pointer. The ADDRESS data type has the following syntax:

```
pointer-type [ (object-type [ basealign-attribute ] ) ]
```

pointer-type

is one of the keywords ADDRESS, POINTER, POINTER_LONG, POINTER_QUAD, POINTER_HW, and HARDWARE_ADDRESS.

ADDRESS, POINTER, POINTER_LONG are 4-byte-addresses.

POINTER_HW and HARDWARE_ADDRESS are 4-byte-addresses if **/VAX** us specified and 8-byte-addresses if **/ALPHA_AXP** is specified. POINTER_QUAD is an 8-byte-address.

object-type

is the optional data type of the object to which the address refers. This construct is ignored for languages in which pointers are distinct data types. Object-type is either a builtin object type, like LONGWORD or ANY, or a user-defined object type, optionally followed by a DIMENSION specification, or an ENTRY declaration.

basealign-attribute

Here, a BASEALIGN attribute can be specified, as desribed in *Section 3.2.2.3, "Alignment Keywords"*.

The following is an example of an aggregate with a member of pointer type:

```
AGGREGATE any_node STRUCTURE;
      flink ADDRESS (any_node);
      blink ADDRESS (any_node);
      END;
```

3.2.3.2. ANY Data Type

The ANY keyword specifies that the parameter being described in an ENTRY declaration can be of any data type. The ANY data type can be used only within the context of a parameter description and has the following syntax:

```
ANY
```

ANY

Specifies that the parameter can be of any data type.

The following is an example of the use of the ANY data type:

```
ENTRY sys$abc PARAMETER (ANY);
```

3.2.3.3. BITFIELD Data Type

The BITFIELD keyword specifies a bit field variable that must be a member of an AGGREGATE declaration. The BITFIELD data type has the following syntax:

```
BITFIELD [LENGTH n] [MASK] [SIGNED]
```

[LENGTH n]

Is any valid VSI SDL expression giving the number of bits in the bitfield. If no length is specified, VSI SDL uses a default length of 1 bit.

[MASK]

Is a keyword specifying that VSI SDL generate both a bitfield variable and a constant bit mask representing the bits defined in this field.

[SIGNED]

Is a keyword specifying that VSI SDL treat the output as a signed field.

Bitfields must be AGGREGATE declaration members. They cannot be scalar items, objects of ADDRESS declarations, parameters, or return data types of entries.

The following is an example of the BITFIELD keyword used in an AGGREGATE declaration that specifies a structure with two bitfield members:

```
AGGREGATE flags STRUCTURE PREFIX tst$;
          resolved BITFIELD MASK SIGNED;
          spare_bits BITFIELD LENGTH 5;
          END;
```

The declaration of `resolved` in the previous example results in two declarations in the output file: a declaration for the bitfield itself and a declaration of a constant mask whose value is 1. Because the PREFIX option is specified for this aggregate, the source output file identifiers produced for this declaration are `tst$v_resolved` and `tst$m_resolved`, where the tag *v_* indicates the bitfield variable and the tag *m_* indicates the mask. Prefixes and tags are described in more detail in *Section 3.2.2.2, "PREFIX, MARKER, and TAG Keywords"*.

3.2.3.4. BOOLEAN Data Type

The BOOLEAN keyword specifies a Boolean data type that is a one-byte field that can have one of two values, 0 or 1. The BOOLEAN data type has the following syntax:

```
BOOLEAN
```

BOOLEAN

Produces a Boolean variable in the output file.

The following is an example of the BOOLEAN keyword used in an ITEM declaration:

```
ITEM true BOOLEAN;
```

3.2.3.5. CHARACTER Data Type

The CHARACTER keyword declares a character string of a given length. The CHARACTER data type has the following syntax:

```
 CHARACTER [LENGTH {n}] [VARYING]
           [       {*}]
```

[LENGTH n]

Is the length of the character string. The length can be specified using any valid VSI SDL expression. If no length is specified, VSI SDL uses a default length of 1 character. You can specify a character string of unknown length using LENGTH *. An unknown length character string can be specified only for parameter types. For example:

```
ENTRY LIB$ROUTINE PARAMETER(CHARACTER LENGTH * NAMED foo);
```

[VARYING]

Is a keyword indicating that the identifier represents a varying-length character string (for languages that support this data type). In a varying-length character string, the first word of the string contains its current length; its declared length is the maximum length that it can have.

The following is an example of the CHARACTER data type used in an AGGREGATE declaration. The aggregate `msg_buffer` contains a character-string member named `message_text` with a length of 256 characters.

```
AGGREGATE msg_buffer STRUCTURE;
    message_text CHARACTER LENGTH 256 VARYING;
    severity WORD;
    END;
```

3.2.3.6. COMPLEX Data Types

The COMPLEX keyword immediately follows any of the four floating-point data type keywords to specify any of the four COMPLEX data types. The COMPLEX data types have the following syntax:

```
F_FLOATING COMPLEX
D_FLOATING COMPLEX
G_FLOATING COMPLEX
H_FLOATING COMPLEX
```

The following is an example of the F_FLOATING COMPLEX data type used in an ITEM declaration:

```
ITEM foo F_FLOATING COMPLEX;
```

Section 3.2.3.8, "Floating-Point Data Types" describes floating-point data types.

3.2.3.7. DECIMAL Data Type

The DECIMAL keyword specifies a packed decimal data type and the size of the data type. The DECIMAL data type has the following syntax:

```
DECIMAL PRECISION (precision,scale)
```

PRECISION (precision,scale)

Is the fixed-point decimal member's precision and scale, respectively. Precision is the total number of decimal digits, and scale is the number of fractional digits.

Both precision and scale must be specified using valid VSI SDL expressions.

The following is an example of the DECIMAL keyword used in an ITEM declaration that indicates a packed decimal data type consisting of three decimal digits, two of which are fractional:

```
ITEM percentage DECIMAL PRECISION (3,2);
```

3.2.3.8. Floating-Point Data Types

The floating-point keywords, F_FLOATING, D_FLOATING, G_FLOATING, and H_FLOATING, declare storage units for single-, double-, G-, and H-floating-point data, respectively. The floating-point data types have the following syntax:

```
F_FLOATING
D_FLOATING
G_FLOATING
H_FLOATING
```

The following is an example of the D_FLOATING data type used in an ITEM declaration:

```
ITEM foo D_FLOATING;
```

Section 3.2.3.6, "COMPLEX Data Types" describes COMPLEX data types.

3.2.3.9. Integer Data Types

The keywords BYTE, WORD, LONGWORD, QUADWORD, and OCTAWORD declare storage units of 8, 16, 32, 64, and 128 bits, respectively, to represent signed integer data.

You may also specify the keyword UNSIGNED with any of these data types to indicate unsigned integer data.

It is also possible to specify the keyword SIGNED.

The keywords INTEGER_BYTE, INTEGER_WORD, INTEGER_LONG and INTEGER_QUAD are synonyms for BYTE, WORD, LONGWORD, QUADWORD, respectively, although some back ends treat INTEGER_QUAD and QUADWORD different (C, Fortran and PL/I). The keyword INTEGER is also synonym for LONGWORD and INTEGER_LONG.

The keywords INTEGER_HW and HARDWARE_INTEGER describe integer data types whose size depends on the underlying hardware. If the qualifier **/VAX** is specified, they are 4 bytes wide, and if **/ALPHA_AXP** is specified, they are 8 bytes wide.

The integer data types have the following syntax:

```
BYTE              [ UNSIGNED | SIGNED ]
INTEGER_BYTE      [ UNSIGNED | SIGNED ]
WORD              [ UNSIGNED | SIGNED ]
INTEGER_WORD      [ UNSIGNED | SIGNED ]
LONGWORD          [ UNSIGNED | SIGNED ]
INTEGER_LONG      [ UNSIGNED | SIGNED ]
```

```
INTEGER           [ UNSIGNED | SIGNED ]
QUADWORD          [ UNSIGNED | SIGNED ]
INTEGER_QUAD      [ UNSIGNED | SIGNED ]
OCTAWORD          [ UNSIGNED | SIGNED ]
INTEGER_HW        [ UNSIGNED | SIGNED ]
HARDWARE_INTEGER  [ UNSIGNED | SIGNED ]
```

The following are examples of the LONGWORD and BYTE data types used in an ITEM declaration:

```
ITEM foo LONGWORD UNSIGNED;
ITEM bar BYTE;
```

3.3. Expressions

A VSI SDL expression evaluates to an arithmetic value and can consist of any of the following syntax elements:

- **Numeric values** are, by default, expressed in decimal notation. You can override this default by preceding a constant with one of the prefixes in the following table.

Prefix	Interpretation	Valid Characters
%X	Hexadecimal	0 - 9, A - F
%O	Octal	0 - 7
%B	Binary	0 and 1
%A[1]	ASCII value	Any ASCII character

[1]The %A operator takes the ASCII value of any ASCII character that follows it.

  VSI SDL treats decimal constants as signed integer longword values.

  You can also use a string of up to four characters as a numeric constant by enclosing the string in quotation marks (" "). VSI SDL inserts the ASCII value of each character into the byte field corresponding to that character's position in the string. If the string you specify has fewer than four characters, VSI SDL pads the string with the null character, which has the ASCII code of zero.

- **Local symbols and output constants** are assigned integer longword values that are available within the context (that is, during processing) of the input file.

- **Operators** perform arithmetic and logical operations on numeric values, local symbols, and output constants. The following table lists the operators in order of precedence, with the operators of higher precedence listed first.

Operator	Meaning
unary -	Arithmetic negation
*	Arithmetic multiplication
/	Arithmetic division
+	Arithmetic addition
-	Arithmetic subtraction
@	Logical shift–x@y shifts the value of x to the left y places; if y is negative, the value of x is shifted y places to the right

Operator	Meaning
&	Logical AND
!	Logical OR

- **Offset symbols** are used in expressions specified in AGGREGATE declarations:
  - The period (.) represents the current byte offset from the origin in an AGGREGATE declaration. If the ORIGIN option is specified, the value of the period is equal to the byte offset from the member specified using the ORIGIN option.
  - The colon (:) represents the current byte offset relative to the first member in an AGGREGATE declaration. The value is not affected by the presence of an ORIGIN option.
  - The circumflex (^) represents the current bit offset relative to the most recently declared aggregate or byte-aligned element.

  Section 4.3.1.8, "Using Offset Symbols" describes the use of offset symbols in AGGREGATE declarations in more detail.
- **Parentheses** group expressions to define the order of evaluation. Expressions within the innermost set of parentheses are evaluated first.

The following is an example of an expression used in a CONSTANT declaration, which appears in the context of an AGGREGATE declaration:

```
CONSTANT foo EQUALS %Ag + 72 / (#abc * boo + .);
```

3.4. Local Comments

A comment that is local to the VSI SDL source file is not written to the output file. Local comments begin with the left brace ({) and extend to the end of the line. They can appear anywhere within the source file (not necessarily within a module) where white space (a space, tab, or carriage return) is allowed. The following is an example of a local comment:

```
{Assigning the value 255 to #max_args.
```

3.5. Output Comments

Comments appearing on lines by themselves are typically written to the output file as separate comment lines. Comments appearing at the end of a line are output at the end of the corresponding target source line, if possible.

Output comments begin with a slash and an asterisk (/*) and terminate at the end of the current line. They can appear in the following contexts:

- Outside MODULE declarations
- At the end of a line containing a declaration, that is, following the semicolon terminator (;)
- On lines by themselves between member, CONSTANT, ENTRY, ITEM, and AGGREGATE declarations
- Between declarations within an aggregate

- Following individual constant names within a comma-delimited list of CONSTANT declarations

The following is an example of an output comment:

```
/* Get Job/Process Information System Service.
```

3.6. INCLUDE Statement

The INCLUDE statement specifies that the contents of an external file are to be incorporated in the VSI SDL input file directly following the INCLUDE statement. The INCLUDE statement has the following syntax:

```
INCLUDE "file-spec";
```

"file-spec"

Is any valid VMS file specification enclosed in quotation marks (" "). An INCLUDE statement cannot appear embedded within an AGGREGATE declaration, but can appear anywhere else within the module. If a directory is not included in the file specification, the current default directory is used.

When VSI SDL encounters an INCLUDE statement, it stops reading from the current file and reads the statements in the included file. When it reaches the end of the included file, VSI SDL resumes translation with the source statement immediately following the INCLUDE statement.

3.7. READ Statement

The READ statement allows a .SDI intermediate file to be included in an SDL source, without causing it to be emitted to the output stream. Apart from these two differences, its operation is similar to INCLUDE.

The READ statement allows previously-compiled definitions, such as constants and userdefined types, to be used by the current compilation, without affecting the language output. The READ statement has the following syntax:

```
READ "file-spec" ;
```

3.8. Conditional SDL Compilation

3.8.1. Conditional SDL Compilation Using the IFLANGUAGE Statement

This feature allows a section of SDL source code to be conditionally compiled, depending on whether output is being generated for a particular language or not.

The syntax for conditional compilation has the format:

```
IFLANGUAGE language-name [ language-name ... ] .
          .
 . [ELSE ;
          .
          .
```

```
          .
  ]
END_IFLANGUAGE [ language-name [ language-name ... ] ] ;
```

Note the following:

- The three keywords may appear wherever a statement is valid.
- Conditional compilation statements may not be nested.
- The list of language-names on the END_IFLANGUAGE is optional, but if it is included, it must match the list on the IFLANGUAGE statement. The languages need not necessarily appear in the same order.
- Language names may not be abbreviated.
- The validity of language names is not checked. This is in keeping with the philosophy of SDL that new back ends may be added without changes to the front end.
- A comment on the same line as the IF_LANGUAGE statement is only output for languages which satisfy the condition. A comment on the same line as the END_IFLANGUAGE statement is always output, as this is considered to be outside the body of the conditional.

In the following example, SDL generates a translation of the ITEM statement if output is being generated for Pascal, Ada, or Fortran. For other languages, SDL does not generate a translation of the ITEM statement.

```
IFLANGUAGE PASCAL ADA FORTRAN ;
ITEM foo LONGWORD;
END_IFLANGUAGE PASCAL ADA FORTRAN;
```

3.8.2. Conditional SDL Compilation Using the IFSYMBOL Statement

This feature allows a section of SDL source code to be conditionally compiled, depending on symbols specified with the command line qualifier **/SYMBOLS**.

The syntax for conditional compilation has the format:

```
IFSYMBOL symbol-name ;
      .
      .
      .
{ ELSE_IFSYMBOL symbol-name ;
      .
      .
      .
}
[ELSE ;
      .
      .
      .
]
END_IFSYMBOL ;
```

Note the following:

- The four keywords may appear wherever a statement is valid.
- Conditional compilation statements may not be nested.
- A comment on the same line as the IFSYMBOL statement is only output if this symbol is specified on the command line. A comment on the same line as the END_IFSYMBOL statement is always output, as this is considered to be outside the body of the conditional.

With the following example:

```
IFSYMBOL s1;
    <sdl code 1>
END_IFSYMBOL;
IFSYMBOL s2;
    <sdl code 2>
ELSE_IFSYMBOL s3;
    <sdl code 3>
ELSE;
    <sdl code 4>
END_IFSYMBOL;

$ SDL/LANG=<LANGUAGE>/SYMBOLS=(S1=0,S2=0,S3=0) <FILE-SPEC>
```

produces

```
<sdl code 4>
$ SDL/LANG=<LANGUAGE>/SYMBOLS=(S1=1,S2=0,S3=0) <FILE-SPEC>
```

produces

```
<sdl code 1>
<sdl code 4>

$ SDL/LANG=<LANGUAGE>/SYMBOLS=(S1=1,S2=1,S3=x) <FILE-SPEC>
```

produces

```
<sdl code 1>
<sdl code 2>

$ SDL/LANG=<LANGUAGE>/SYMBOLS=(S1=1,S2=0,S3=1) <FILE-SPEC>
```

produces

```
<sdl code 1>
<sdl code 3>
```

3.9. Text Pass-Through

Text pass-through allows literal text to be passed through to the output language file without translation. It is normally used in conjunction with conditional compilation for a specific target language. The purpose is to allow language-specific constructs, which cannot be represented in SDL, to be emitted.

The syntax for text pass-through is:

```
LITERAL;
Any number of lines to be passed directly to
```

```
the output stream without translation
END_LITERAL;
```

Note the following:

- The keywords LITERAL and END_LITERAL may appear wherever a statement is valid.
- The keyword END_LITERAL terminates a literal construct, and therefore cannot be included in a line of literal text. Any text preceding END_LITERAL on the same line is output as a line of literal text.

Literal text is processed identically in all back ends. The literal text is merely written directly to the output file.

The following is an example of text pass-through for the C language.

```
IFLANGUAGE CC
LITERAL
#define ctext "This appears in C language output only"
END_LITERAL
END_IFLANGUAGE CC
```

3.10. DECLARE Statement

The DECLARE statement allows you to declare a data item of a type that you define, which may be unknown in the current SDL compilation. When you use DECLARE, the type is made known when the target language source is compiled.

The DECLARE statement uses the SIZEOF clause to allow you to specify the size of the userdefined data type being declared. The parameter you specify in the SIZEOF clause may be a built-in SDL data type, a user-defined type defined in the current SDL compilation, or an expression.

The DECLARE statement has the following syntax:

```
DECLARE user-type SIZEOF { data-type }   [ PREFIX prefix-string ]
                         { user-type }   [ TAG tag-string       ]
                         { ( expression ) }
```

user-type

Represents the unknown data type name you wish to declare.

SIZEOF

A clause that must be appended to *user-type*. The SIZEOF clause may be specified in several ways, as shown, to indicate the size of the user-defined type being declared.

data-type

Represents either a built-in SDL data type, a user-defined type that is known at SDL compile time, or a data type that has been sized by a previous SIZEOF clause.

user-type

Represents a user-defined data type.

(expression)

Represents an expression. If specified, the expression must specify the number of bytes to be reserved for this data type. If the data type is dimensioned, the SIZEOF clause must specify the size of a single element. When you specify an expression, always enclose it in parentheses, as shown in this syntax.

Notes:

1. DECLARE identifies the size of the data type when included in an AGGREGATE declaration. (SDL needs to know its equivalent predefined type so that the correct default tag letter, if required, can be output.)

2. You may declare a user-defined data type more than once (either explicitly or implicitly), but any subsequent declaration must match the first.

3. The default tag letter for the unknown variable being sized is derived from the type specified in the SIZEOF clause. If you use an expression in place of a data type to reserve a fixed number of bytes, the default tag letter is T. You may override the default tag by using an explicit TAG option.

4. SIZEOF clauses cannot be nested.

5. You cannot qualify a reference to the name of a previously-declared aggregate using the SIZEOF clause.

6. Although DECLARE statements and implicit SIZEOF declarations appear in the output tree for use by the back ends, these do not result in specific generated code.

7. Do not use the SIZEOF clause for data types that are aggregate names. Also, do not nest SIZEOF clauses where the syntax would otherwise allow aggregate names. For example, the following statements are valid:

   ```
   DECLARE type SIZEOF ADDRESS (CHARACTER);
   ITEM type ADDRESS (bar SIZEOF LONGWORD);
   ```

   However, the following statement generates an error message:

   ```
   DECLARE type SIZEOF ADDRESS (bar SIZEOF LONGWORD);
   ```

Examples

A database contains information on a number of forests in a region, each of which contains a number of different types of trees. The definition of the *tree* structure is held in a different SDL file from the other definitions. When the second SDL file is compiled, the composition of the *tree* structure is unknown — the definitions will only come together when the output files are included in a compilation in the target language.

The following shows the two SDL files and the corresponding output in C.

Example 3.1. TREE1.SDL:

```
AGGREGATE tree STRUCTURE TYPEDEF;
     flink ADDRESS (tree);
     blink ADDRESS (tree);
     height LONGWORD;
     age LONGWORD;
```

```
END;
```

Example 3.2. TREE2.SDL:

```
DECLARE tree SIZEOF (16);
AGGREGATE forest STRUCTURE TYPEDEF;
     oak tree;
     ash tree;
     elm tree;
     confirs tree DIMENSION 6;
END;
ITEM tree_pointer ADDRESS (tree);
ITEM tree_storage tree DIMENSION 1000;
ITEM region forest DIMENSION 4; { No SIZEOF, since 'forest' defined here}
```

Example 3.3. TREE1.H:

```
typedef struct _tree {
     _tree     *flink;
     _tree     *blink;
     long int height;
     long int age;
     } tree;
```

Example 3.4. TREE2.H:

```
typedef struct _forest {
    tree     oak;
    tree     ash;
    tree     elm;
    tree     conifers[6];
    }  forest;
tree    *tree_pointer;
tree    tree_storage[1000];
forest  region[4];
```

Note that the SIZEOF information is discarded by C, but is used by other languages, such as VSI MACRO.

Chapter 4. VSI SDL Declarations

This chapter describes the function and format of each of the following VSI SDL declarations:

- MODULE declaration
- ITEM declaration
- AGGREGATE and subaggregate declarations
- CONSTANT declaration
- ENTRY declaration

VSI SDL declarations are composed of the language elements described in *Chapter 3, "VSI SDL Language Elements"*. The output generated by each VSI SDL declaration depends on which back end is used for the translation.

Online examples of output files for each language are available by processing the VSI SDL source file EXAMPLE.SDL in SYS$EXAMPLES. *Appendix B, "VSI SDL Language Translation Summaries"* provides translation summaries for each output language.

4.1. MODULE Declaration

The following sections describe the function and format of a MODULE declaration.

4.1.1. MODULE Description

A MODULE declaration groups all related symbols and data structures. All declarations (other than local symbol assignments and local and output comments) must occur within a module, which is delimited by the MODULE and END_MODULE keywords. A VSI SDL source file may contain multiple MODULE declarations, but modules may not be nested.

You must specify a module name on the MODULE declaration; this name corresponds to the name of a macro or module in a given output language. (*Section 3.1, "Names"* describes the syntax for names.) For example:

```
MODULE $modef;
```

This declaration generates the beginning of the declaration for the macro or module named `$modef`.

You can use the IDENT option on a MODULE declaration to pass a version number or other information that must be enclosed in quotation marks (" ") to the output file. For example:

```
MODULE params IDENT "V2.0";
```

You can optionally specify the module name on the END_MODULE statement; this is particularly useful if you place more than one MODULE declaration in the same VSI SDL source file. The module name is case-sensitive and must match exactly (in case) the name specified on the MODULE declaration.

4.1.2. MODULE Format

A MODULE declaration has the following syntax:

```
MODULE module-name [ IDENT "ident-string" ];
[ module-body ];
      .
      .
      .
END_MODULE [ module-name ];
```

MODULE module-name

Specifies any valid VSI SDL identifier you want to use to identify the module.

[IDENT "ident-string"]

Specifies any valid VSI SDL identifier, or a string of any characters, that must be enclosed in quotation marks (" ") and that either further identifies the module or is a version number.

[module-body]

Is one or more of the following:

- ITEM declaration
- AGGREGATE declaration
- CONSTANT declaration
- ENTRY declaration
- Local symbol assignment
- Local or output comment
- INCLUDE statement
- DECLARE statement
- READ statement
- IFLANGUAGE construct
- IFSYMBOL construct
- LITERAL construct

END_MODULE [module-name]

Marks the end of the MODULE declaration. The module-name, if specified, must match the name on the most recently specified MODULE declaration.

4.2. ITEM Declaration

The following sections describe the function and format of an ITEM declaration.

4.2.1. ITEM Description

An ITEM declaration defines scalar items and single-dimensional arrays of scalar items that are not members of aggregates. You must specify an item name and data type on the ITEM declaration. For example:

```
ITEM block_list_id WORD;
```

This declaration specifies the scalar item block_list_id of data type WORD.

4.2.2. ITEM Format

An ITEM declaration has the following syntax:

```
ITEM item-name   { data-type }           [ COMMON ]
                 { user-type sizeopt }   [ GLOBAL ]
                                         [ TYPEDEF ]
                                         [ BASEALIGN basealign-option ]
                                         [ DIMENSION [ lbound: ] hbound ]
                                         [ PREFIX prefix-string ]
                                         [ TAG tag-string ];
```

ITEM item-name

Specifies any valid VSI SDL name used to identify the item.

data-type

Is any valid VSI SDL data type (see *Section 3.2.3, "Data Type Keywords"*).

user_type sizeopt

Is a user-defined data type using the DECLARE statement's SIZEOF clause, shown and described in *Section 3.10, "DECLARE Statement"*.

[COMMON]
[GLOBAL]
[TYPEDEF]

Is the storage class of the item, if other than the default (based) storage class (see *Section 3.2.2.4, "Storage Class Keywords"*).

[BASEALIGN basealign-option]

BASEALIGN specifies the alignment of an ITEM. basealign-option is either an integer expression in parentheses or the name of a data type.

BASEALIGN aligns the item on a multiple of the value of the basealign-option.

[DIMENSION [lbound:]hbound]

Specifies that the item is an array. If a single value is specified, that value indicates the number of elements in the array. Otherwise, lbound and hbound represent lower and upper bounds of the array, respectively (see *Section 3.2.2.7, "DIMENSION Keyword"*).

[PREFIX prefix-string]

Specifies a user-defined prefix that becomes part of the identifier. It can be any valid VSI SDL name with 0 to 32 characters, may or may not be enclosed in quotation marks (" "), and may be null (see *Section 3.2.2.2, "PREFIX, MARKER, and TAG Keywords"*).

[TAG tag-string]

Specifies the user-defined tag that follows the prefix used in forming the identifier. It can be any valid VSI SDL name with 0 to 32 alphabetic or numeric characters; the tag must be enclosed in quotation marks (" ") if it begins with a numeric character (see *Section 3.2.2.2, "PREFIX, MARKER, and TAG Keywords"*).

4.3. AGGREGATE Declaration

The following sections describe the function and format of AGGREGATE and subaggregate declarations.

4.3.1. AGGREGATE Description

An AGGREGATE declaration defines nonscalar items and dimensional arrays of nonscalar items. An aggregate must contain at least one member declaration. The body of an aggregate can also contain local symbol assignments, CONSTANT declarations, and subaggregate declarations.

You must specify a valid aggregate name on the AGGREGATE declaration. For example:

```
AGGREGATE dcb STRUCTURE;
      type CHARACTER;
      size WORD;
      next ADDRESS;
      END dcb;
```

You must terminate an AGGREGATE declaration with the END statement. You can optionally specify the aggregate name on the END statement. The aggregate name is case-sensitive and must match exactly (in case) the name specified on the AGGREGATE declaration.

An aggregate that is the target of an ADDRESS declaration must have the default (BASED) storage class.

4.3.1.1. Subaggregate Declaration

Subaggregates are declared using the keyword STRUCTURE or UNION. Within an aggregate, subaggregates can be nested up to eight levels deep. For example:

```
AGGREGATE tree_node STRUCTURE;
  opcode WORD;
  lang_bits UNION;
   pli_bits STRUCTURE;
      resolved BITFIELD;
      psv BITFIELD;
      mark1 BITFIELD;
      spare_bits BITFIELD LENGTH 5;
      END pli_bits;
   c_bits STRUCTURE;
      value_variable_size BITFIELD;
      psv BITFIELD;
      expanded BITFIELD;
      resolved BITFIELD;
      reduced BITFIELD;
      spare_bits BITFIELD LENGTH 3;
      END c_bits;
   END lang_bits;
```

```
END tree_node;
```

In this example, the structures `pli_bits` and `c_bits` are both subaggregates of the union `lang_bits`. Because `lang_bits` is a union, `c_bits` and `pli_bits` occupy the same storage.

The COMMON, GLOBAL, BASED pointer-name, and ORIGIN options are invalid for a subaggregate. All other AGGREGATE options are valid.

4.3.1.2. STRUCTURE Declaration

STRUCTURE declarations produce aggregate or subaggregate declarations that are structures. The members are not overlaid; each member has a unique offset from the beginning of the structure, which means that members occupy consecutive storage locations.

The following shows the syntax of a first-level STRUCTURE declaration:

```
AGGREGATE aggregate-name STRUCTURE [ options ];
```

The following shows the syntax of a STRUCTURE declaration as an aggregate member, that is, a subaggregate STRUCTURE declaration:

```
member-name STRUCTURE [ options ];
```

4.3.1.3. UNION Declaration

UNION declarations produce aggregate or subaggregate declarations that are unions. The first-level members are overlaid, which means that they occupy the same storage location. Each first-level member begins at the beginning of the union and, thus, has an offset of zero. A UNION declaration lets you represent the same storage location using different names and different data types.

The following shows the syntax of a first-level UNION declaration:

```
AGGREGATE aggregate-name UNION [ options ];
```

The following shows the syntax of a UNION declaration as an aggregate member, that is, a subaggregate UNION declaration:

```
member-name UNION [ options ]
```

4.3.1.4. Implicit Union Declarations

You may specify data types on an AGGREGATE declaration to cause the AGGREGATE declaration to become an implicit union declaration. An implicit union declaration has these features:

- Gives VSI SDL the ability to detect subfield overflow without the need to define extraneous STRUCTURE and UNION declaration names.
- Makes user-defined fill fields unnecessary. This feature is most useful for data structures containing substructures that are required to begin at fixed offsets.

The following VSI SDL source code shows the syntax of an implicit union (structure B defines the implicit union declaration):

```
AGGREGATE A STRUCTURE;
   B STRUCTURE LONGWORD;
      bit_string1 BITFIELD LENGTH 1;
      bit_string2 BITFIELD LENGTH 4;
   END B;
```

```
   last_item WORD;
END A;
```

This implicit union declaration would be more cumbersome if represented as shown in the following STRUCTURE subaggregate:

```
AGGREGATE A STRUCTURE;
   X UNION ;
     B LONGWORD;
     Y STRUCTURE ;
        bit_string1 BITFIELD LENGTH 1;
        bit_string2 BITFIELD LENGTH 4;
     END Y;
   END X;
   last_item WORD;
END A;
```

In the previous example, the names X and Y become VSI SDL generated filler names in the implicit union case.

You do not have to define the union X and the structure Y if an implicit union declaration is used. By giving the structure Y a type, VSI SDL creates a union of field Y with the specified type overlaid with a structure containing the fields in Y. In some language translations, such as VSI BLISS and VSI MACRO, the VSI SDL-generated union and structure (X and Y above) can be suppressed in the output because they are considered extraneous fields. In other languages, such as PL/I and VSI C, it is necessary to use the VSI SDL-generated union and structure to generate the correct offsets within a structure. Because the union is of the length specified in the structure type, no filler is necessary to ensure that subsequent fields (for example, **last_item**) are at the correct offset. If the fields of a structure extend past the size specified, VSI SDL flags the overflow. The PL/I translation for the previous implicit union declaration example is as follows:

```
DCL 1 A BASED ,
     2 fill_0 union,
      3 B fixed binary(31),
      3 fill_1 ,
       4 bit_string1 bit(1),
       4 bit_string2 bit(4),
       4 fill_1$$v_fill_2 bit(3),
     2 last_item fixed binary(15);
```

The VSI SDL source file EXAMPLE.SDL in SDL$EXAMPLES shows an example of an implicit union declaration.

4.3.1.5. Implicit Union Declarations with the Optional DIMENSION Keyword

The following is an example of a VSI SDL structure defined with a data type and the DIMENSION keyword:

```
AGGREGATE fid STRUCTURE WORD DIMENSION 3;
     first WORD;
     second WORD;
     third WORD;
END fid;
```

In the previous example, a single structure is overlaid by an array of elements of the type specified by the structure type.

The **fid** structure is a three-word field that can be addressed as a single field or by each individual word so that it is easily defined in VSI SDL as represented above. The following is the PL/I translation for the VSI SDL source code in the previous example:

```
DCL 1 fill_0 union BASED ,
     2 fid (1:3) fixed binary(15),
     2 fill_1 ,
      3 first fixed binary(15),
      3 second fixed binary(15);
      3 third fixed binary(15);
```

The implicit union declaration allows structures to grow without the need to modify any trailing fillers. VSI SDL detects any overflow that may occur if the structure grows past the size of its data type. The size of the aggregate (in bytes) is equal to the size of the data type (in bytes) multiplied by the upper dimension (if any). If the size of the aggregate is greater than the sum of the size of all its members, VSI SDL still translates the declaration. However, if the size of the members exceeds the size of the aggregate, VSI SDL issues a message that has the following format:

```
%SDL-E-TOOMANYFIELDS, Structure fill_0 has too many fields [Line ?]
```

In the previous example, fill_0 is the VSI SDL-generated name for the first union aggregate.

4.3.1.6. Forcing Negative Offsets

The default origin of an aggregate is the beginning of the first aggregate member. You may specify the ORIGIN option on a level-1 AGGREGATE declaration to indicate that the origin is located at the beginning of any aggregate member. The resulting declaration forces all aggregate members declared before the specified origin to be located at negative offsets from the origin. For example:

```
AGGREGATE nodes STRUCTURE ORIGIN qflink;
   flink ADDRESS;
   blink ADDRESS;
   qflink ADDRESS;
   qblink ADDRESS;
END;
```

This declaration defines the origin of the structure nodes to be at the member `qflink`, so you may address `flink` and blink as negative offsets from `qflink`. Specifying an origin does not change the values of the current bit and byte offset symbols (^ and :). These are always calculated as being relative to the beginning of the aggregate.

4.3.1.7. Forcing Data Alignment

VSI SDL forces all AGGREGATE and subaggregate declarations to begin on byte boundaries. Thus, if an aggregate or subaggregate ends with BITFIELD declarations that do not end on byte boundaries, VSI SDL ensures that the next aggregate begins on a byte boundary by supplying BITFIELD declarations as fillers, if necessary.

When VSI SDL adds BITFIELD declarations, it determines the length required and provides a unique name of the form `string$V_FILL_n`. The string is the prefix supplied in the AGGREGATE declaration, or the aggregate name if no prefix was supplied. Within an aggregate, *n* begins at 0 and is incremented for each filler required.

The subaggregate declarations shown in the following example declare filler bitfields to force byte alignment at the end of each subaggregate; this programming practice makes it unnecessary for VSI SDL to perform the alignment.

```
AGGREGATE tree_node STRUCTURE;
  opcode WORD;
  lang_bits UNION;
   pli_bits STRUCTURE;
      resolved BITFIELD;
      psv BITFIELD;
      mark1 BITFIELD;
      spare_bits BITFIELD LENGTH 5;
      END pli_bits;
   c_bits STRUCTURE;
     value_variable_size BITFIELD;
     psv BITFIELD;
     expanded BITFIELD;
     resolved BITFIELD;
     reduced BITFIELD;
     spare_bits BITFIELD LENGTH 3;
     END c_bits;
   END lang_bits;
END tree_node;
```

The current bit offset is set at 0 at the beginning of each aggregate and is incremented by the bit lengths of each structure member in the aggregate at that level. In the following example, VSI SDL forces the structure flags to be aligned on a byte boundary.

```
AGGREGATE dcb STRUCTURE PREFIX dcb$;
  .
  .
  .
uflags STRUCTURE;
   context BITFIELD LENGTH 3;
   local BITFIELD;
   END uflags;
flags STRUCTURE;
   extern BITFIELD;
   relo BITFIELD;
   END flags;
```

The PL/I translation of the VSI SDL AGGREGATE declaration in the previous example is as follows:

```
%replace dcb$s_dcb by 2;
dcl 1 dcb based,
     2 dcb$r_uflags ,
      3 dcb$v_context bit(3),
      3 dcb$v_local bit(1),
      3 dcb$v_fill_0 bit(4),
     2 dcb$r_flags ,
      3 dcb$v_extern bit(1),
      3 dcb$v_relo bit(1),
      3 dcb$v_fill_1 bit(6);
```

The bit offsets of the members of the structures in the previous VSI SDL example are shown in the following table.

Member	**Bit Offset**
dcb$v_context	0
dcb$v_local	3

Member	Bit Offset
dcb$v_fill_0	4
dcb$v_extern	0
dcb$v_relo	1
dcb$v_fill_1	2

4.3.1.8. Using Offset Symbols

The period (.) represents the current byte offset from the origin in an AGGREGATE declaration. If the ORIGIN option is specified, the value of the period is equal to the byte offset from the member specified using the ORIGIN option. The current byte offset is useful for capturing the size of an aggregate or a portion of it. For example, in the declaration of a variable-length data structure, you can capture the size of the fixed-length portion.

The colon (:) represents the current byte offset relative to the first member in an AGGREGATE declaration. The value is not affected by the presence of an ORIGIN option.

The circumflex (^) represents the current bit offset relative to the most recently declared aggregate or byte-aligned element.

The following example shows the use of the byte offset symbols.

```
AGGREGATE operator STRUCTURE PREFIX opr_;
   flink ADDRESS;
   blink ADDRESS;
   opcount WORD;
   optype CHARACTER LENGTH 1;
   id WORD;
   flags STRUCTURE;
      is_constant_size BITFIELD LENGTH 1;
      is_terminator BITFIELD LENGTH 1;
      context BITFIELD LENGTH 3;
      filler BITFIELD LENGTH 8-^;
      END;
   #opsize = .;
   operands LONGWORD DIMENSION(#max_args);
   #instruction_size = :;
   END;
CONSTANT node_size EQUALS #opsize / 2;
CONSTANT inst_size EQUALS #instruction_size;
```

In the previous example, the local symbol `#opsize` captures the byte offset following the fixed-length portion of the structure operator. This value is subsequently used in the CONSTANT declaration, which defines the size of the constant portion of the structure in words.

The flags structure is byte aligned by the field filler, which makes use of the current bit offset symbol to compute the size of this field.

The CONSTANT `inst_size` defines the size of the entire structure.

4.3.2. AGGREGATE Format

An AGGREGATE declaration has the following syntax:

```
AGGREGATE aggregate-name { STRUCTURE } [ data-type ]
                         { UNION     }
                 [ COMMON ]
                 [ GLOBAL ]
                 [ TYPEDEF ]
                 [ BASED pointer-name ]
                 [ ALIGN ]
                 [ NOALIGN ]
                 [ BASEALIGN basealign-option ]
                 [ DIMENSION [ lbound: ] hbound ]
                 [ MARKER marker-string ]
                 [ PREFIX prefix-string ]
                 [ TAG tag-string ]
                 [ ORIGIN member-name ]
                 [ FILL ];
   aggregate-body
    .
    .
    .
END [ aggregate-name ];
```

AGGREGATE aggregate-name

Specifies any valid VSI SDL name used to identify the aggregate.

{STRUCTURE} {UNION}

Is the type of aggregate (see *Section 4.5, "ENTRY Declaration"*).

[data-type]

If specified, causes the AGGREGATE declaration to become an implicit union declaration (see *Section 4.5, "ENTRY Declaration"*).

[COMMON]
[GLOBAL]
[BASED pointer-name]
[TYPEDEF]

Is the storage class of the aggregate, if other than the default (BASED) (see *Section 3.2.2.4, "Storage Class Keywords"*). If an aggregate is the object of an ADDRESS declaration, it must have either the default or the BASED pointer-name storage class.

[ALIGN]
[NOALIGN]
[BASEALIGN basealign-option]

The ALIGN and NOALIGN keywords can be used to align (or de-align) the members of an aggregate on their natural boundary. The BASEALIGN keyword ensures that the size of an aggregate is a multiple of the given alignment. The BASEALIGN keyword therefore takes an argument, which specifies the alignment, either an expression in parenthesis or the name of a data type.

For example:

```
AGGREGATE MyStruct STRUCTURE ALIGN;
```

This aligns every member in the structure.

```
AGGREGATE MyStruct STRUCTURE NOALIGN;
```

No action will be taken to ensure that the members of the aggregate will be aligned.

```
AGGREGATE MyStruct STRUCTURE BASEALIGN (8);
```

The aggregate will be padded, so that in an array of elements of this aggregate, all elements will have a size that is a multiple of the given alignment (256, 2^8).

Alignment attributes on aggregates can be partially overridden by specifying alignment attributes on the members of the aggregate.

See also *Section 3.2.2.3, "Alignment Keywords"*.

[DIMENSION [lbound:]hbound]

Specifies that the aggregate is an array. If a single value is specified, that value indicates the number of elements in the array. Otherwise, lbound and hbound represent lower and upper bounds of the array, respectively.

[MARKER marker-string]

Specifies the prefix used to form the aggregate name. It may be any valid VSI SDL name with 0 to 32 characters, may or may not be enclosed in quotation marks (" "), and may be null.

[PREFIX prefix-string]

Specifies the prefix used in forming the names of aggregate members. It may be any valid VSI SDL name with 0 to 32 characters, may or may not be enclosed in quotation marks (" "), and may be null (see *Section 3.2.2.2, "PREFIX, MARKER, and TAG Keywords"*).

[TAG tag-string]

Specifies the tag used to form the aggregate name. The tag is appended to the prefix, if a prefix was specified. It can have 0 to 32 alphabetic or numeric characters; the tag must be enclosed in quotation marks (" ") if it begins with a numeric character (see *Section 3.2.2.2, "PREFIX, MARKER, and TAG Keywords"*).

[ORIGIN member-name]

Specifies the name of a member of this aggregate that is to be used as the origin of the aggregate.

Member declarations produce declarations of the members of an aggregate and have the following syntax:

```
member-name { data-type }           [ ALIGN ]
            { aggregate-name }      [ NOALIGN ]
            { user-type sizeopt }   [ BASEALIGN basealign-option ]
                                    [ DIMENSION [ lbound: ] hbound ]
                                    [ PREFIX prefix-string ]
                                    [ TAG tag-string ]
                                    [ FILL ];
```

member-name

Is any valid VSI SDL name used to identify the member.

{data-type}

Is any valid VSI SDL data type (see *Section 3.2.3, "Data Type Keywords"*).

{aggregate-name}

Is the name of the previously declared aggregate to be used as a type name. The name must be the full (VSI SDL-expanded) aggregate name, including the prefix and tag.

{user-type sizeopt}

Is a user-defined variable using the DECLARE statement's SIZEOF clause, shown and described in *Section 3.10, "DECLARE Statement"*.

[ALIGN]
[NOALIGN]
[BASEALIGN basealign-option]

Alignment attributes on a structure can be overridden with alignment attributes on a member declaration. For example:

```
StructMember1 LONGWORD SIGNED ALIGN;
```

This ensures that the member is aligned, even if it is within an aggregate that is not aligned.

```
StructMember1 LONGWORD SIGNED NOALIGN;
```

No action will be taken to ensure that this member will be aligned.

```
StructMember1 LONGWORD SIGNED BASEALIGN (4);
```

Here, StructMember1 will have an offset that is a multiple of 16 (2^4).

basealign-option can either be an expression in parentheses or the name of a data type.

See also *Section 3.2.2.3, "Alignment Keywords"*.

[DIMENSION [lbound:]hbound]

Specifies that the member is an array. If a single value is specified, that value indicates the number of elements in the array. Otherwise, lbound and hbound represent lower and upper bounds of the array, respectively (see *Section 3.2.2.7, "DIMENSION Keyword"*).

[PREFIX prefix-string]

Specifies the prefix used to form the member name. For subaggregates, the prefix is used to form the names of subaggregate members. It may be any valid VSI SDL name with 0 to 32 characters, may or may not be enclosed in quotation marks (" "), and may be null (see *Section 3.2.2.2, "PREFIX, MARKER, and TAG Keywords"*).

[TAG tag-string]

Specifies the tag used to form the member name. The tag is appended to the prefix, if a prefix was specified. It can have 0 to 32 alphabetic or numeric characters; the tag must be enclosed in quotation marks (" ") if it begins with a numeric character (see *Section 3.2.2.2, "PREFIX, MARKER, and TAG Keywords"*).

[FILL]

Indicates that the specified member or aggregate occurs only as a **fill** to force byte alignment on the following member or aggregate, respectively. In some languages, filler member and aggregate declarations do not appear in the output file.

aggregate-body

Is one or more of the following:

- Member declaration
- Subaggregate declaration
- CONSTANT declaration
- Local symbol assignment

END [aggregate-name]

Marks the end of the AGGREGATE or subaggregate declaration. The aggregate-name, if specified, must match the name on the most recently specified AGGREGATE or subaggregate declaration.

4.4. CONSTANT Declaration

The following sections describe the function and format of a CONSTANT declaration.

4.4.1. CONSTANT Description

A CONSTANT declaration generates a list of one or more named constants in the output file. You may specify a valid constant name or names and the constant values to be assigned to them. For example:

```
CONSTANT block_node_size EQUALS 24;
CONSTANT Strcon EQUALS STRING "This is a string constant" PREFIX Jg$
```

The first declaration creates the named constant `block_node_size` and assigns it the value 24. The second declaration creates the named string constant `Strcon` and assigns it the specified value.

The values of both declared constants (except string constants) and local symbols may be used in VSI SDL expressions. However, there is an important difference between declared constants and local symbols: declared constants are translated directly to the output file, whereas local symbols are not passed directly to the output file. For example, you can define the local symbol `#block_size` as follows:

```
#block_size = 24;
```

A subsequent CONSTANT declaration may refer to `#block_size` and use the value 24, as follows:

```
CONSTANT block_node_size EQUALS #block_size;
```

CONSTANT declarations (except string constants) can also be specified in a comma-delimited list, as follows:

```
CONSTANT
xyz EQUALS 10,
alpha EQUALS 0,
noname EQUALS 63;
```

To specify related constants with the same or incremental values, use a CONSTANT declaration with the INCREMENT option. In this form, the EQUALS expression gives the value to be assigned to the first named constant; values for subsequent constants are derived by incrementing the first value by the specified increment and assigning the result to the next name in the list. For example:

```
CONSTANT (
    bits,
    bytes,
    words,
    longs,
    quads,
    octas
    ) EQUALS 0 INCREMENT 1 PREFIX ctx$;
```

When VSI SDL assigns incremental values, it loops until values are assigned to all the names in a list. If there is no INCREMENT clause, the increment value is 0; thus, the same initial value is assigned to all the names. If names are omitted from a comma-delimited list, VSI SDL reserves the numbers that would be assigned to names in those positions. This lets you reserve numeric values for later assignment of names. For example:

```
CONSTANT
   bad_block,bad_data,,,,
   overlay,rewrite) EQUALS 0 INCREMENT 4;
```

In the previous example, VSI SDL assigns the values 0 and 4 to the names `bad_block` and `bad_data`, reserves the values 8, 12, and 16, and assigns the values 20 and 24 to the names `overlay` and `rewrite`.

The COUNTER option saves the last assigned value in a specified local symbol for subsequent use. For example:

```
CONSTANT (pli,c,bliss,macro)
   EQUALS 4 INCREMENT 4 PREFIX lang$
   COUNTER #lang;

 CONSTANT (basic,pascal,fortran)
   EQUALS #lang + 4 INCREMENT 4 PREFIX lang$;
```

The following table shows the constant names produced by these two declarations.

Constant Name	**Value**
lang$k_pli	4
lang$k_c	8
lang$k_bliss	12
lang$k_macro	16
lang$k_basic	20
lang$k_pascal	24
lang$k_fortran	28

You can comment individual declarations in a CONSTANT declaration list. For example:

```
CONSTANT(
     pli,    /* PL/I
```

```
     c,      /* C
     macro   /* MACRO-32
     ) EQUALS 4 INCREMENT 4 PREFIX lang$;
```

4.4.1.1. Defining Global Constants in VSI MACRO

VSI SDL does not directly generate definitions for global symbols (constants), but instead generates local constant definitions. The VSI MACRO back end, however, generates all declarations within macros so that they can be invoked with arguments that will produce global definitions.

VSI SDL also produces VSI MACRO output declarations that can generate global definitions when an invocation of the macro is assembled. In the following example, the VSI SDL module $IODEF contains the following constant declarations:

```
CONSTANT (
   nop,
   unload,
   loadmcode,
   seek,
   spacefile,
   startmproc,
   recal,
   stop,
   drvclr,
   initialize) EQUALS 0 INCREMENT 1 PREFIX "io$";
```

The VSI MACRO output generated by this declaration is a macro definition that must be invoked as follows in order to result in global rather than local constant definitions:

```
$IODEF <= =>
```

This argument results in global constant definitions rather than local constant definitions. To request that the locations of data fields be defined globally, add the argument <::> following <= =>.

4.4.2. CONSTANT Format

A CONSTANT declaration has the following syntax:

```
CONSTANT constant-name constant-class
       constant-class = { EQUALS expression numeric-options }
                        { EQUALS STRING string string-options }
       numeric-options = [ PREFIX prefix-string ]
                         [ TAG tag-string ]
                         [ COUNTER #local-name ]
                         [ TYPENAME type-name ] ;
       string-options = [ PREFIX prefix-string ]
                        [ TAG tag-string ] ;
CONSTANT (constant-name,...) EQUALS expression
                       [ INCREMENT expression ]
                       [ PREFIX prefix-string ]
                       [ TAG tag-string ]
                       [ COUNTER #local-name ]
                       [ TYPENAME type-name ];
CONSTANT (constant-name,...) EQUALS expression,
                       .
                       .
```

.

;

CONSTANT constant-name

Specifies any valid VSI SDL name used to identify the constant.

When more than one name is specified, separate the names with commas and enclose the list in parentheses.

EQUALS expression
EQUALS STRING string

Specifies the value to be assigned to the constant.

[PREFIX prefix-string]

Specifies the prefix used to form the constant name. It may be any valid VSI SDL name with 0 to 32 characters, may or may not be enclosed in quotation marks (" "), and may be null (see *Section 3.2.2.2, "PREFIX, MARKER, and TAG Keywords"*).

[TAG tag-string]

Specifies the tag used to form the constant name. The tag is appended to the prefix, if a prefix was specified. It can have 0 to 32 alphabetic or numeric characters; the tag must be enclosed in quotation marks (" ") if it begins with a numeric character (see *Section 3.2.2.2, "PREFIX, MARKER, and TAG Keywords"*).

[COUNTER #local-name]

Specifies the local symbol assigned to the last value that is assigned to a constant in the list.

[TYPENAME type-name]

Specifies a named data type that is not a VSI SDL data type.

(constant-name,...)

Is a list of valid names.

A constant-name in this list can be null, though the list itself cannot be null. If a member of the list is null, the corresponding value is reserved.

If INCREMENT is not specified, all names are assigned the value specified in the EQUALS expression.

[INCREMENT expression]

Specifies the value to be added to the EQUALS expression for each iteration of VSI SDL's generation of named CONSTANT declarations. It must be a valid VSI SDL integer expression. VSI SDL assigns the value of the EQUALS expression to the first constant name; the value of the INCREMENT expression is added to the EQUALS expression and assigned to the next constant name.

4.5. ENTRY Declaration

The following sections describe the function and format of the ENTRY declaration.

4.5.1. ENTRY Description

An ENTRY declaration produces an external procedure or function declaration in the output file. You must specify a valid entry name on the ENTRY declaration. You may also specify any or all of the ENTRY declaration options described in *Table 3.2, "Keywords Used in Declarations"* in *Section 3.2.2, "Declaration Modifier Keywords"* and shown within the context of an ENTRY declaration in *Section 4.5.1.1, "ENTRY Format"*.

The following example shows the ENTRY declaration for the OpenVMS system service SYS$GETJPI:

```
ENTRY SYS$GETJPI ALIAS $GETJPI PARAMETER (
 LONGWORD UNSIGNED VALUE NAMED EFN DEFAULT 0 TYPENAME EFNUM,
 LONGWORD UNSIGNED NAMED PIDADR IN OUT DEFAULT 0 TYPENAME PROCID,
 CHARACTER DESCRIPTOR NAMED PRCNAM IN DEFAULT 0 TYPENAME PROCNAME,
 ANY NAMED ITMLST IN TYPENAME ITEMLIST,
 QUADWORD UNSIGNED NAMED IOSB OUT DEFAULT 0 TYPENAME IOSB,
 ADDRESS(ENTRY) NAMED ASTADR DEFAULT 0 TYPENAME ASTADR,
 LONGWORD UNSIGNED VALUE NAMED ASTPRM DEFAULT 0 TYPENAME USERPARM )
RETURNS LONGWORD TYPENAME CONDVALU;
```

4.5.1.1. ENTRY Format

An ENTRY declaration has the following syntax:

```
 ENTRY entry-name [ ALIAS internal-name ]
 [ PARAMETER (param-desc,...) ]
 [ LINKAGE link-name ];
 [ VARIABLE ]
 [ RETURNS return-data-type [ NAMED param-name ] ]
 [ TYPENAME type-name ];
```

ENTRY entry-name

Specifies any valid VSI SDL name used to name the external entry point.

[ALIAS internal-name]

Specifies an alternate internal name used to designate the entry point.

[PARAMETER (param-desc,...)]

Describes the parameters of the external entry, if any. Param-desc must be specified as follows:

```
{ data-type      }       [ DESCRIPTOR ]       [ IN   ]
{ aggregate-name }       [ RTL_STR_DESC ]
                         [ VALUE ]            [ OUT ]
                         [ REFERENCE ]
                                              [ NAMED param-name ]
                                              [ DIMENSION expression ]
                                              [ DEFAULT constant-value ]
                                              [ TYPENAME type-name ]
                                              [ OPTIONAL ]
                                              [ LIST ]
```

{data-type}

Is any valid VSI SDL data type.

{aggregate-name}

Is the name of a previously declared aggregate that describes the data type of the parameter.

[DESCRIPTOR]

Indicates that the parameter is passed by descriptor.

[RTL_STR_DESC]

Indicates that the parameter can be passed by any of the classes of string descriptors. The data type must be CHARACTER. For example:

```
ENTRY xyz PARAMETER ( CHARACTER RTL_STR_DESC );
```

[VALUE]

Indicates that the parameter is passed by immediate value.

[REFERENCE]

Indicates that the parameter is passed by reference. REFERENCE is the default.

[IN]

Indicates an input parameter. IN is the default.

[OUT]

Indicates an output parameter.

[NAMED param-name]

Specifies the parameter name.

[DIMENSION expression]

Is the number of elements of an array parameter.

If **expression** is an asterisk (*), the number of dimensions depends on the dimensions of the actual parameter.

[DEFAULT constant-value]

Specifies a default value for a parameter. In languages supporting this option, the omission of an actual parameter is allowed.

[TYPENAME type-name]

Specifies a named data type that is not a VSI SDL data type.

[OPTIONAL]

Specifies that the parameter may or may not appear in the sequence of (keyword) parameters for a call using the entry point name. This is supported only in Ada, VSI BASIC, VSI BLISS, VSI MACRO, and PL/I output.

[LIST]

Indicates that the routine may be called with one or more parameters of the type being described. LIST may be specified only for the last parameter.

Note

All the PARAMETER options, if specified, must follow the data type or aggregatename in the declaration.

[LINKAGE link-name]

Specifies (for VSI MACRO only) that a special call macro (spelled *link-name*) will be used in the expansion of the entry macro.

[VARIABLE]

Indicates that not all parameters are described; that is, the entry point has a variable number of parameters and not all corresponding arguments need be present in the argument list when the entry point is invoked. See also the description of the LIST parameter option.

[RETURNS return-data-type [NAMED param-name]]

Specifies the data type and, optionally, the name of the parameter returned by the external entry, if it is a function.

The VOID keyword cannot be used in a PARAMETER clause.

```
return-data-type = { data-type }
                   { user-type sizeopt }
                   { VOID }
```

The argument *user-type sizeopt* specifies a user-defined type declared using the DECLARE statement's SIZEOF clause, shown and described in *Section 3.10, "DECLARE Statement"*.

The return type VOID indicates that the procedure returns no value.

[TYPENAME type-name]

Specifies a named data type for the return value that is not a VSI SDL data type.

Appendix A. VSI SDL Diagnostic Messages

This appendix summarizes the VSI SDL diagnostic messages. All messages indicating errors in VSI SDL syntax specify the line number in the VSI SDL source file at which the error occurred.

ABORT

Fatal: An internal error has occurred.

Description: Fatal internal error; unable to continue execution.

User Action: Please report the problem to VSI.

ADROBJBAS

Address object *object-name* must have based storage class [Line *n*]

Error: An address item is pointing to an aggregate that is not based.

User Action: Change the storage class of the aggregate to BASED.

BADNODETYPE

Internal node type is unknown for language *language-name*

Warning: A language backend has encountered a node type in the parsed SDL input that reflects a SDL element which the language does not support. For example, VSI DCL does not support aggregates or entries, only constants.

User Action: Depending on your needs, ignore the warning or change the SDL input file.

BASEALIGN

Invalid expression with BASEALIGN option. Value must be in range 0 to 124. *basealign-parameter* [Line *n*]

Error: The value for the /BASEALIGN qualifier is smaller than 0 or larger than 124.

User Action: Use a value in the range [0 ... 124].

BUGCHECK

Internal consistency failure [Line *n*] – please submit a bug report

Fatal: SDL has detected an internal error or inconsistency.

User Action: Please submit a bug report to VMS Software.

DIMENSIONSTAR

DIMENSION * for MEMBER "*member-name*" has no known discriminant [Line *n*]

Warning: The use of "DIMENSION *" within an aggregate is not allowed.

User Action: Use fixed values for "DIMENSION" within all aggregate.

DUPCONATT

Item *item-name* has duplicate or conflicting attributes [Line *n*]

Error: A declaration contains keywords that are not compatible.

User Action: Verify the syntax of the VSI SDL declaration, correct the declaration, and invoke VSI SDL again.

ERREXIT

Error exit

Fatal: Previous errors prevent continuation.

User Action: Correct the errors and invoke VSI SDL again.

FILFORMUNSUPP

RMS file format error reading intermediate file

Error: An intermediate file (.SDI) cannot be read due to wrong RMS attributes.

User Action: Make sure that the intermediate file is a sequential file, variable length, maximum 510 bytes, longest 510 bytes, carriage return carriage control.

IDENTGTR31

SDL-generated identifier longer than 31 characters exceeds capacity of *language-name* compiler [Line *n*]

Warning: The Pascal backend appends "$TYPE" to data types that contain "DEF". The resulting name then can exceed 31 characters, the maximum length of Pascal type names on OpenVMS.

User Action: Don’t use "DEF" in your type names and/or shorten your type names.

ILLFORWREF

Illegal forward reference for output language *language* [Line *n*]

Error: The specified output language does not allow forward referencing, or the language does not allow forward referencing in this context.

User Action: Correct or remove the forward reference.

IMMGTR32

Cannot pass values larger than 32 bits by immediate mechanism [Line *n*]

Warning: Using VALUE is invalid for this parameter because its size is greater than that of a longword. A reasonable translation was attempted, however.

User Action: Verify that the output file contains a satisfactory translation of the parameter description in your VSI SDL source file.

INCDEFSTRUC

Incompletely defined structure—*structure-name* [Line *n*]

Error: A structure name has been referenced before the structure has been completely defined.

User Action: Remove the reference and invoke VSI SDL again.

INFILOPN

Unable to open input file *file-spec*

Fatal: SDL cannot locate or open the SDL source file.

User Action: Verify that you correctly specified the name of the source file.

INFILSDI

File format error reading intermediate file *file-spec*. Possible version mismatch

Error: An intermediate file (.SDI) could not be read.

User Action: Check the spelling of the file name, existence and protection of the file.

INTOVF

Integer overflow in expression [Line *n*]

Error: Evaluation of an SDL expression resulted in a value that does not fit in a longword.

User Action: Correct the expression.

INVALIGN

Illegal value for **`/ALIGNMENT`** qualifier in command line

Error: The value of the **`/ALIGNMENT`** qualifier is not a positive number.

User Action: Use a positive number as value for the **`/ALIGNMENT`** qualifier.

INVBITFLD

Invalid bitfield *bitfield-name*—bitfields must be aggregate members [Line *n*]

Error: Bit fields must be members of aggregates. They cannot be scalar items.

User Action: Incorporate the BITFIELD declaration in an aggregate.

INVDECL

Invalid DECLARE for type *user-defined-name* [Line *n*]

Error: A DECLARE statement refers to a user defined data type that is invalid.

User Action: Correct the DECLARE statement.

INVEXPR

Invalid expression—cannot be resolved to a constant as required, *name-of-defineditem* [Line *n*]

Error: An non-constant expression has been used in a context which requires a constant expression.

User Action: Use a constant expression.

INVFLDSIZ

Item *item-name* has bit field or offset length greater than 32 [Line *n*]

Error: SDL cannot generate bit fields larger than 32 bits or cannot generate the proper bit mask.

User Action: Verify the BITFIELD declaration and correct it. If the BITFIELD declaration occurs within an aggregate and you specify the MASK option, verify that the bit offset of the start of the declaration plus the bit field size does not exceed 32 bits.

INVLISTOPT

Invalid use of LIST attribute—LIST may only appear on the last parameter. *entry-name* [Line *n*]

Error: The LIST attribute appears on a parameter other than the last.

User Action: Remove the LIST attribute.

INVNAME

Item name is invalid

Error: The item name contains illegal characters or is specified in an illegal context.

User Action: Correct or relocate the item name.

INVOUT

Invalid attributes for output language *language* [Line *n*]

Error: An SDL construct or data type is invalid for the specified target language.

User Action: Determine whether you specified the data type or VSI SDL declaration correctly, or whether you may be requesting language output that you do not require. Either correct the declaration or reissue the SDL command so that the indicated language output routine does not execute.

INVPARMTYP

Invalid parameter type for language *language* [Line *n*]

Error: A parameter specification is illegal for the specified language.

User Action: Modify the parameter specification and invoke VSI SDL again.

INVREQPARAM

Required parameter encountered after an optional parameter *parametername* [Line *n*]

Error: Required parameters must not follow optional parameters.

User Action: Correct the error and invoke VSI SDL again.

INVSHRIMG

Shareable image not found *specified-language*

Error: SDL cannot find the image to support the specified language.

User Action: Verify that the language image is located in the appropriate area. VSI

SDL searches for the image in SYS$SHARE. Language support images are of the form SDL $language.EXE, where language is an identifying character string for the language. For example, the VSI Fortran back end is SDL$FORTRAN.EXE, and the VSI MACRO back end is SDL $MACRO.EXE.

INVSYMDEF

Invalid symbol *symbol-name* specified in **/SYMBOLS** qualifier

Error: The value of the **/SYMBOLS** qualifier is not correct. The **/SYMBOLS** qualifier expects a list of symbol definitions in the form symbol-name:value or symbol-name=value. The error message indicates a missing comma, colon, or value.

User Action: Correct the syntax of the **/SYMBOLS** qualifier.

INVUNKLEN

Unknown length attribute valid only for parameter type [Line *n*]

Error: CHARACTER LENGTH * is only allowed in a parameter description. Specifying an unknown length for an ITEM or AGGREGATE member is an error.

User Action: Remove the LENGTH specification or replace the "*" with a valid expression.

LANGDUP

Language name *language-name* appears more than once in list [Line *n*]

Warning: In an IFLANGUAGE or END_IFLANGUAGE statement, the name of a language appears twice.

User Action: Remove the duplicate file name.

LANGMATCH

Language *language-name* does not appear in list of matching IF statement [Line *n*]

Warning: The list of language specified after the END_IFLANGUAGE keyword does not match the list of languages specified after the corresponding IFLANGUAGE keyword.

User Action: Correct the language list.

LANGMISS

Language *language-name* in list of matching IF statement missing from END list [Line *n*]

Warning: In an END_IFLANGUAGE statement, one of the languages from the IFLANGUAGE statement is missing.

User Action: Add the missing language to the END_IFLANGUAGE statement or remove all languages from the END_IFLANGUAGE statement.

LISFILOPN

Unable to open listing file *file-spec*

Error: SDL cannot open the indicated listing file.

User Action: Verify that you have write access to the directory to which the VSI SDL listing file is directed.

MASKTOOBIG

Cannot create correct mask value for *bitfield-name* [Line *n*]

Error: The MASK on a BITFIELD can only be used if the BITFIELD is part of a STRUCTURE which is part of a UNION, together with an integer field large enough to hold the mask.

User Action: Remove the MASK option, or add an integer field big enough so that it can hold all values for the MASK.

MATCHEND

End name does not match declaration name *name* [Line *n*]

Warning: The name specified on the END_MODULE or END delimiter does not match the most recent module name or aggregate name.

User Action: Verify that the spelling of the names specified on the END and END_ MODULE delimiters match. Check whether you have illegally nested MODULE declarations. This is only a warning message, but it may indicate an error.

MULTDEFSYM

Multiply defined symbol—*symbol-name* [Line *n*]

Error: A structure contains a duplicate symbol name.

User Action: Remove the duplicate name and invoke VSI SDL again.

NAMTRUNC

Generated name too long - truncated to 64 characters *name-to-be-truncated*

Warning: The BLISSF backend appends various prefixes and suffixes to names. The resulting name then can exceed 64 characters, the maximum length.

The DCL, TPU, and UIL backends limit the name of constants to 64.

User Action: Shorten the names.

NEGORIGIN

Aggregate *aggregate-name* has a negative origin - negative offset elements will be ignored [Line *n*]

Informational: The ORIGIN attribute defines a member as origin which is not at the beginning of the aggregate. This diagnostic will only be issued if the /VMS_DEVELOPMENT qualifier is present.

User Action: Specify the first member of the aggregate as origin.

NOOFFSET

Base offset for mask value for *bitfield-name* [Line *n*] cannot be determined

Error: The MASK on a BITFIELD can only be used if the BITFIELD is part of a STRUCTURE which is part of a UNION, together with an integer field large enough to hold the mask.

User Action: Remove the MASK option, or make the integer field big enough so that it can hold all values for the MASK.

NOOUTPUT

No language output produced

Warning: There were too many errors, or fatal errors, which prevented SDL from generating any output files.

User Action: Correct the errors indicated by the accompanying messages.

NULLSTRUC

Null structure *structure-name* has no members [Line *n*]

Error: An AGGREGATE or subaggregate declaration did not have any members.

User Action: Verify that the AGGREGATE or subaggregate declaration is correctly positioned in the file.

OFFSETEXPR

Offset or origin relative expression involves a forward or circular reference. *???-nod$t_name-???* [Line *n*]

Warning: The offset or origin cannot be calculated.

User Action: Correct the error and invoke VSI SDL again.

OUTFILOPN

Unable to open output file *file-spec*

Error: SDL cannot locate or open an SDL output file.

User Action: Verify that you correctly specified the name of the source file.

POSSCIRC

Possible circular definition for type *???-nod$t_naked-???* [Line *n*]

Informational: The definition for the type cannot be processed.

User Action: Correct the error and invoke VSI SDL again.

REVCHECK

Front-end / back-end version mismatch. Check installation.

Fatal: A language backend has a different version than the calling frontend.

User Action: Check your VSI SDL installation.

SIZENEST

Illegal nesting of SIZEOF clauses (Item *item-name*) [Line *n*]

Error: SIZEOF clauses cannot be nested.

User Action: Remove the nested SIZEOF clause.

SIZEQUAL

Item *item-name*, an aggregate, cannot be qualified by SIZEOF [Line *n*]

Error: The SIZEOF clause is not allowed in this context.

User Action: Remove the SIZEOF clause.

SIZEREDEF

Size or type of item *item-name* redefined [Line *n*]

Error: SDL has detected a redefinition of the size or data type of the specified item.

User Action: Remove the clause causing the redefinition.

STRINGCONST

String constant *item-name* used in arithmetic expression [Line *n*]

Error: A reference to a string constant is not allowed in the context of an arithmetic expression.

User Action: Remove the string constant reference.

SYMALRDEF

Symbol *symbol-name* was already defined in command line

Error: The value of the **/SYMBOLS** qualifier contains a symbol name more than once.

User Action: Remove the duplicate name.

SYMNOTDEF

Symbol *symbol-name* was not defined in command line, value zero assumed [Line *n*]

Warning: A symbol is used in an IFSYMBOL or ELSE_IFSYMBOL statement, that has not been defined using the **/SYMBOLS** qualifier.

User Action: Define the symbol using the **/SYMBOLS** qualifier.

SYMTABOVR

Symbol table overflow

Fatal: SDL exceeded its symbol table space.

User Action: Reduce the size or complexity of the VSI SDL source file; if possible, separate the file into several different files or modules.

SYNTAXERR

Syntax error [Line *n*]

Error: The SDL translator detected a syntax error. This message is accompanied by a message indicating the type of error and tells you what type of token or keyword SDL expected but did not find.

User Action: Determine the syntax error from the accompanying message and correct it.

TOKOVF

Token exceeds maximum size of *maximum-token-length* [Line *n*]

Error: A line in the VSI SDL source file is longer than the maximum length.

User Action: Shorten the offending line.

TOOMANYFIELDS

Structure *structure-name* has too many fields [Line *n*]

Error: The structure has too many fields.

User Action: Simplify the structure.

TYPNAM

Aggregate type name not supported [Line *n*]

Warning: An illegal aggregate name has been used.

User Action: Choose another aggregate name.

TYPNOTSUP

Output language does not support data type *data-type-name* [Line *n*].

Warning: The specified data type is not supported by the output language. A reasonable translation was attempted, however.

User Action: Verify that the output file contains a satisfactory translation of the data type you specified in your VSI SDL source file.

UNALIGNED

member-name does not align on its natural boundary [Line *n*]

Warning: A member does not fall on its natural alignment (if **/CHECK_ALIGNMENT** is present on the command line) or on the alignment specified with the **/ALIGNMENT** qualifier.

User Action: Check the layout of the aggregate in question.

UNDEFCON

Undefined constant name *constant-name* used in expression [Line *n*]

Error: In the definition of a CONSTANT, an undefined name has been used.

User Action: Verify the spelling of the name.

UNDEFFIL

Unable to open include file *file-name* [Line *n*]

Error: A file to be INCLUDEd could not be opened.

User Action: Check the spelling of the file name, existence and protection of the file.

UNDEFORG

Definition of ORIGIN name *member-name* not found in aggregate [Line *n*]

Error: The member used as argument to the ORIGIN attribute was not found within the aggregate.

User Action: Verify the spelling of the member name.

UNDEFSYM

Undefined local symbol *symbol-name* used in expression [Line *n*]

Error: A name preceded by a pound sign (#) is not defined.

User Action: Verify that the local symbol name is spelled correctly and that it appears before its reference in the VSI SDL source file.

UNDEFUSER

Undefined user type name *type-name* referenced [Line *n*]

Error: A DECLARE statement refers to a data type that is neither a builtin nor a known user defined data type.

User Action: Check the spelling of the data type referenced.

WARNEXIT

Warning exit

Warning: A warning message has been issued.

User Action: Output can be compiled, but the results may be unexpected.

ZERODIV

Zero divide in expression [Line *n*]

Error: An expression specified in an SDL declaration resulted in a divide-by-zero exception condition.

User Action: Verify the expression and correct it.

ZEROLEN

Item *item-name* has 0 or negative length [Line *n*]

Warning: A BITFIELD or CHARACTER declaration or a DIMENSION option specified a length of 0 or less.

User Action: Correct the declaration. If the length or bound value was specified using a VSI SDL expression, verify the local symbol values and the results of arithmetic operations in the expression, if any.

Appendix B. VSI SDL Language Translation Summaries

This appendix shows the translation summaries of VSI SDL language elements to their corresponding output in each of the following supported languages:

- Ada
- VSI BASIC
- VSI BLISS
- VSI C
- VSI Datatrieve
- VSI DCL
- VSI Fortran
- VSI MACRO
- VSI Pascal
- PL/I

B.1. Ada Translation Summary

The following table shows the VSI SDL to Ada language translation summary.

VSI SDL Declaration	**Ada Output**
MODULE name IDENT string	- - module name IDENT string
/* comment	- - comment
CONSTANT x	
EQUALS n;	x : constant := n ;
EQUALS STRING "s";	x : constant STRING := "s";
ENTRY name	procedure name (parameter-list ...); pragma INTERFACE (EXTERNAL, name); pragma IMPORT_VALUED_PROCEDURE (name,"name",) (type,...), (passing-mechanism,...);
PARAMETER (type,...)	parameter-name : type; .

VSI SDL Declaration	Ada Output
	. .
ANY	UNSIGNED_LONGWORD ;
DESCRIPTOR	Generates the DESCRIPTOR passing mechanism name in the IMPORT_VALUED_PROCEDURE pragma for the parameter.
RTL_STR_DESC	Generates the DESCRIPTOR passing mechanism name in the IMPORT_VALUED_PROCEDURE pragma for the parameter.
IN	Causes the IN parameter mode to appear on the formal parameter.
OUT	Causes the OUT parameter mode to appear on the formal parameter.
NAMED param-name	Parameter name. If none exists, names will be generated of the form PARAMETER_1 ... PARAMETER_n.
VALUE	Generates the VALUE passing mechanism name in the IMPORT_VALUED_PROCEDURE pragma for the parameter.
REFERENCE	Generates the REFERENCE passing mechanism name in the IMPORT_VALUED_PROCEDURE pragma for the parameter.
DEFAULT n	If DEFAULT = 0, see the table at the end of this section for the initial values of each VSI SDL data type. If the parameter contains the VALUE attribute and the default value is other than 0, that := value is added to the parameter.
LIST	Generates 10 formal parameters of the type of the parameter being described. If optional is not specified, all but the first of the corresponding parameter specifications contain a default expression of the form type'NULL_PARAMETER. If optional is specified, all 10 parameter specifications contain the default expression. An appropriate entry for FIRST_OPTIONAL_ PARAMETER is generated.
OPTIONAL	Generates a parameter specification containing a default expression of the form type'NULL_PARAMETER. If mode is OUT (or IN OUT), an additional parameter specification is generated to allow for use as an output parameter. An appropriate entry for FIRST_OPTIONAL_PARAMETER is generated.
TYPENAME type-name	Specifies a non-SDL keyword data type to be used as the parameter type.
RETURNS return-data-type	The first parameter in the parameter list is the return parameter. A comment of the form `- - return value` is placed beside the return parameter.
NAMED param-name	Specifies the name of a parameter in the IMPORT_VALUED_ PROCEDURE argument list that receives the return value.
VARIABLE	n/a

VSI SDL Declaration	Ada Output
ALIAS internal-name	The internal-name is used instead of the entry name as the identifier associated with the entry point.
LINKAGE	n/a
TYPENAME type-name	Specifies a non-SDL keyword data type to be used as the entry type.
STRUCTURE	An Ada record type is generated as well as a record representation clause and an initialization constant for the record. If any substructures exist, the record types for those structures are generated first. Following this table, see the note on structures for a discussion of structure to Ada record translation.
UNION	An Ada record type is generated depending on the union. See the notes following this table for a discussion of union translations.
BYTE [SIGNED]	INTEGER_8
INTEGER_BYTE [SIGNED]	INTEGER_8
WORD [SIGNED]	INTEGER_16
INTEGER_WORD [SIGNED]	INTEGER_16
LONGWORD [SIGNED]	INTEGER_32
INTEGER_LONG [SIGNED]	INTEGER_32
INTEGER [SIGNED]	INTEGER_32
QUADWORD [SIGNED]	UNSIGNED_QUADWORD
INTEGER_QUAD [SIGNED]	UNSIGNED_QUADWORD
INTEGER_HW [SIGNED]	INTEGER_64 for /ALPHA_AXP INTEGER_32 for /VAX
HARDWARE_INTEGER [SIGNED]	INTEGER_64 for /ALPHA_AXP INTEGER_32 for /VAX
OCTAWORD	UNSIGNED_LONGWORD_ARRAY(0 .. 3)
BYTE UNSIGNED	UNSIGNED_BYTE
INTEGER_BYTE UNSIGNED	UNSIGNED_BYTE
WORD UNSIGNED	UNSIGNED_WORD
INTEGER_WORD UNSIGNED	UNSIGNED_WORD
LONGWORD	UNSIGNED_LONGWORD
INTEGER_LONG UNSIGNED	UNSIGNED_LONGWORD
INTEGER UNSIGNED	UNSIGNED_LONGWORD
QUADWORD UNSIGNED	UNSIGNED_QUADWORD

VSI SDL Declaration	Ada Output
INTEGER_QUAD UNSIGNED	UNSIGNED_QUADWORD
INTEGER_HW UNSIGNED	UNSIGNED_QUADWORD for /ALPHA_AXP UNSIGNED_LONGWORD for /VAX
HARDWARE_INTEGER UNSIGNED	UNSIGNED_QUADWORD for /ALPHA_AXP UNSIGNED_LONGWORD for /VAX
OCTAWORD UNSIGNED	UNSIGNED_LONGWORD_ARRAY(0 .. 3)
F_FLOATING	F_FLOAT
D_FLOATING	D_FLOAT
G_FLOATING	G_FLOAT
H_FLOATING	LONG_LONG_FLOAT
F_FLOATING COMPLEX	F_FLOATING_COMPLEX
D_FLOATING COMPLEX	D_FLOATING_COMPLEX
G_FLOATING COMPLEX	G_FLOATING_COMPLEX
H_FLOATING COMPLEX	H_FLOATING_COMPLEX
DECIMAL PRECISION (p,q)	This data type has not yet been implemented. A comment appears in the output, and a message is printed to this effect.
BITFIELD	BOOLEAN (one bit)
LENGTH n	UNSIGNED_n if n > 1; BOOLEAN if n = 1 for /ALPHA_AXP BIT_ARRAY (0..n-1) for /VAX
MASK	prefix_M_name : constant : = 16#mask-value#;
SIGNED	n/a
CHARACTER	CHARACTER
LENGTH n	STRING(1 .. n) if n > 1
LENGTH *	STRING
VARYING	UNSIGNED_WORD field for the string length is generated if VARYING is specified; n/a in parameter context
ADDRESS	ADDRESS
POINTER	ADDRESS
POINTER_LONG	ADDRESS
POINTER_HW	UNSIGNED_QUADWORD for /ALPHA_AXP ADDRESS for /VAX
HARDWARE_ADDRESS	UNSIGNED_QUADWORD for /ALPHA_AXP ADDRESS for /VAX

VSI SDL Declaration	Ada Output
POINTER_QUAD	UNSIGNED_QUADWORD
BOOLEAN	BOOLEAN
user-type-name	user-type-name_TYPE
Default storage class	n/a
COMMON storage class	n/a
GLOBAL storage class	n/a
BASED pointer-name	n/a
TYPEDEF	For an ITEM, a subtype definition is generated. For an AGGREGATE, the behavior is as though TYPEDEF had not been specified.
DIMENSION [lbound]:hbound	If other than a BITFIELD data type, append `_ARRAY (lbound .. hbound)` to the type. For BITFIELDS, a BIT_ARRAY is generated of the form `BIT_ ARRAY(0 .. hbound*length-1)`. If lbound is not supplied, it defaults to 1.
ORIGIN member-name	n/a

Note

1. **Ada Names**

   Dollar signs ($) are illegal in Ada names and are replaced with underscores (_).

2. **Union Criteria**

   If the union has the following form, it is treated as a record type:

```
A ... union;
ITEM_1 ...;           /* First member is not a union or
                         a structure, and the size of the
                         first member is the same as the
                         size of the union.
ITEM_2 structure ...; /* Second member is a structure,
                         and the size of the first member
                         is the same as the size of the
                         union. (Needed so size based
                         on structure components is
                         correct.)
            I_2_A ...;
            I_2_B ...;
            ...
            end ITEM_2;
            ITEM_3 ...;
            ...
            end A;
```

   If the union does not have the form shown in the previous example, the union is ignored and only the first member is used.

```
Example 1 - Criteria are satisfied:
---------
     aggregate X prefix XXX$;
         X_1 longword unsigned;
         X_2_OVERLAY union;
              X_2 longword unsigned;
              X_2_FIELDS structure;
                  X_2_B_1 byte unsigned;
                  X_2_B_2 byte unsigned;
end X_2_FIELDS;
X_2_C word unsigned;
X_2_D structure;
                  X_2_D_1 bitfield(3);
                  X_2_D_2 bitfield;
              end X_2_D;
        end X_2_OVERLAY;
end X;
```

The union `X_2_OVERLAY` satisfies the criteria in the previous example and would be treated as a record declaration. Source code similar to the following would be generated:

```
type XXX_X_2_TYPE is record
    X_2_B_1 : BYTE;
    X_2_B_2 : BYTE;
    end record;
type XXX_X_2_D_TYPE is record
    X_2_D_1 : FLAGS(1 .. 3);
    X_2_D_2 : BOOLEAN;
    end record;
type XXX_TYPE is record
    X_1 : LONGWORD;
    X_2 : XXX_X_2_TYPE;
----X_2_C overlaps X_2
----
----X_2_C : UNSIGNED_16;
----X_2_D overlaps X_2
----
----X_2_D : XXX_X_2_D_TYPE;
    end record;
Example 2 - Criteria are not satisfied
---------
   aggregate Y prefix YYY$;
    Y_1 longword unsigned;
    Y_2_OVERLAY union;
        Y_2_A byte unsigned;
        Y_2_B byte unsigned;
    end Y_2_OVERLAY;
   end Y;
```

In the previous example, the second member is not a structure, and the following source code would be generated:

```
type YYY_TYPE is record
     Y_1 : LONGWORD;
     Y_2_A : BYTE;
----Y_2_B overlaps Y_2_A
----
```

```
----Y_2_B : BYTE;
     end record;
```

3. **IN and OUT Parameter Modes**

   If the mode is not explicitly given for a parameter, the default mode (IN) appears.

4. **OPTIONAL Parameters**

   An entry for FIRST_OPTIONAL_PARAMETER is generated, naming the first trailing optional parameter, if one exists. The appropriate signatures are generated for OPTIONAL OUT (or IN OUT) parameters to allow for any valid combination of actual parameters to be specified.

5. **DEFAULT Values**

   DEFAULT generates a default expression in the parameter specification.

   The following is a list of VSI SDL data types and the default Ada values for the INITIALIZATION constant.

VSI SDL Data Type	**Ada INITIALIZATION**
BYTE [SIGNED]	0
INTEGER_BYTE [SIGNED]	INTEGER_8_ZERO
WORD [SIGNED]	0
INTEGER_WORD [SIGNED]	INTEGER_16_ZERO
LONGWORD [SIGNED	0
INTEGER_LONG [SIGNED]	INTEGER_32_ZERO
INTEGER [SIGNED]	INTEGER_32_ZERO
INTEGER_HW [SIGNED]	INTEGER_64_ZERO for /ALPHA_AXP INTEGER_32_ZERO for /VAX
HARDWARE_INTEGER [SIGNED]	INTEGER_64_ZERO for /ALPHA_AXP INTEGER_32_ZERO for /VAX
QUADWORD [SIGNED]	(0, 0)
INTEGER_QUAD [SIGNED]	UNSIGNED_QUADWORD_ZERO
OCTAWORD [SIGNED]	(0, 0, 0, 0)
BYTE UNSIGNED	0
INTEGER_BYTE UNSIGNED	UNSIGNED_BYTE_ZERO
WORD UNSIGNED	0
INTEGER_WORD UNSIGNED	UNSIGNED_WORD_ZERO
LONGWORD UNSIGNED	0
INTEGER_LONG UNSIGNED	UNSIGNED_LONGWORD_ZERO
INTEGER UNSIGNED	UNSIGNED_LONGWORD_ZERO
INTEGER_HW UNSIGNED	UNSIGNED_QUADWORD_ZERO for /ALPHA_AXP

VSI SDL Data Type	Ada INITIALIZATION
	UNSIGNED_LONGWORD_ZERO for /VAX
HARDWARE_INTEGER UNSIGNED	UNSIGNED_ZERO for /ALPHA_AXP UNSIGNED_LONGWORD_ZERO for /VAX
QUADWORD UNSIGNED	(0, 0)
INTEGER_QUAD UNSIGNED	UNSIGNED_QUADWORD_ZERO
OCTAWORD [SIGNED]	(0, 0, 0, 0)
CHARACTER	ASCII.NUL
DECIMAL PRECISION (p,s)	NYI_PACKED_DECIMAL_ZERO
F_FLOATING	0.0
F_FLOATING_ COMPLEX	F_FLOATING_COMPLEX_ZERO
D_FLOATING	0.0
D_FLOATING_ COMPLEX	D_FLOATING_COMPLEX_ZERO
G_FLOATING	0.0
G_FLOATING_ COMPLEX	G_FLOATING_COMPLEX_ZERO
H_FLOATING	0.0
H_FLOATING_ COMPLEX	H_FLOATING_COMPLEX_ZERO
ADDRESS	ADDRESS_ZERO
POINTER	ADDRESS_ZERO
POINTER_LONG	ADDRESS_ZERO
POINTER_HW	UNSIGNED_QUADWORD_ZERO for /ALPHA_AXP ADDRESS_ZERO for /VAX
HARDWARE_ ADDRESS	UNSIGNED_QUADWORD_ZERO for /ALPHA ADDRESS_ZERO for /VAX
POINTER_QUAD	(0, 0)

6. **Structure of Ada Record Translation**

   For each VSI SDL structure, the corresponding Ada translation consists of three parts: 1) a record type definition, 2) a record representation clause, and 3) an initialization constant for the record. The three parts have the following form:

```
type structure-name is
   record
      member-name : type; ...
   end record;
   for structure-name use
      record
      member-name at offset range 0 .. size;
      ...
   end record
  for structure-name'size use total-size;
 structure-name_INIT : constant structure-name :=
  (member-name => initial_value,
   ...)
```

Size is the size of the field in bits, and total_size is the size of the entire record in bits.

B.2. VSI BASIC Translation Summary

The following table shows the VSI SDL to VSI BASIC language translation summary.

VSI SDL Declaration	**VSI BASIC Output**
MODULE name IDENT string	!*** MODULE name IDENT string ***
/* comment	! comment
CONSTANT x	
EQUALS n;	DECLARE LONG CONSTANT x = n
EQUALS STRING "s";	DECLARE STRING CONSTANT x = "s"
ENTRY name	EXTERNAL SUB name
PARAMETER (type,...)	(& type, & . .)
ANY	ANY
DESCRIPTOR	BY DESC
RTL_STR_DESC	BY DESC
IN	n/a
OUT	n/a
NAMED param-name	n/a
VALUE	BY VALUE
REFERENCE	BY REF
DEFAULT n	n/a

VSI SDL Declaration	VSI BASIC Output
LIST	Yields one explicit parameter description plus m-n commas, where m is the maximum number of parameters allowable in VSI BASIC, and n is the number of parameters already specified (including the LIST parameter description).
OPTIONAL	OPTIONAL data-type
TYPENAME type-name	n/a
RETURNS return-data-type	EXTERNAL data-type FUNCTION name
NAMED param-name	n/a
VARIABLE	n/a
ALIAS	n/a
LINKAGE	n/a
TYPENAME type-name	n/a
STRUCTURE	DECLARE LONG CONSTANT markerS_structure-name = size RECORD struct-name type member-name . . . END RECORD struct-name
UNION	DECLARE LONG CONSTANT markerS_union-name = size RECORD union-name VARIANT CASE type member-name . . . END VARIANT END RECORD union-name
BYTE [SIGNED]	BYTE
INTEGER_BYTE [SIGNED]	BYTE

VSI SDL Declaration	VSI BASIC Output
WORD [SIGNED]	WORD
INTEGER_WORD [SIGNED]	WORD
LONGWORD [SIGNED]	LONG
INTEGER_LONG [SIGNED]	LONG
INTEGER [SIGNED]	LONG
INTEGER_HW [SIGNED]	BASIC$QUADWORD for /ALPHA_AXP LONG for /VAX
HARDWARE_INTEGER [SIGNED]	BASIC$QUADWORD for /ALPHA_AXP LONG for /VAX
QUADWORD [SIGNED]	BASIC$QUADWORD
INTEGER_QUAD [SIGNED]	BASIC$QUADWORD
OCTAWORD	BASIC$OCTAWORD
BYTE UNSIGNED	BYTE
INTEGER_BYTE UNSIGNED	BYTE
WORD UNSIGNED	WORD
INTEGER_WORD UNSIGNED	WORD
LONGWORD UNSIGNED	LONG
INTEGER_LONG UNSIGNED	LONG
INTEGER UNSIGNED	LONG
QUADWORD UNSIGNED	BASIC$QUADWORD
INTEGER_QUAD UNSIGNED	BASIC$QUADWORD
INTEGER_HW UNSIGNED	BASIC$QUADWORD for /ALPHA_AXP LONG for /VAX
HARDWARE_INTEGER UNSIGNED	BASIC$QUADWORD for /ALPHA_AXP LONG for /VAX
OCTAWORD UNSIGNED	BASIC$OCTAWORD for /ALPHA_AXP LONG for /VAX
F_FLOATING	SINGLE
D_FLOATING	DOUBLE
G_FLOATING	GFLOAT
H_FLOATING	BASIC$HFLOAT_AXP for /ALPHA_AXP HFLOAT for /VAX
F_FLOATING COMPLEX	BASIC$F_FLOATING_COMPLEX

VSI SDL Declaration	VSI BASIC Output
D_FLOATING COMPLEX	BASIC$D_FLOATING_COMPLEX
G_FLOATING COMPLEX	BASIC$G_FLOATING_COMPLEX
H_FLOATING COMPLEX	BASIC$H_FLOATING_COMPLEX_AXP for /ALPHA_AXP BASIC$H_FLOATING_COMPLEX for /VAX
DECIMAL PRECISION (p,q)	DECIMAL (p,q)
BITFIELD	type name_bits ! COMMENT ADDED BY SDL name_bits contains bits name1 through name2
LENGTH n	n is used in computing size (type) of name_bits field.
MASK	DECLARE LONG CONSTANT prefix$m_name = x'mask-value'
SIGNED	n/a
CHARACTER	STRING name = 1
LENGTH n	STRING name = n
LENGTH *	STRING
VARYING	group name WORD str-len STRING str-text = n, where n is the maximum length of the string end group name
ADDRESS	LONG name
POINTER	LONG name
POINTER_LONG	LONG name
POINTER_HW	BASIC$QUADWORD name for /ALPHA_AXP LONG name for /VAX
HARDWARE_ADDRESS	BASIC$QUADWORD name for /ALPHA_AXP LONG name for /VAX
POINTER_QUAD	BASIC$QUADWORD name
BOOLEAN	BYTE name
user-type-name	user_type_name
COMMON storage class	COMMON name name
GLOBAL storage class	EXTERNAL name name
BASED pointer-name	n/a

VSI SDL Declaration	VSI BASIC Output
TYPEDEF	For an ITEM, a record definition is generated. For an AGGREGATE, the behavior is as though TYPEDEF had not been specified.
DIMENSION [lbound:]hbound	type name (lbound to hbound); if lbound was not specified in VSI SDL, lbound is 1 in the VSI BASIC output
ORIGIN member-name	n/a

B.3. VSI BLISS Translation Summary

The following table shows the VSI SDL to VSI BLISS language translation summary.

VSI SDL Declaration	VSI BLISS Output
MODULE name IDENT string	!*** MODULE name IDENT string ***
/* comment	! comment
CONSTANT x	
EQUALS n;	LITERAL x = n;
EQUALS STRING "s";	macro x = ’s’%;
ENTRY name	EXTERNAL ROUTINE name EXTERNAL ROUTINE name: NOVALUE ;
PARAMETER (type,...)	n/a
ANY	n/a
DESCRIPTOR	n/a
RTL_STR_DESC	n/a
IN	n/a
OUT	n/a
NAMED param-name	n/a
VALUE	n/a
REFERENCE	n/a
DEFAULT n	Used in KEYWORDMACRO when the **/VMS_DEVELOPMENT** qualifier is specified
LIST	Generates invocation of special predefined macro to handle LIST parameters
OPTIONAL	Generates invocation of special predefined macro to handle OPTIONAL parameters when the **/VMS_DEVELOPMENT** qualifier is specified
TYPENAME type-name	n/a
RETURNS return-data-type	n/a
NAMED param-name	n/a

VSI SDL Declaration	VSI BLISS Output
VARIABLE	n/a
ALIAS internal-name	When the **/VMS_DEVELOPMENT** qualifier is specified, internal-name becomes the name of the VSI BLISS KEYWORDMACRO generated for the ENTRY declaration.
LINKAGE	n/a
TYPENAME type-name	n/a

STRUCTURE and UNION	Each aggregate or member declaration in VSI SDL produces a VSI BLISS macro declaration of the form:
	
	MACRO name = **off,pos,size,ext %;**
	
	The BLISSF back end translates each aggregate and associated member declaration in VSI SDL to a VSI BLISS field declaration. The field-set-name is of the form prefix $name_FIELDSET, where prefix is declared for the entire aggregate and name is the aggregate name.
	
	`FIELD prefix$name_FIELDSET=`
	`SET`
	`member-name = [off,pos,size,ext];`
	`.`
	`.`
	`.`
	`TES;`
	`literal prefix$s_name = size;`
	`MACRO prefix$r_name = BLOCK`
	`[pref$_name,BYTE]`
	`FIELD`
	`(prefix$name_FIELDSET)%;`
	
	● off
	
	Byte offset of this aggregate or item within the current aggregate
	
	● pos
	
	Bit position from the offset
	
	● size
	
	Size of the aggregate of item, in bits, if the size is 4 bytes or less. Otherwise, this field contains 0, and VSI SDL generates the size declaration.
	
	● ext
	
	Contains 0 if the value is zero extended, or 1 if the value is sign extended or SIGNED bit.
	
	The following VSI SDL data types generate field specifications of the form "off,pos,size,ext" in various

VSI SDL Declaration	VSI BLISS Output
	contexts. Where necessary, constants indicating the size of the data type (field) are also generated.
BYTE [SIGNED]	*off,pos*,8,1
INTEGER_BYTE [SIGNED]	*off,pos*,8,1
WORD [SIGNED]	*off,pos*,16,1
INTEGER_WORD [SIGNED]	*off,pos*,16,1
LONGWORD [SIGNED]	*off,pos*,32,1
INTEGER_LONG [SIGNED]	*off,pos*,32,1
INTEGER [SIGNED]	*off,pos*,32,1
INTEGER_HW [SIGNED]	*off,pos*,0,1 for /ALPHA_AXP *off,pos*,64,1 for /ALPHA_AXP/B64 *off,pos*,32,1 for /VAX
HARDWARE_INTEGER [SIGNED]	*off,pos*,0,1 for /ALPHA_AXP *off,pos*,64,1 for /ALPHA_AXP/B64 *off,pos*,32,1 for /VAX
QUADWORD [SIGNED]	*off,pos*,0,1 for /ALPHA_AXP *off,pos*,64,1 for /ALPHA_AXP/B64 *off,pos*,0,1 for /VAX
INTEGER_QUAD [SIGNED]	*off,pos*,0,1 for /ALPHA_AXP *off,pos*,64,1 for /ALPHA_AXP/B64 *off,pos*,0,1 for /VAX
OCTAWORD [SIGNED]	*off,pos*,0,1
BYTE UNSIGNED	*off,pos*,8,0
INTEGER_BYTE UNSIGNED	*off,pos*,8,0
WORD UNSIGNED	*off,pos*,16,0
INTEGER_WORD UNSIGNED	*off,pos*,16,0
LONGWORD UNSIGNED	*off,pos*,32,0
INTEGER_LONG UNSIGNED	*off,pos*,32,0
INTEGER UNSIGNED	*off,pos*,32,0
INTEGER_HW UNSIGNED	*off,pos*,0,0 for /ALPHA_AXP *off,pos*,64,0 for /ALPHA_AXP/B64 *off,pos*,32,0 for /VAX
HARDWARE_INTEGER UNSIGNED	*off,pos*,0,0 for /ALPHA_AXP *off,pos*,64,0 for /ALPHA_AXP/B64 *off,pos*,32,0 for /VAX
QUADWORD [SIGNED]	*off,pos*,0,0 for /ALPHA_AXP

VSI SDL Declaration	VSI BLISS Output
	off,pos,64,0 for /ALPHA_AXP/B64 *off,pos*,0,0 for /VAX
INTEGER_QUAD [SIGNED]	*off,pos*,0,0 for /ALPHA_AXP *off,pos*,64,0 for /ALPHA_AXP/B64 *off,pos*,0,0 for /VAX
OCTAWORD UNSIGNED	*off,pos*,0,0
F_FLOATING	*off,pos*,32,0
D_FLOATING	*off,pos*,0,0 for /ALPHA_AXP *off,pos*,64,0 for /ALPHA_AXP/B64 *off,pos*,0,0 for /VAX literal prefix$s_name = 8;
G_FLOATING	*off,pos*,0,0 for /ALPHA_AXP *off,pos*,64,0 for /ALPHA_AXP/B64 *off,pos*,0,0 for /VAX literal prefix$s_name = 8;
H_FLOATING	*off,pos*,0,0 literal prefix$s_name = 16; If you specify **/LANGUAGES=BLISSF**, a special set of macros and FIELDSETs, which define the particular COMPLEX data type and its real and imaginary components, are generated.
F_FLOATING COMPLEX	*off,pos*,0,0 for /ALPHA_AXP
COMPLEX	*off,pos*,64,0 for /ALPHA_AXP/B64
COMPLEX	*off,pos*,0,0 for /VAX literal prefix$s_name = 8;
D_FLOATING COMPLEX	*off,pos*,0,0 literal prefix$s_name = 16;
G_FLOATING COMPLEX	*off,pos*,0,0 literal prefix$s_name = 16;
H_FLOATING COMPLEX	*off,pos*,0,0 literal prefix$s_name = 32;
DECIMAL PRECISION (p,q)	n/a

VSI SDL Declaration	**VSI BLISS Output**
BITFIELD	*off,pos,size*,0 literal prefix$_name = size;
LENGTH n	*off,pos*,n,0 literal prefix$_name = n;
MASK	literal prefix$m_name = mask-value; where mask-value is the decmal equivalent of the binary mask
SIGNED	*off,pos,size*,1
CHARACTER	*off,pos,size*,0
LENGTH n	literal prefix$_name = n;
LENGTH *	n/a
VARYING	*off,pos,size*,0; where size = (length + 2) * 8. If (length + 2) > 4, then size = 0. literal prefix$s_name = <length + 2>;
ADDRESS	*off,pos*,32,0
POINTER	*off,pos*,32,1
POINTER_LONG	*off,pos*,32,1
POINTER_HW	*off,pos*,0,1 for /ALPHA_AXP *off,pos*,64,1 for /ALPHA_AXP/B64 *off,pos*,32,1 for /VAX
HARDWARE_ADDRESS	*off,pos*,0,1 for /ALPHA_AXP *off,pos*,64,1 for /ALPHA_AXP/B64 *off,pos*,32,1 for /VAX
POINTER_QUAD	*off,pos*,0,1 for /ALPHA_AXP *off,pos*,64,1 for /ALPHA_AXP/B64 *off,pos*,0,1 for /VAX
BOOLEAN	*off,pos*,8,0
user-type-name	*off,pos,size,0;* where *size* is dependent upon the type indicated by user-type-name
Default storage class	n/a
COMMON storage class	EXTERNAL
with /GLOBALDEF	GLOBAL
GLOBAL storage class	EXTERNAL
with /GLOBALDEF	GLOBAL
BASED pointer-name	n/a
TYPEDEF	n/a
DIMENSION [lbound:]hbound	lbound and hbound are only used to determine the total size of the item, which is reflected in: literal prefix$s_name = size;

VSI SDL Declaration	VSI BLISS Output
ORIGIN member-name	n/a

Note

1. The VSI BLISS output routine does not assign data to storage classes.
2. For all literals above, size is given in bytes.

B.4. VSI C/C++ Translation Summary

The C backend differentiates along the qualifiers **/ALPHA_AXP** vs. **/VAX, /[NO]VMS_DEVELOPMENT**, and **/[NO]C_DEVELOPMENT**.

/VMS_DEVELOPMENT and **/C_DEVELOPMENT** can be combined.

When **/C_DEVELOPMENT** is present, every definition is translated twice, once between

```
#ifdef __NEW_STARLET
     /* translation */
#else      /* __OLD_STARLET */
```

and once between

```
#else /* __OLD_STARLET */
    /* possibly different translation */
#endif      /* #ifdef __NEW_STARLET */
```

So there are 12 possibly different translations for a single data type. To make the translation summary more readable, abbreviations are used.

Abbreviation	Qualifier combination
a	/ALPHA_AXP
b	/ALPHA_AXP/VMS_DEVELOPMENT
c	/ALPHA_AXP/VMS_DEVELOPMENT/C_DEVELOPMENT, #ifdef _ _NEW_STARLET
d	/ALPHA_AXP/VMS_DEVELOPMENT/C_DEVELOPMENT, #else
e	/ALPHA_AXP/C_DEVELOPMENT, #ifdef _ _NEW_STARLET
f	/ALPHA_AXP/C_DEVELOPMENT, #else
g	/VAX
h	/VAX/VMS_DEVELOPMENT
i	/VAX/VMS_DEVELOPMENT/C_DEVELOPMENT, #ifdef _ _NEW_STARLET
j	/VAX/VMS_DEVELOPMENT/C_DEVELOPMENT, #else
k	/VAX/C_DEVELOPMENT, #ifdef _ _NEW_STARLET
l	/VAX/C_DEVELOPMENT, #else

The following table shows the VSI SDL to VSI C/C++ language translation summary.

VSI SDL Declaration	**VSI C/C++ Output**
MODULE name IDENT string	/*** MODULE name IDENT string ***/
/* comment	/* comment */
CONSTANT x	
EQUALS n;	#define x n
EQUALS STRING "s";	#define x "s"
ENTRY name	return-type name( )
PARAMETER (type,...)	n/a
ANY	n/a
DESCRIPTOR	n/a
RTL_STR_DESC	n/a
IN	n/a
OUT	n/a
NAMED param-name	n/a
VALUE	n/a
REFERENCE	n/a
DEFAULT n	n/a
LIST	n/a
OPTIONAL	n/a
TYPENAME type-name	n/a
RETURNS return-type	return-type name( )
NAMED param-name	n/a
VARIABLE	n/a
ALIAS internal-name	n/a
LINKAGE	n/a
TYPENAME type-name	n/a
STRUCTURE	struct
UNION	union
name BYTE [SIGNED]	char name
name INTEGER_BYTE [SIGNED]	char name
name WORD [SIGNED]	short int name
name INTEGER_WORD [SIGNED]	short int name
name LONGWORD [SIGNED]	int name
name INTEGER_LONG [SIGNED]	int name

VSI SDL Declaration	VSI C/C++ Output
name INTEGER [SIGNED]	int name
name INTEGER_HW [SIGNED]	_ _int64 name for a-f int name for g-l
name HARDWARE_INTEGER [SIGNED]	_ _int64 name for a-f int name for g-l
name QUADWORD [SIGNED]	int name [2] for a, d, f, g, h, j, l _ _int64 name for b, c, e, i, k
name INTEGER_QUAD [SIGNED]	_ _int64 name for a-f int name[2] for g-l
name OCTAWORD [SIGNED]	int name [4]
name BYTE UNSIGNED	unsigned char
name INTEGER_BYTE UNSIGNED	unsigned char
name WORD UNSIGNED	unsigned short int
name INTEGER_WORD UNSIGNED	unsigned short int
name INTEGER_LONGWORD UNSIGNED	unsigned int
name INTEGER_LONG UNSIGNED	unsigned int
name INTEGER UNSIGNED	unsigned int
name INTEGER_HW UNSIGNED	unsigned _ _int64 name for a-f unsigned int name for g-l
name HARDWARE_INTEGER UNSIGNED	unsigned _ _int64 name for a-f unsigned int name for g-l
name QUADWORD UNSIGNED	unsigned int name [2] for a, d, f, g, h, j, l unsigned _ _int64 name for b, c, e, i, k
name INTEGER_QUAD UNSIGNED	unsigned _ _int64 name for a-f unsigned int name[2] for g-l
name OCTAWORD UNSIGNED	unsigned int name [4]
name F_FLOATING	float name
name D_FLOATING	double float name
name G_FLOATING	double float name
name H_FLOATING	int name [4]
name F_FLOATING COMPLEX	float name [2]
name D_FLOATING COMPLEX	double float name [2]
name G_FLOATING COMPLEX	double float name [2]

VSI SDL Declaration	VSI C/C++ Output
name H_FLOATING COMPLEX	int name [8]
name DECIMAL PRECISION (p,q)	char name [p/2+1]
name BITFIELD	unsigned name:1
LENGTH n	unsigned name:n
MASK	n/a
SIGNED	n/a
name CHARACTER	char name
LENGTH n	char name [n]
LENGTH *	n/a
VARYING	struct {short string_length; char string_text[n];} name; where n is the maximum length of the character string
name ADDRESS (object-type)	object-type *name for a-c, e, g-i, k void * for d, f, j, l
name POINTER (object-type)	object-type *name for a-c, e, g-i, k unsigned int for d, f, j, l
name POINTER_LONG (objecttype)	int for a-l void * for b, c, e, h, i, k
name POINTER_HW (object-type)	object-type *name for a, g-i, k ([1]) for b-c, e unsigned _ _int64 name for d, f unsigned int for j, l
name HARDWARE_ADDRESS (object-type)	object-type *name for a-l
name POINTER_QUAD (objecttype)	object-type *name for a, g ([1]) for b-c, e, h, i, k unsigned _ _int64 name for d, f unsigned int name [2] for j, l
name BOOLEAN	char name
name user-type-name	user-type-name name
Default storage class	struct or union with <member-name> as the tag, and no declared variable names
COMMON storage class	extern attribute
GLOBAL storage class	globalref attribute
with /GLOBALDEF	globaldef attribute
BASED pointer-name	A pointer will be generated for the based item

VSI SDL Declaration	VSI C/C++ Output
TYPEDEF	For an AGGREGATE, a TYPEDEF STRUCT is generated. In this case, a pre-tag is generated as well as the post-tag. The pre-tag consists of the structure named prefixed by an underscore. Any reference to the structure type within the definition (such as for forward and backward linkages) is also output with the underscore. For an ITEM, the TYPEDEF keyword is generated, followed by the data type of the ITEM.
DIMENSION [lbound:]hbound	The declaration specifies an array of the number of elements specified with subscripts ranging from 0 to (hbound - lbound +1)
ORIGIN member-name	n/a

[1]#ifdef __INITIAL_POINTER_SIZE #pragma __required_pointer_size __long object-type *name; #else unsigned __int64 name; #endif

B.5. VSI Datatrieve Translation Summary

The following table shows the VSI SDL to VSI Datatrieve language translation summary.

VSI SDL Declaration	VSI Datatrieve Output
MODULE name IDENT string	! *** MODULE name IDENT string ***
/* comment	! comment
CONSTANT x EQUALS n;	! x=n
ENTRY	! name ENTRY
PARAMETER (type,...)	n/a
ANY	n/a
DESCRIPTOR	n/a
RTL_STR_DESC	n/a
IN	n/a
OUT	n/a
NAMED param-name	n/a
VALUE	n/a
REFERENCE	n/a
DEFAULT n	n/a
LIST	n/a
OPTIONAL	n/a
TYPENAME type-name	n/a
RETURNS return-data-type	n/a

VSI SDL Declaration	VSI Datatrieve Output
NAMED param-name	n/a
VARIABLE	n/a
ALIAS internal-name	n/a
LINKAGE	n/a
TYPENAME type-name	n/a
STRUCTURE	A group field declaration. An AGGREGATE declaration is always assigned a level number of 1; subsequent subaggregates are assigned level numbers 2, 3, and so on.
UNION	A group field declaration, with level numbers assigned as above. All but the first union member have a REDEFINES first-fieldname clause. The redefined fields cannot be larger than the first union member.
BYTE [SIGNED]	USAGE IS BYTE
INTEGER_BYTE [SIGNED]	USAGE IS BYTE
WORD [SIGNED]	USAGE IS WORD
INTEGER_WORD [SIGNED]	USAGE IS WORD
LONGWORD [SIGNED]	USAGE IS LONG
INTEGER_LONG [SIGNED]	USAGE IS LONG
INTEGER [SIGNED]	USAGE IS LONG
INTEGER_HW [SIGNED]	USAGE IS QUAD for /ALPHA_AXP USAGE IS LONG for /VAX
HARDWARE_INTEGER [SIGNED]	USAGE IS QUAD for /ALPHA_AXP USAGE IS LONG for /VAX
QUADWORD [SIGNED]	USAGE IS QUAD
INTEGER_QUAD [SIGNED]	USAGE IS QUAD
OCTAWORD [SIGNED]	USAGE IS QUAD OCCURS 2 TIMES
BYTE UNSIGNED	USAGE IS BYTE
INTEGER_BYTE UNSIGNED	USAGE IS BYTE
WORD UNSIGNED	USAGE IS WORD
INTEGER_WORD UNSIGNED	USAGE IS WORD
LONGWORD UNSIGNED	USAGE IS LONG
INTEGER_LONG UNSIGNED	USAGE IS LONG
INTEGER UNSIGNED	USAGE IS LONG
INTEGER_HW UNSIGNED	USAGE IS QUAD for /ALPHA_AXP USAGE IS LONG for /VAX

VSI SDL Declaration	VSI Datatrieve Output
HARDWARE_INTEGER UNSIGNED	USAGE IS QUAD for /ALPHA_AXP USAGE IS LONG for /VAX
QUADWORD UNSIGNED	USAGE IS QUAD
INTEGER_QUAD UNSIGNED	USAGE IS QUAD
OCTAWORD UNSIGNED	USAGE IS QUAD OCCURS 2 TIMES
F_FLOATING	USAGE IS REAL
D_FLOATING	USAGE IS DOUBLE
G_FLOATING	USAGE IS DOUBLE
H_FLOATING	USAGE IS QUAD OCCURS 2 TIMES
F_FLOATING COMPLEX	USAGE IS REAL OCCURS 2 TIMES
D_FLOATING COMPLEX	USAGE IS DOUBLE OCCURS 2 TIMES
G_FLOATING COMPLEX	USAGE IS DOUBLE OCCURS 2 TIMES
H_FLOATING COMPLEX	USAGE IS QUAD OCCURS 2 TIMES
DECIMAL PRECISION (p,q)	USAGE IS PACKED PIC 9(p-q)V9(q)
BITFIELD	! name BIT position:size See the description following the table
LENGTH n	! name BIT position:n
MASK	! maskname = value
SIGNED	n/a
CHARACTER	PIC X(n)
LENGTH n	X(n)
LENGTH *	n/a
VARYING	A group field of the form: level field-name level+1 STRING_LENGTH USAGE IS WORD level+1 STRING_TEXT PIC X(n)
ADDRESS	USAGE IS LONG
POINTER	USAGE IS LONG
POINTER_LONG	USAGE IS LONG
POINTER_HW	USAGE IS QUAD for /ALPHA_AXP USAGE IS LONG for /VAX
HARDWARE_ADDRESS	USAGE IS QUAD for /ALPHA_AXP USAGE IS LONG for /VAX

VSI SDL Declaration	VSI Datatrieve Output
POINTER_QUAD	USAGE IS QUAD
BOOLEAN	USAGE IS BYTE
Default storage class	n/a
COMMON storage class	n/a
GLOBAL storage class	n/a
with /GLOBALDEF	n/a
BASED pointer-name	n/a
pointer-name	USAGE IS LONG
name DIMENSION [lbound:]hbound	OCCURS hbound-lbound+1 TIMES
ORIGIN member-name	n/a

Note

1. Bitfields must be an integral number of bytes, union members, or be fully contained in a structure composed only of bit members. If they are in a structure, bit names themselves are commented out, but the appropriate amount of storage is allocated at the structure name level. USAGE IS BYTE, WORD, LONG, or QUAD is used when possible to allocate the storage; otherwise, USAGE IS BYTE OCCURS n TIMES is used.

2. All items and aggregates are output as record definitions as follows:

```
DEFINE RECORD aggregate-name_RECORD USING
      1 aggregate-name.
         2 member-name datatype.
              .
              .
              .
           ;
```

   The record name is composed of the top-level aggregate or item name and the string `_ RECORD`.

3. A dollar sign ($) appearing in a name is replaced by a hyphen (-).

B.6. VSI Fortran Translation Summary

The following table shows the VSI SDL to VSI Fortran language translation summary.

VSI SDL Declaration	VSI Fortran Output
MODULE name IDENT string	!*** MODULE name IDENT string ***
/* comment	! comment
CONSTANT x	
EQUALS n;	PARAMETER x = n
EQUALS STRING "s";	CHARACTER*(*) x

VSI SDL Declaration	VSI Fortran Output
	PARAMETER (x = 's')
ENTRY name	EXTERNAL name
PARAMETER (type,...)	n/a
ANY	n/a
DESCRIPTOR	n/a
RTL_STR_DESC	n/a
IN	n/a
OUT	n/a
NAMED param-name	n/a
VALUE	n/a
REFERENCE	n/a
DEFAULT n	n/a
LIST	n/a
OPTIONAL	n/a
TYPENAME type-name	n/a
RETURNS return-data-type	data-type function-name
NAMED param-name	n/a
VARIABLE	n/a
ALIAS internal-name	n/a
LINKAGE	n/a
TYPENAME type-name	n/a
STRUCTURE	If this is a top-level AGGREGATE declaration, a VSI Fortran STRUCTURE declaration is generated. If this is a subaggregate declaration, no structure is generated.
UNION	UNION and associated MAP declarations
name BYTE [SIGNED]	BYTE name
name INTEGER_BYTE [SIGNED]	INTEGER*1 name
name WORD [SIGNED]	INTEGER*2 name
name INTEGER_WORD [SIGNED]	INTEGER*2 name
name LONGWORD [SIGNED]	INTEGER*4 name
name INTEGER_LONG [SIGNED]	INTEGER*4 name
name INTEGER [SIGNED]	INTEGER*4 name
name INTEGER_HW [SIGNED]	INTEGER*8 for /ALPHA_AXP INTEGER*4 name for /VAX

VSI SDL Declaration	VSI Fortran Output
name HARDWARE_INTEGER [SIGNED]	INTEGER*8 for /ALPHA_AXP INTEGER*4 name for /VAX
name QUADWORD [SIGNED]	INTEGER*4 name(2)
name INTEGER_QUAD [SIGNED]	INTEGER*8 for /ALPHA_AXP INTEGER*4 name(2) for /VAX
name OCTAWORD [SIGNED]	INTEGER*4 name(4)
name BYTE UNSIGNED	BYTE name
name INTEGER_BYTE UNSIGNED	BYTE name
name WORD UNSIGNED	INTEGER*2 name
name INTEGER_WORD UNSIGNED	INTEGER*2 name
name LONGWORD UNSIGNED	INTEGER*4 name
name INTEGER_LONG UNSIGNED	INTEGER*4 name
name INTEGER UNSIGNED	INTEGER*4 name
name INTEGER_HW UNSIGNED	INTEGER*8 for /ALPHA_AXP INTEGER*4 name for /VAX
name HARDWARE_INTEGER UNSIGNED	INTEGER*8 for /ALPHA_AXP INTEGER*4 name for /VAX
name QUADWORD UNSIGNED	INTEGER*4 name(2)
name INTEGER_QUAD UNSIGNED	INTEGER*8 for /ALPHA_AXP INTEGER*4 name(2) for /VAX
name OCTAWORD UNSIGNED	INTEGER*4 name(4)
name F_FLOATING	REAL*4 name
name D_FLOATING	REAL*8 name
name G_FLOATING	REAL*8 name
name H_FLOATING	REAL*16 name
name F_FLOATING COMPLEX	COMPLEX name
name D_FLOATING COMPLEX	COMPLEX*16 name
name G_FLOATING COMPLEX	COMPLEX*16 name
name H_FLOATING COMPLEX	BYTE %FILL (32)
DECIMAL PRECISION (p,q)	Undefined data type; error INVOUT
BITFIELD	Offset and size declarations with "V_" and "S_" tags
LENGTH n	n specifies the size for size declaration
MASK	Mask declaration, with "M" tag

VSI SDL Declaration	VSI Fortran Output
SIGNED	n/a
name CHARACTER	CHARACTER*n name
LENGTH n	CHARACTER*n name
LENGTH *	n/a
VARYING	STRUCTURE/name/name INTEGER*2 LEN CHARACTER*length TXT END STRUCTURE
name ADDRESS	INTEGER*4 name
name POINTER	INTEGER*4 name
name POINTER_LONG	INTEGER*4 name
name POINTER_HW	INTEGER*8 name for /ALPHA_AXP INTEGER*4 name for /VAX
name HARDWARE_ADDRESS	INTEGER*8 name for /ALPHA_AXP INTEGER*4 name for /VAX
name POINTER_QUAD	INTEGER*8 name for /ALPHA_AXP INTEGER*4 name(2) for /VAX
name BOOLEAN	BYTE name
name user-type-name	data-type name ! type is "user-type-name"
Default storage class	Local, static
COMMON storage class	RECORD /name/ name COMMON /name/ name
GLOBAL storage class	n/a
with /GLOBALDEF	n/a
BASED pointer-name	n/a
TYPEDEF	For an ITEM, a comment is generated. For an AGGREGATE, the behavior is as though TYPEDEF had not been specified.
DIMENSION [lbound:]hbound	name(hbound - lbound)
ORIGIN member-name	n/a

Note

Because VSI Fortran does not have a comparable data type for BITFIELD, the VSI Fortran back end translates bitfields to PARAMETERS with the same value as that of the associated structure offset. BYTE fillers (using the VSI Fortran %FILL feature) are placed in the structures for alignment.

B.7. VSI MACRO Translation Summary

The following table shows the VSI SDL to VSI MACRO language translation summary.

VSI SDL Declaration	**VSI MACRO Output**
MODULE name IDENT string	.MACRO name ;IDENT string
/* comment	;comment
CONSTANT x	
EQUALS n;	If /NOVMS_DEVELOMENT (default), generates: x'..equ'n If /VMS_DEVELOPMENT is specified, generates: $EQU x n
EQUALS STRING "s";	See Note 2.
ENTRY	If /NOVMS_DEVELOPMENT (default), generates: ;EXTERNAL entry entry-name If /VMS_DEVELOPMENT, generates a special set of macros that facilitate calling of the routine using either the CALLG or CALLS instruction.
PARAMETER (type,...)	If /VMS_DEVELOPMENT, generates the formal argument list for keyword macros associated with the ENTRY declaration
ANY	n/a
DESCRIPTOR	n/a
RTL_STR_DESC	n/a
IN	n/a
OUT	n/a
NAMED param-name	If /VMS_DEVELOPMENT, param-name becomes a formal argument name used in the argument list for keyword macros associated with the ENTRY declaration
VALUE	n/a
REFERENCE	n/a
DEFAULT n	If /VMS_DEVELOPMENT, n becomes a default value in the macro argument list for the parameter being described
LIST	Fills the macro argument list with up to 20 parameters of the type of the parameter being described
OPTIONAL	Generates an "OPTIONAL" macro argument, which either defaults to 0 or is truncated from the actual argument list when the macro is expanded

VSI SDL Declaration	VSI MACRO Output
TYPENAME type-name	n/a
RETURNS return-data-type	n/a
NAMED param-name	n/a
VARIABLE	If /VMS_DEVELOPMENT, causes a special form of macro to be generated that handles a variable number of parameters. See also LIST.
ALIAS internal-name	Macro name (default is entry name)
LINKAGE	Call instruction (default is CALLS)
TYPENAME type-name	n/a
STRUCTURE	See Note 1.
name BYTE [SIGNED]	.
name INTEGER_BYTE [SIGNED]	.
name WORD [SIGNED]	.
name INTEGER_WORD [SIGNED]	.
name LONGWORD [SIGNED]	.
name INTEGER_LONG [SIGNED]	.
name INTEGER [SIGNED]	.
name INTEGER_HW [SIGNED]	.
name HARDWARE_INTEGER [SIGNED]	.
name QUADWORD [SIGNED]	.
name INTEGER_QUADWORD [SIGNED]	.
name OCTAWORD [SIGNED]	.
name BYTE UNSIGNED	.
name INTEGER_BYTE UNSIGNED	.
name WORD UNSIGNED	.
name INTEGER_WORD UNSIGNED	.
name LONGWORD UNSIGNED	.
name INTEGER_LONG UNSIGNED	.
name INTEGER UNSIGNED	.
name INTEGER_HW UNSIGNED	.

VSI SDL Declaration	VSI MACRO Output
name HARDWARE_INTEGER UNSIGNED	.
name QUADWORD UNSIGNED	.
name INTEGER_QUADWORD UNSIGNED	.
name OCTAWORD UNSIGNED	.
name F_FLOATING	.
name D_FLOATING	.
name G_FLOATING	.
name H_FLOATING	.
name F_FLOATING COMPLEX	.
name D_FLOATING COMPLEX	.
name G_FLOATING COMPLEX	.
name H_FLOATING COMPLEX	.
name DECIMAL PRECISION (p,q)	.
name ADDRESS	.
name POINTER	.
name POINTER_LONG	.
name POINTER_HW	.
name HARDWARE_ADDRESS	.
name POINTER_QUAD	.
name BOOLEAN	.
name user-type-name	.
name CHARACTER	.
LENGTH n	
LENGTH *	
VARYING	
name BITFIELD LENGTH n	Bitfield identifiers are equal to the bit offset of the item, and a prefix$_name identifier is equal to the size of the bitfield in bits; see Note 1.
MASK	Mask declarations are constants with the "m_" tag
SIGNED	n/a; see Note 1.
Default storage class	The aggregate is placed in the absolute Psect ABS, and the value of the current location counter is set to the origin at the beginning of the aggregate. Element names can then be used as displacements off a register that contains the address of the actual aggregate.

VSI SDL Declaration	VSI MACRO Output
COMMON storage class	An aggregate or item is placed in a Psect that has the same name as the top-level aggregate or item, and the attributes SHR, GBL, and OVR. Constant offsets are produced for all aggregate members.
GLOBAL storage class	Generates .EXTERNAL declaration for the top-level name, and produces offset constants for any aggregate members.
with /GLOBALDEF	Generates .GLOBAL declaration and BLKB length for top-level name, and produces offset constants for any aggregate members.
BASED pointer-name	n/a
TYPEDEF	n/a
DIMENSION [lbound:]hbound	Offsets are appropriately adjusted to allow for the size of the array.
ORIGIN member-name	Aggregate members preceding member-name may be referenced using negative offsets with member-name as the base.

The VSI MACRO output routine produces declarations that can generate either local symbol or global symbol definitions. The following is an example of VSI SDL source code:

```
MODULE simple;
   CONSTANT bits EQUALS 4;
   ITEM field BYTE PREFIX tst$ COMMON;
END_MODULE;
```

The following is the resulting VSI MACRO output:

```
  .MACRO simple,..EQU=<=>,..COL=<:>
bits'..equ'4
  .SAVE
  .PSECT tst$b_field PIC,OVR,REL,GBL,-
                    SHR,NOEXE,RD,WRT,LONG
tst$b_field'..col' <P>lkb 1
  .RESTORE
; tst$b_field'..equ'0
  .ENDM
```

When the macro is invoked without arguments, the resulting local definitions are as follows:

```
bits = 4
tst$b_field1:
  blkb 1
```

To generate these names as global symbols, invoke the macro with the arguments <= => and

<::>, as follows:

```
simple ..EQU=<==> ..COL=<::>
```

This invocation results in the following definitions:

```
bits == 4
tst$b_field1::
  blkb 1
```

The VSI MACRO output routine always generates a size variable for bitfields, aggregates, arrays, and character strings, using the tag S_ preceding the output identifier.

The **/VMS_DEVELOPMENT** qualifier on the SDL command produces special forms of macros for entry point declarations.

Note

1. Each identifier produces a constant assignment equal to the byte offset of the item, as follows:

   ```
   name = offset-value
   ```

   Offset-value is the byte offset relative to the origin of the level-1 aggregate. A constant assignment of the following form gives the size in bytes for aggregates, arrays, and character strings:

   ```
   prefix$_name = byte-size
   ```

2. 2. If **/VMS_DEVELOPMENT** was specified, the CONSTANT x EQUALS STRING "s" declaration translates to the following:

   ```
   .SAVE
   .PSECT module_name_STRCONST PIC,CON,REL,NOEXE,GBL,SHR,RD,NOWRT,LONG
   $EQU S_x size
   $DEF x .ASCII /s/
   .RESTORE
   ```

   If **/NOVMS_DEVELOPMENT** was specified (default), this declaration translates to the following:

   ```
   .SAVE
   .PSECT module_name STRCONST PIC,CON,REL,NOEXE,GBL,SHR,RD,NOWRT,LONG
   S_x ..equ'size
   x'..col' .ASCII /s/
   .RESTORE
   ```

   where *size* represents the number of bytes in the string.

Note

- If the string contains the / character, another delimiter character (pulled from a priority list) is used.
- If the length of the PSECT name exceeds 31 characters after appending _STRCONST, the module_name is truncated appropriately before appending _STRCONST.

B.8. VSI Pascal Translation Summary

The following table shows the VSI SDL to VSI Pascal language translation summary.

VSI SDL Declaration	VSI Pascal Output
MODULE name IDENT string	(*** MODULE name IDENT string ***) Note that when /MODULE is used, a VSI Pascal MODULE statement of the form MODULE name; is generated.
/* comment	(* comment *)

VSI SDL Declaration	VSI Pascal Output
CONSTANT x	
EQUALS n;	CONST x = n;
EQUALS STRING "s";	CONST x = 's';
ENTRY	[ASYNCHRONOUS] FUNCTION [ASYNCHRONOUS] PROCEDURE
PARAMETER (type,...)	(formal param_list)
ANY	%REF param-name: [UNSAFE] ARRAY [$l1..$u1:INTEGER] OF $UBYTE
DESCRIPTOR	[CLASS_S] for scalars; [CLASS_S] for CHARACTER LENGTH 1; [CLASS_A] for nonscalars; %DESCR for fixed-length VARYING;
RTL_STR_DESC	PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR;
IN	Default semantics (unless overridden by OUT)
OUT (or IN OUT)	VAR, except for CHARACTER LENGTH * and ANY (which generate %REF)
NAMED param-name	Parameter name. If none is given, names will be generated of the form $P1,...$Pn.
VALUE	%IMMED
REFERENCE	%REF for CHARACTER LENGTH * and ANY; VAR for all others if mode OUT or IN OUT; otherwise (IN only) default mechanism
DEFAULT n	:= %IMMED value
LIST	If OPTIONAL is not specified, generates one required parameter followed by a parameter with the [LIST] attribute; if OPTIONAL is specified, only the parameter with [LIST] is generated.
OPTIONAL	:=%IMMED 0
TYPENAME type-name	n/a
RETURNS return-data-type	FUNCTION name : data-type;
NAMED param-name	n/a
VARIABLE	n/a
ALIAS internal-name	Results in [EXTERNAL (entry-name)] and internal-name being used as entry-name
LINKAGE	n/a
TYPENAME type-name	n/a

VSI SDL Declaration	VSI Pascal Output
STRUCTURE	RECORD
UNION	RECORD CASE INTEGER OF 0: ... . . n: ...
BYTE [SIGNED]	$BYTE—[BYTE] -128..127
INTEGER_BYTE [SIGNED]	$BYTE—[BYTE] -128..127
WORD [SIGNED]	$WORD—[WORD] -32768..32767
INTEGER_WORD [SIGNED]	$WORD—[WORD] -32768..32767
LONGWORD [SIGNED]	INTEGER
INTEGER_LONG [SIGNED]	INTEGER
INTEGER [SIGNED]	INTEGER
INTEGER_HW [SIGNED]	$QUAD—[QUAD,UNSAFE]RECORD L0: UNSIGNED; L1: INTEGER; END for /ALPHA_AXP INTEGER for /VAX
HARDWARE_INTEGER [SIGNED]	$QUAD for /ALPHA_AXP $QUAD for /ALPHA_AXP, INTEGER for /VAX
QUADWORD [SIGNED]	$QUAD
INTEGER_QUADWORD [SIGNED]	$QUAD
OCTAWORD [SIGNED]	$OCTA—[OCTA,UNSAFE]RECORD L0,L1,L2: UNSIGNED; L3: INTEGER; END
BYTE UNSIGNED	$UBYTE—[BYTE] 0..255
INTEGER_BYTE UNSIGNED	$UBYTE—[BYTE] 0..255
WORD UNSIGNED	$UWORD—[WORD] 0..65535
INTEGER_WORD UNSIGNED	$UWORD—[WORD] 0..65535
LONGWORD UNSIGNED	UNSIGNED
INTEGER_LONG UNSIGNED	UNSIGNED
INTEGER UNSIGNED	UNSIGNED
INTEGER_HW UNSIGNED	$UQUAD—[QUAD,UNSAFE]RECORD L0,L1: UNSIGNED; END for /ALPHA_AXP INTEGER for /VAX
HARDWARE_INTEGER UNSIGNED	$UQUAD for /ALPHA_AXP

VSI SDL Declaration	VSI Pascal Output
	INTEGER for /VAX
QUADWORD UNSIGNED	$UQUAD
INTEGER_QUADWORD UNSIGNED	$UQUAD
OCTAWORD UNSIGNED	$UOCTA—[OCTA,UNSAFE]RECORD L0,L1,L2,L3: UNSIGNED; END
F_FLOATING	SINGLE
D_FLOATING	DOUBLE (D_FLOAT$$TYPE if the logical name SDLPASCAL$FLAG is defined)
G_FLOATING	DOUBLE (G_FLOAT$$TYPE if the logical name SDLPASCAL$FLAG is defined)
H_FLOATING	QUADRUPLE
F_FLOATING COMPLEX	$UQAD
D_FLOATING COMPLEX	$UOCTA
G_FLOATING COMPLEX	$UOCTA
H_FLOATING COMPLEX	$UOCTAQUAD –[OCTA(2),UNSAFE]RECORD L0,L1,L3,L4,L5,L6,L7:UNSIGNED END;
DECIMAL PRECISION (p,q)	PACKED ARRAY [1..p+2-mod(p,2)] OF $PACKED_DEC $PACKED_DEC—[BIT(4),UNSAFE] 0..15
BITFIELD LENGTH *n*	$BIT1—[BIT(1), UNSAFE] BOOLEAN $BIT*n*—[BIT(*n*),UNSAFE] 0..2**n-1
MASK	CONST prefixM_name = mask-value;
SIGNED	n/a
CHARACTER	CHAR
LENGTH n	PACKED ARRAY [1..n] OF CHAR
LENGTH *	PACKED ARRAY [$l..$u] OF CHAR or VARYING [$m] OF CHAR (if VARYING is also specified)
VARYING	VARYING [n] OF CHAR or VARYING [$m] OF CHAR (if LENGTH * is also specified)
ADDRESS (object-type)	^object-type If object type is not supplied, generates $DEFPTR, which is defined as ^$DEFTYP; $DEFTYP is defined as [UNSAFE] INTEGER
POINTER (object-type)	^object-type

VSI SDL Declaration	VSI Pascal Output
	If object type is not supplied, generates UNSIGNED
POINTER_LONG (object-type)	^object-type If object type is not supplied, generates UNSIGNED
POINTER_HW (object-type)	$QUAD for /ALPHA_AXP INTEGER for /VAX
HARDWARE_ADDRESS (objecttype)	$QUAD for /ALPHA_AXP INTEGER for /VAX
POINTER_QUAD (object-type)	$QUAD
BOOLEAN	BOOLEAN
user-type-name	user-type-name
Default storage class	TYPE
COMMON storage class	[COMMON]
GLOBAL storage class	[EXTERNAL]
with /GLOBALDEF	[GLOBAL]
BASED pointer-name	TYPE pointer-name = structure-name
TYPEDEF	n/a
name DIMENSION [lbound:]hbound	ARRAY [lbound..hbound] OF data type; if lbound is not supplied it defaults to 1
ORIGIN member-name	n/a

Note

1. When the **/MODULE** (default) qualifier is used, VSI SDL generates the VSI Pascal MODULE statement followed by a block of data type definitions. Both of these items are omitted when **/NOMODULE** is specified. This behavior facilitates combining multiple VSI SDL-generated VSI Pascal output files into a single module.

2. Where type names are required (for example, in parameter lists, function return types, and pointer data types), they will be generated in a TYPE block at the beginning of the module. Names have the form module-name$$TYPn, where module-name is truncated to 20 characters if necessary, and n is an integer beginning at 1 and incremented by 1 for each type generated in a module.

3. Bitfields cannot be more than 32 bits in length, so type names giving the appropriate subranges will be generated for each possible bitfield size. Bitfields of length 1 are a special case.

4. The mask-value generated by VSI SDL for the MASK option of the BITFIELD data type is an integer.

5. VSI SDL generates only single-level records for VSI Pascal. Thus, you can avoid writing many long intermediate field names in the VSI Pascal source program. If the outer level is a structure, a PACKED RECORD is typically declared. If the outer level is a union, a PACKED RECORD CASE INTEGER is declared.

In the case of multiple levels of structures or unions in VSI SDL, the entire aggregate is transformed into a PACKED RECORD CASE INTEGER. The intermediate field-names are themselves declared as BYTE_DATA fields. This translation scheme may result in several fields in the record having the same name.

B.9. PL/I Translation Summary

The following table shows the VSI SDL to PL/I language translation summary.

VSI SDL Declaration	**PL/I Output**
MODULE name IDENT string	/*** MODULE name IDENT string ***/
/* comment	/*comment*/
CONSTANT x	
EQUALS n;	%REPLACE x BY n;
EQUALS STRING "s";	%REPLACE x BY 's';
ENTRY	ENTRY
PARAMETER (type,...)	(parameter-descriptor,...)
ANY	ANY
DESCRIPTOR	DESCRIPTOR or CHARACTER(*) for CHARACTER data type
RTL_STR_DESC	ANY CHARACTER(*)
IN	n/a
OUT	n/a
NAMED param-name	n/a
VALUE	VALUE
REFERENCE	n/a
DEFAULT n	n/a
LIST	LIST
OPTIONAL	OPTIONAL or OPTIONAL TRUNCATE
TYPENAME type-name	Special VMS TYPENAME values are recognized.
RETURNS return-data-type	RETURNS (returns-descriptor)
NAMED param-name	n/a
VARIABLE	OPTIONS(VARIABLE)
ALIAS internal-name	n/a
LINKAGE	n/a
TYPENAME type-name	Special VMS TYPENAME values are recognized.

VSI SDL Declaration	PL/I Output
STRUCTURE	Structure declaration. An AGGREGATE declaration is always assigned a level number of 1; subsequent subaggregates are assigned level numbers 2, 3, and so on.
UNION	UNION
BYTE [SIGNED]	FIXED BINARY(7)
INTEGER_BYTE [SIGNED]	FIXED BINARY(7)
WORD [SIGNED]	FIXED BINARY(15)
INTEGER_WORD [SIGNED]	FIXED BINARY(15)
LONGWORD [SIGNED]	FIXED BINARY(31)
INTEGER_LONG [SIGNED]	FIXED BINARY(31)
INTEGER [SIGNED]	FIXED BINARY(31)
INTEGER_HW [SIGNED]	(2) FIXED BIN(31) for /ALPHA_AXP FIXED BIN(31) for /VAX
HARDWARE_INTEGER [SIGNED]	(2) FIXED BIN(31) for /ALPHA_AXP FIXED BIN(31) for /VAX
QUADWORD [SIGNED]	BIT(64) ALIGNED
INTEGER_QUAD [SIGNED]	BIT(64) ALIGNED
OCTAWORD [SIGNED]	BIT(128) ALIGNED
BYTE UNSIGNED	BIT(8) ALIGNED FIXED BINARY(7) for /PLI_DEVELOPMENT
INTEGER_BYTE UNSIGNED	BIT(8) ALIGNED FIXED BINARY(7) for /PLI_DEVELOPMENT
WORD UNSIGNED	BIT(16) ALIGNED FIXED BINARY(15) for /PLI_DEVELOPMENT
INTEGER_WORD UNSIGNED	BIT(16) ALIGNED FIXED BINARY(15) for /PLI_DEVELOPMENT
LONGWORD UNSIGNED	BIT(32) ALIGNED FIXED BINARY(31) for /PLI_DEVELOPMENT
INTEGER_LONG UNSIGNED	BIT(32) ALIGNED FIXED BINARY(31) for /PLI_DEVELOPMENT
INTEGER UNSIGNED	BIT(32) ALIGNED FIXED BINARY(31) for /PLI_DEVELOPMENT
INTEGER_HW UNSIGNED	BIT(64) ALIGNED for /ALPHA_AXP

VSI SDL Declaration	PL/I Output
	(2) FIXED BIN(31)/ALPHA_AXP/PLI_DEVELOPMENT BIT(32) ALIGNED for /VAX FIXED BIN(31)/VAX/PLI_DEVELOPMENT
HARDWARE_INTEGER UNSIGNED	BIT(64) ALIGNED for /ALPHA_AXP (2) FIXED BIN(31)/ALPHA_AXP/PLI_DEVELOPMENT BIT(32) ALIGNED for /VAX FIXED BIN(31)/VAX/PLI_DEVELOPMENT
QUADWORD UNSIGNED	BIT(64) ALIGNED
INTEGER_QUAD UNSIGNED	BIT(64) ALIGNED
OCTAWORD UNSIGNED	BIT(128) ALIGNED
F_FLOATING	FLOAT BINARY (24)
D_FLOATING	FLOAT BINARY(53)
G_FLOATING	FLOAT BINARY(53)
H_FLOATING	FLOAT BINARY(113)
F_FLOATING COMPLEX	ANY except as a
D_FLOATING COMPLEX	- function return type or union
G_FLOATING COMPLEX	- type of a structure member
H_FLOATING COMPLEX	- array type In these cases, respective translations are: BIT(64) BIT(128) BIT(128) BIT(256)
DECIMAL PRECISION (p,q)	DECIMAL (p,q)
BITFIELD LENGTH n	BIT(n)
MASK	%REPLACE prefixM_name BY mask-value;
SIGNED	n/a
CHARACTER	CHARACTER(n)
LENGTH n	CHARACTER(n)
LENGTH *	CHARACTER(*)
VARYING	VARYING
ADDRESS	POINTER

VSI SDL Declaration	PL/I Output
POINTER	POINTER
POINTER_LONG	POINTER
POINTER_HW	(2) POINTER for /ALPHA_AXP POINTER for /VAX
HARDWARE_ADDRESS	(2) POINTER for /ALPHA_AXP POINTER for /VAX
POINTER_QUAD	(2) POINTER
BOOLEAN	BIT(1) ALIGNED
user-type-name	data-type /* user-type-name */
Default storage class	BASED attribute
COMMON storage class	STATIC EXTERNAL
GLOBAL storage class	GLOBALREF
with /GLOBALDEF	GLOBALDEF
BASED pointer-name	aggregate BASED (pointer-name)
TYPEDEF	n/a
name DIMENSION [lbound:]hbound	name ([lbound]:hbound)
ORIGIN member-name	n/a

Note

1. In PL/I, you can access values declared using the VSI SDL UNSIGNED keyword by specifying the name in a POSINT built-in function. The integer value of the name will be returned; however, this will work only if the sign bit is 0.

2. The mask value generated by VSI SDL for the MASK option of the BITFIELD data type is a bit-string constant.

3. The VSI SDL declarations BYTE UNSIGNED, WORD UNSIGNED, and LONGWORD UNSIGNED produce the same PL/I output as the VSI SDL declarations BYTE, WORD, and LONGWORD, when the /VMS_DEVELOPMENT qualifier is specified.

B.10. VSI DCL Translation Summary

The following table shows the VSI SDL to VSI DCL language translation summary. Since VSI DCL does not understand procedures or aggregates, only constants are translated.

VSI SDL Declaration	VSI DCL Output
MODULE name IDENT string	$! MODULE name
/* comment	$! comment

VSI SDL Declaration	VSI DCL Output
CONSTANT x	
EQUALS n;	x = = n
EQUALS STRING "s";	x = = "s"
ENDMODULE name	$! ENDMODULE