Understand Your Software...

Perl and C Application Program Interface for Understand

User Guide and Reference Manual

Version 1.4
May 24, 2004

Scientific Toolworks, Inc.

Scientific Toolworks, Inc.
321 N. Mall Drive Suite I-201
St. George, UT 84790

Part Number: USTAND-API-UG-257 (5/04)

Contents

Chapter 1 APIs to Understand Databases

What Can the Understand APIs Do? 1–2
Which API Should I Use? 1–2

Chapter 2 Using the Perl API

Running Perl Scripts 2–2
- Understand License Requirements 2–2
- Perl Version 2–2
- Running Scripts with uperl 2–3
- Running Scripts with Other Perl Installations 2–3

Understand Package Classes 2–4
Scripting with the Perl API 2–5
- Traversing an Understand Database 2–5
- Starting a Script 2–5
- Opening a Database 2–5
- Closing a Database 2–6
- Getting a List of Entities 2–6
- Getting Entity Attributes 2–7
- Getting a List of References 2–7
- Getting Associated Comments 2–8
- Getting Project Metrics 2–9
- Getting Entity Metrics 2–9
- Getting Info Browser Text 2–10
- Saving Graphics Files 2–10
- Lexer 2–10
- Using the GUI Class 2–11

Chapter 3 Perl API Class and Method Reference

Understand Package 3–2
- Understand::license() 3–2
- Understand::open() 3–2
- Understand::version() 3–3

Understand::Db class 3–4
- $db->check_comment_association() 3–4
- $db->close() 3–4
- $db->ents() 3–4
- $db->language() 3–4
- $db->last_id() 3–4
- $db->lookup() 3–5
- $db->lookup_uniquename() 3–5

$db->metric() 3–5
$db->metrics() 3–5
$db->name() 3–5
Understand::Ent class 3–6
$ent->comments() 3–6
$ent->draw() 3–7
$ent->ents() 3–9
$ent->filerefs() 3–9
$ent->ib() 3–10
$ent->id() 3–11
$ent->kind() 3–11
$ent->kindname() 3–11
$ent->language() 3–11
$ent->lexer() 3–12
$ent->library() 3–12
$ent->longname() 3–12
$ent->metric() 3–12
$ent->metrics() 3–13
$ent->name() 3–13
$ent->parameters() 3–13
$ent->refs() 3–13
$ent->ref() 3–14
$ent->relname() 3–14
$ent->type() 3–14
$ent->uniquename() 3–14
Understand::Gui class 3–15
Gui::active() 3–15
Gui::column() 3–15
Gui::db() 3–15
Gui::disable_cancel() 3–15
Gui::entity() 3–16
Gui::filename() 3–16
Gui::line() 3–16
Gui::progress_bar() 3–16
Gui::selection() 3–17
Gui::word() 3–17
Gui::yield() 3–17
Understand::Kind class 3–18
$kind->check() 3–18
$kind->inv() 3–18
Kind::list_entity() 3–18
Kind::list_reference() 3–18
$kind->longname() 3–19
$kind->name() 3–19
Understand::Lexeme class 3–20

$lexeme->column_begin() 3–20
$lexeme->column_end() 3–20
$lexeme->ent() 3–20
$lexeme->inactive() 3–20
$lexeme->line_begin() 3–20
$lexeme->line_end() 3–20
$lexeme->next() 3–21
$lexeme->previous() 3–21
$lexeme->ref() 3–21
$lexeme->text() 3–21
$lexeme->token() 3–21
Understand::Lexer class 3–22
$lexer->first() 3–22
$lexer->lexeme() 3–22
$lexer->lexemes() 3–22
$lexer->lines() 3–22
Understand::Metric class 3–23
Metric::description() 3–23
Metric::list() 3–23
Understand::Ref class 3–23
$ref->column() 3–23
$ref->ent() 3–23
$ref->file() 3–24
$ref->kind() 3–24
$ref->kindname() 3–24
$ref->line() 3–24
$ref->scope() 3–24
Understand::Util class 3–25
Util::checksum() 3–25
Understand::Visio class 3–25
Visio::draw() 3–25
Visio::quit() 3–25

Chapter 4 Using the C API

Using the *Understand* C API 4–2
Downloading 4–2
Licensing 4–2
API Memory Allocation 4–2
Getting Started with the C API 4–3
API Include file 4–3
Understand Licensing 4–3
Opening and Closing the Database 4–4
Get a List of Entities 4–4
Get Information About an Entity 4–4

Get Reference Information for an Entity 4–5
About Library Support 4–5
Compiling & Linking with the API library 4–6
Code Sample Using the C API 4–7

Chapter 5 C API Functions

API Function Reference 5–4
udbComment 5–5
udbCommentRaw 5–7
udbDbClose 5–8
udbDbLanguage 5–9
udbDbName 5–10
udbDbOpen 5–11
udbEntityDraw 5–13
udbEntityId 5–16
udbEntityKind 5–17
udbEntityLanguage 5–18
udbEntityLibrary 5–19
udbEntityNameLong 5–20
udbEntityNameShort 5–21
udbEntityNameSimple 5–22
udbEntityNameUnique 5–23
udbEntityRefs 5–24
udbEntityTypetext 5–25
udbInfoBuild 5–26
udbIsKind 5–27
udbIsKindFile 5–28
udbKindInverse 5–29
udbKindLanguage 5–30
udbKindList 5–31
udbKindListCopy 5–32
udbKindListFree 5–33
udbKindLocate 5–34
udbKindLongname 5–35
udbKindParse 5–36
udbKindShortname 5–37
udbLexemeColumnBegin 5–38
udbLexemeColumnEnd 5–39
udbLexemeEntity 5–40

udbLexemeInactive 5–41
udbLexemeLineBegin 5–42
udbLexemeLineEnd 5–43
udbLexemeNext 5–44
udbLexemePrevious 5–45
udbLexemeReference 5–46
udbLexemeText 5–47
udbLexemeToken 5–48
udbLexerDelete 5–49
udbLexerFirst 5–50
udbLexerLexeme 5–51
udbLexerLexemes 5–52
udbLexerLines 5–53
udbLexerNew 5–54
udbLibraryCheckEntity 5–55
udbLibraryCompare 5–56
udbLibraryFilterEntity 5–57
udbLibraryList 5–58
udbLibraryListFree 5–59
udbLibraryName 5–60
udbLicenseInfo 5–61
udbListEntity 5–62
udbListEntityFilter 5–63
udbListEntityFree 5–64
udbListFile 5–65
udbListKindEntity 5–66
udbListKindFree 5–67
udbListKindReference 5–68
udbListReference 5–69
udbListReferenceFile 5–70
udbListReferenceFilter 5–71
udbListReferenceFree 5–72
udbLookupEntity 5–73
udbLookupEntityByReference 5–74
udbLookupEntityByUniquename 5–75
udbLookupFile 5–76
udbLookupReferenceExists 5–77
udbMetricDescription 5–78

udbMetricIsDefinedEntity 5–79
udbMetricIsDefinedProject 5–80
udbMetricKind 5–81
udbMetricListEntity 5–82
udbMetricListKind 5–83
udbMetricListLanguage 5–84
udbMetricListProject 5–85
udbMetricLookup 5–86
udbMetricName 5–87
udbMetricValue 5–88
udbMetricValueProject 5–89
udbReferenceColumn 5–90
udbReferenceCopy 5–91
udbReferenceCopyFree 5–92
udbReferenceEntity 5–93
udbReferenceFile 5–94
udbReferenceKind 5–95
udbReferenceLine 5–96
udbReferenceScope 5–97

Chapter 6 C API Code Samples

Open Database and Get All Entities 6–2
Report All Entities 6–4
Report All Files 6–6
Report Functions with their Parameters and Types 6–7
Report Global Objects 6–9
Report All Structs and their Member Types 6–11
Find All References To and From an Entity 6–13

Chapter 7 Entity and Reference Kinds

Kind Name Usage 7–2
- Using Kinds in the Perl API 7–2
- Using Kinds in the C API 7–3
- Kind Name Filters 7–4
- Examples 7–5

Ada Entity Kinds 7–6
- About “Local” Kinds 7–6
- About Derived Types and SubTypes 7–6
- Ada Component Kinds 7–6
- Ada Constant Kinds 7–7

Ada Entry Kinds 7–8
Ada Enumeration Literal Kinds 7–8
Ada Exception Kinds 7–8
Ada File Kinds 7–9
Ada Function Kinds 7–9
Ada Implicit Kinds 7–10
Ada Object Kinds 7–10
Ada Package Kinds 7–11
Ada Parameter Kinds 7–12
Ada Procedure Kinds 7–12
Ada Protected Kinds 7–13
Ada Task Kinds 7–14
Ada Type Kinds 7–15
Ada Unknown Kinds 7–18
Ada Unresolved Kinds 7–18
Ada Reference Kinds 7–19
Ada Abort and Abortby Kinds 7–19
Ada AccessAttrTyped and AccessAttrTypedby Kinds . . 7–19
Ada Association and Associationby Kinds 7–20
Ada Call and Callby Kinds 7–20
Ada CallParamFormal and CallParamFormalfor Kinds 7–22
Ada Child and Parent Kinds 7–22
Ada Declare and Declarein Kinds 7–23
Ada Derive and Derivefrom Kinds 7–25
Ada Dot and Dotby Kinds 7–26
Ada End and Endby Kinds 7–26
Ada Elaborate Body and Elaborate Bodyby Kinds 7–27
Ada Handle and Handleby Kinds 7–27
Ada Instance and Instanceof Kinds 7–28
Ada Operation and Operationfor Kinds 7–29
Ada Override and Overrideby Kinds 7–29
Ada Raise and Raiseby Kinds 7–30
Ada Ref and Refby Kinds 7–30
Ada Rename and Renameby Kinds 7–31
Ada Root and Rootin Kinds 7–32
Ada Separatefrom and Separate Kinds 7–32
Ada Set and Setby Kinds 7–33
Ada Subtype and Ada Subtypefrom Kinds 7–33
Ada Typed and Typedby Kinds 7–34
Ada Use and Useby Kinds 7–35
Ada Usepackage and Usepackageby Kinds 7–36
Ada UseType and UseTypeby Kinds 7–36
Ada With and Withby Kinds 7–37
C/C++ Entity Kinds 7–39
C Class Kinds 7–39

C Enum Kinds 7–40
C Enumerator Kinds 7–41
C File Kinds 7–42
C Function Kinds 7–42
C Macro Kinds 7–45
C Namespace Kinds 7–46
C Object Kinds 7–46
C Parameter Kinds 7–47
C Struct Kinds 7–48
C Typedef Kinds 7–49
C Union Kinds 7–50
C/C++ Reference Kinds 7–52
C Base and Derive Kinds 7–52
C Call and Callby Kinds 7–53
C Declare and Declarein Kinds 7–54
C Define and Definein Kinds 7–55
C End and Endby Kinds 7–55
C Exception Kinds 7–56
C Friend and Friendby Kinds 7–56
C Include and Includeby Kinds 7–57
C Modify and Modifyby Kinds 7–57
C Overrides and Overriddenby Kinds 7–58
C Set and Setby Kinds 7–58
C Typed and Typedby Kinds 7–59
C Use and Useby Kinds 7–59
FORTRAN Entity Kinds 7–62
Fortran Block Data 7–63
Fortran Block Variable 7–63
Fortran Common Block 7–63
Fortran Datapool 7–63
Fortran Derived Type 7–63
Fortran Dummy Argument 7–64
Fortran Entry 7–64
Fortran File 7–64
Fortran Function 7–64
Fortran Include File 7–64
Fortran Interface 7–64
Fortran Intrinsic Function or Subroutine 7–65
Fortran Main Program 7–65
Fortran Module 7–65
Fortran NameList 7–65
Fortran Pointer 7–65
Fortran Subroutine 7–66
Fortran Unknown Include File 7–66
Fortran Unknown Module 7–66

Fortran Unresolved Function 7–66
Fortran Unresolved Include File 7–66
Fortran Unresolved Subroutine 7–66
Fortran Variable 7–67
FORTRAN Reference Kinds 7–68
Fortran Call 7–69
Fortran Call Ptr 7–70
Fortran Contain 7–70
Fortran Declare and Define 7–70
Fortran End 7–73
Fortran Equivalence 7–73
Fortran Include 7–74
Fortran ModuleUse 7–74
Fortran Ref 7–74
Fortran Set 7–75
Fortran Typed 7–75
Fortran Use 7–75
Fortran UseModuleEntity and ModuleUse Only 7–76
Fortran UseRenameEntity 7–76
Java Entity Kinds 7–77
General Notes 7–79
Java Abstract Class Type Member 7–80
Java Abstract Method Member 7–80
Java Catch Parameter 7–80
Java Class Type Anonymous Member 7–80
Java Class Type Member 7–81
Java File 7–81
Java File Jar 7–81
Java Final Class Type Member 7–81
Java Final Method Member 7–81
Java Final Variable Member 7–82
Java Final Variable Local 7–82
Java Interface Type 7–82
Java Method Constructor Member 7–82
Java Method Member 7–82
Java Package 7–83
Java Parameter 7–83
Java Static Abstract Class Type Member 7–83
Java Static Class Type Member 7–83
Java Static Final Class Type Member 7–84
Java Static Final Method Member 7–84
Java Static Final Variable Member 7–84
Java Static Method Member 7–84
Java Static Method Public Main Member 7–85
Java Static Variable Member 7–85

Java Unknown Class Type Member 7–85
Java Unknown Method Member 7–85
Java Unknown Package 7–85
Java Unknown Variable Member 7–85
Java Unresolved Method 7–85
Java Unresolved Package 7–86
Java Unresolved Type 7–86
Java Unresolved Variable 7–86
Java Unused 7–86
Java Variable Member 7–86
Java Variable Local 7–86
Java Reference Kinds 7–87
Java Call 7–88
Java Call Nondynamic 7–89
Java Cast 7–89
Java Contain 7–90
Java Couple 7–90
Java Create 7–90
Java Define 7–91
Java DotRef 7–91
Java End 7–91
Java Extend Couple 7–92
Java Implement Couple 7–93
Java Import 7–93
Java Modify 7–93
Java Override 7–94
Java Set 7–94
Java Throw 7–95
Java Typed 7–95
Java Use 7–95
JOVIAL Entity Kinds 7–96
Jovial Copy File 7–99
Jovial CompoolFile 7–99
Jovial Constant Variable Item 7–99
Jovial Constant Variable Table 7–99
Jovial Constant Component Variable Item 7–99
Jovial Constant Component Variable Table 7–100
Jovial Variable Block 7–100
Jovial Variable Item 7–100
Jovial Variable Table 7–100
Jovial Component Variable Block 7–101
Jovial Component Variable Item 7–101
Jovial Component Variable Table 7–101
Jovial File 7–101
Jovial Macro 7–102

Jovial Compool Module 7–102
Jovial Input Parameter / Jovial Output Parameter . . . 7–102
Jovial Program Procedure Subroutine 7–102
Jovial Statusname 7–102
Jovial Function Subroutine 7–102
Jovial Procedure Subroutine 7–102
Jovial Type Block 7–103
Jovial Type Item 7–103
Jovial Type Table 7–103
Jovial Component Type Block 7–103
Jovial Component Type Item 7–103
Jovial Component Type Table 7–104
Jovial Unknown 7–104
Jovial Unknown Copy File 7–104
Jovial Unresolved Compool 7–104
Jovial Unresolved Copy File 7–104
Jovial Unresolved Macro 7–104
Jovial Unresolved Subroutine 7–104
Jovial Unresolved Type 7–105
Jovial Unresolved Variable 7–105
JOVIAL Reference Kinds 7–106
Jovial Call and Jovial Callby 7–107
Jovial CompoolAccess/Accessby All 7–108
Jovial CompoolAccess and Jovial CompoolAccessby . 7–108
Jovial CompoolFileAccess/Accessby 7–109
Jovial ItemAccess and Jovial ItemAccessby 7–109
Jovial ItemAccess All and Jovial ItemAccessby All 7–110
Jovial ItemAccess/Accessby Implicit 7–110
Jovial Copy and Jovial Copyby 7–111
Jovial Declare and Jovial Declarein 7–111
Jovial Define and Jovial Definein 7–111
Jovial Declare Inline and Jovial Declarein Inline 7–112
Jovial Like and Jovial Likeby 7–112
Jovial Overlay and Jovial Overlayby 7–113
Jovial Overlay Implicit and Jovial Overlayby Implicit . 7–113
Jovial Typed Ptr and Jovial Typedby Ptr 7–113
Jovial Set and Jovial Setby 7–114
Jovial Set Init and Jovial Setby Init 7–114
Jovial Typed and Jovial Typedby 7–114
Jovial Use and Jovial Useby 7–115
Jovial Value and Jovial Valueof 7–115
Pascal Entity Kinds 7–116
Pascal Constant 7–118
Pascal CompUnit Module 7–118
Pascal CompUnit Program 7–118

Pascal Enumerator 7–118
Pascal Environment 7–119
Pascal Field 7–119
Pascal Field Discrim 7–119
Pascal File 7–119
Pascal File Include 7–119
Pascal Routine Function 7–120
Pascal Routine Function Asynchronous 7–120
Pascal Parameter Function 7–120
Pascal Parameter Procedure 7–120
Pascal Parameter Value 7–120
Pascal Parameter Var 7–121
Pascal Predeclared Routine 7–121
Pascal Predeclared Type 7–121
Pascal Predeclared Variable 7–121
Pascal Routine Procedure 7–121
Pascal Routine Procedure Asynchronous 7–122
Pascal Routine Procedure Local Initialize 7–122
Pascal Type Unnamed Local 7–122
Pascal Type 7–122
Pascal Variable 7–122
Pascal Unknown Environment 7–123
Pascal Unknown Variable 7–123
Pascal Unknown Function 7–123
Pascal Unknown Type 7–123
Pascal Unresolved Environment 7–123
Pascal Unresolved Global Entity 7–123
Pascal Unresolved Global Function 7–123
Pascal Unresolved Global Procedure 7–124
Pascal Unresolved Global Variable 7–124
Pascal Sql Alias 7–124
Pascal Sql Column 7–124
Pascal Sql Cursor 7–124
Pascal Sql Group 7–125
Pascal Sql Index 7–125
Pascal Sql Procedure 7–125
Pascal Sql Role 7–125
Pascal Sql Rule 7–125
Pascal Sql Statement 7–125
Pascal Sql Table 7–126
Pascal Sql Table GlobalTemp 7–126
Pascal Sql User 7–126
Pascal Sql Unresolved 7–126
Pascal Sql Unresolved Table 7–126
Pascal Reference Kinds 7–127

Pascal Call and Pascal Callby 7–128
Pascal Contain and Pascal Containin 7–129
Pascal Declare and Pascal Declarein 7–129
Pascal Define and Pascal Definein 7–130
Pascal End and Pascal Endby 7–130
Pascal Inherit and Pascal Inheritby 7–131
Pascal Inheritenv and Pascal Inheritenvby 7–131
Pascal Hasenvironment/Hasenvironmentby 7–132
Pascal Set and Pascal Setby 7–132
Pascal Set Init and Pascal Setby Init 7–132
Pascal Typed and Pascal Typedby 7–133
Pascal Use and Pascal Useby 7–133
Pascal Sql Call and Pascal Sql Callby 7–133
Pascal Sql Call/Callby Statement 7–134
Pascal Sql Define and Pascal Sql Definein 7–134
Pascal Sql Set and Pascal Sql Setby 7–135
Pascal Sql Typed and Pascal Sql Typedby 7–135
Pascal Sql Use and Pascal Sql Useby 7–135

Chapter 1 APIs to Understand Databases

All language-specific versions of *Understand* include both a PERL and a C/C++ Application Programming Interface (API) that can be used to create custom scripts or programs to access Understand databases. This chapter provides an overview of the two APIs.

Section	Page
What Can the Understand APIs Do?	1–2
Which API Should I Use?	1–2

What Can the Understand APIs Do?

Using the Perl or C/C++ API, you can write scripts and programs to to access Understand databases. These scripts and programs can generate custom documentation for your *Understand* databases. The APIs give you the same access we used in developing *Understand* reports and metrics. With either API you can answer many questions specific to your organization, and generate custom documentation with any look and format you prefer.

Note that the APIs provide read-only access to *Understand* databases. The databases cannot be changed by API functions.

Which API Should I Use?

If you are familiar with both Perl and C/C++, the preferred Understand API is the Perl interface. Perl is recommended for the following reasons:

- The Perl API is easier to use if you know both C and Perl.
- The Perl API is used by Scientific Toolworks to create sample scripts. Numerous example scripts in Perl can be found at http://www.scitools.com/perl.shtml.
- The Perl API is built into every *Understand* distribution. The C/C++ API is a separate download.
- The Perl API is technically more advanced than the C/C++ API (though not by a huge margin).
- The Perl API supports integration with Visio on Windows systems.

Chapter 2 Using the Perl API

This chapter describes how to use the Perl Application Programming Interface (API) to access Understand databases and the Understand environment. This manual does not describe how to use Perl itself. For information about Perl scripting, see a book or online resource regarding Perl.

Section	Page
Running Perl Scripts	2–2
Understand Package Classes	2–4
Scripting with the Perl API	2–5

Chapter 3, “Perl API Class and Method Reference” contains detailed information for all API classes and methods and is intended to be used as a reference.

Chapter 7, “Entity and Reference Kinds” contains lists of entity and reference kinds for use with the C API.

Running Perl Scripts

All language-specific versions of *Understand* include a Perl package called "Understand". This package provides a set of Perl classes and methods that you can use to create Perl scripts with read-only access to *Understand* databases and the *Understand* environment. Note that *Understand* databases cannot be changed by API functions.

This chapter provides an overview of the Understand package. This manual does not describe how to use Perl itself. For information about Perl scripting, see a book or online resource regarding Perl.

Chapter 3, "Perl API Class and Method Reference" provides a complete list of the classes and methods in the Understand package.

Understand License Requirements

To use the Understand package, you must have a proper license. Currently, this means you must have a user- or host-locked license or an available floating license for the *Understand* product that matches the database you are attempting to analyze. That is, for example, you must have an *Understand for Java* license in order to analyze the database for a Java project.

A Perl script may specify a license regcode, file, or directory using the Understand::license(filename) method. However, this is usually unnecessary, because the license file is found automatically by performing checks in the following sequence:

1. The Understand::license(name) method
2. The $STILICENSE environment variable
3. The conf/license subdirectory of an installed *Understand* product.

Perl Version

The "Understand" Perl package allows you to query Understand databases. It is an extension, which means it uses a shareable library written, in this case, in C++. It was created with Perl 5.6.0. It is likely this package will work with some earlier versions of Perl, perhaps as early as version 5.0; however, it is not guaranteed.

Running Scripts with uperl

The *Understand* installation includes a special compilation of Perl called "uperl". This Perl binary file is located in *Understand*'s binary directory. For example, on Windows it is the uperl.exe file in C:\Program Files\STI\bin\pc-win95.

The uperl program is ready to use once *Understand* is installed. Any needed packages (Understand or other) are found automatically in *Understand*'s installation directory.

To run scripts, simply run them with uperl. For example:

```
uperl metrics.pl
```

Note that uperl includes only a subset of Perl's standard libraries. This is done to reduce size. If you write a script that requires more Perl libraries, contact Scientific Toolworks or install the Understand package for use with a full Perl installation as described in the following section.

Running Scripts with Other Perl Installations

Follow these steps only if you wish to use *Understand* with an existing Perl installation. For example, this may be ActivePERL on Windows or the existing Perl installation that comes with many UNIX systems.

The Understand package may be used directly from the location where it is installed, or it may be installed into the system's Perl installation. By default, it is located in the Perl directory within Understand's system-specific bin directory. For example, on Windows it is in C:\Program Files\STI\bin\pc-win95\Perl.

To use it in its current location, use one of the following methods:

- Set the PERLLIB environment variable as follows:

  ```
  % setenv PERLLIB /stihome/bin/pc-win95/Perl/STI
  ```

- Add a command similar to the following at the top of your Perl scripts (where the directory appropriate for your platform and installation location is used):

  ```
  use lib "/stihome/bin/sun4-solaris_2.5/Perl/STI";
  ```

One could install the Understand package directly in the Perl setup. This can be useful, as Understand would be available within all Perl scripts, without the need for an environment variable or a "use lib" command. However, this should only be done by someone with a fair amount of knowledge about administering Perl installations.

Understand's DLL cannot be used with the Cygwin PERL implementation (which doesn't like standard Windows DLLs). For those users needing a full version of PERL (versus UPerl which is smaller), we recommend ActivePERL (http://www.activeperl.com).

Understand Package Classes

The Understand package provides a Perl class-oriented interface to Understand databases. To make this package available within a script, add a use command like the following:

```
use Understand;
```

The following classes provide methods for various types of access to Understand databases:

- *Understand::Db class* on page 3–4
- *Understand::Ent class* on page 3–6
- *Understand::Gui class* on page 3–15
- *Understand::Kind class* on page 3–18
- *Understand::Lexeme class* on page 3–20
- *Understand::Lexer class* on page 3–22
- *Understand::Metric class* on page 3–23
- *Understand::Ref class* on page 3–23
- *Understand::Util class* on page 3–25
- *Understand::Visio class* on page 3–25

Scripting with the Perl API

Numerous sample Perl scripts that access Understand databases are provided in the following locations:

- In your *Understand* installation in the \sample\scripts directory
- http://www.scitools.com/perl.shtml

Traversing an *Understand* Database

Using the Perl API to access the *Understand* database is generally done in the following way:

- Open *Understand* database
- Retrieve a filtered list of entities
- Retrieve desired information about each entity in the filtered list Entity Info includes:
  - names, types, kinds, etc.
  - metric values
  - reference information
- Close the database

Starting a Script

Scripts should contain a use command that specifies the Understand package:

```
use Understand;
```

Opening a Database

A database is opened with the following command:

```
($db, $status) = Understand::open($name);
```

If the open fails, $status will be defined with a string indicating the kind of failure. On a successful open, the returned $db will be an object from the class Understand::Db. For example, to test the returned status:

```
($db, $status) = Understand::open("test.udc");
die "Error status: ",$status,"\n" if $status;
```

See *Understand::open()* on page 3–2 for detailed information.

Only one database may be opened at a time. An error is returned if you attempt to open a second, simultaneous database.

Opening a database consumes an *Understand* license.

Closing a Database

An open database may be closed with the command $db->close().

See *$db->close()* on page 3–4 for detailed information.

Getting a List of Entities

A list of all entities, such as files, functions and variables, may be obtained from an Understand::Db object with the command $db->ents(). The list is unsorted. All entities returned are objects of the class Understand::Ent.

```
foreach $ent ($db->ents()) {
   # print entity and its kind
   print $ent->name()," [",$ent->kindname(),"]\n";
}
```

See *$db->ents()* on page 3–4 for detailed information.

The returned list may be refined with a filter that specifies the kind of entities desired. For example, $db->ents("File") returns just file entities.

```
foreach $file ($db->ents("File")) {
   # print the long name including directory names
   print $file->longname(),"\n";
}
```

See Chapter 7, “Entity and Reference Kinds” for lists of entity kinds for each programming language supported by *Understand*. In particular, see *Using Kinds in the Perl API* on page 7–2 for a list of methods that deal with entity kinds.

You can also filter the entities returned by name using the $db->lookup method. For example:

```
# find all 'File' entities that match test*.cpp
foreach $file ($db->lookup("test*.cpp","File")) {
   print $file->name(),"\n";
}
```

Getting Entity Attributes

Once you have gotten an entity (or a group of entities), you can obtain all kinds of information about it. There are a variety of attributes available for an Understand::Ent object.

```
foreach $func ($db->ents("Function")) {
   print $func->longname(),"(";
   $first = 1;
   # get list of refs that define a parameter entity
   foreach $param ($func->ents("Define","Parameter")) {
     print ", " unless $first;
     print $param->type()," ",$param->name;
     $first = 0;
  }
    print ")\n";
}
```

The command $ent->name() returns the name of the entity, while $ent->longname() returns a long name, if available. Examples of entities with long names include files, C++ members, and most Ada entities. See *$ent->name()* on page 3–13 and *$ent->longname()* on page 3–12 for detailed information.

If an entity has a type or return type associated with, for example a variable, type or function, the type may be determined with the command $ent->type(). See *$ent->type()* on page 3–14 for detailed information.

The kind of an entity, such as File or Function, may be determined with the command $ent->kindname(). See *$ent->kindname()* on page 3–11 for detailed information.

If desired, the command $ent->kind() may be used instead, which returns an object of the class Understand::Kind. This is sometimes useful when more detailed information about the kind is required. See *$ent->kind()* on page 3–11 for detailed information.

Getting a List of References

A list of references for an entity may be obtained from an Understand::Ent object with the command $ent->refs(). All references returned are objects of the class Understand::Ref.

The returned list may be refined with a filter that specifies the kind of references desired. For example, $ent->refs("Define") will return definition references.

The list may be even further refined with a second filter that specifies the kind of referenced entities desired. For example, @refs=$ent->refs("Define","Parameter") will return just definition references for parameter entities.

A final parameter with value 1 may be used to specify that only unique entities should be returned. For example, @refs=$ent->refs("Call","Function",1) returns a list of references to called functions, where only the first reference to each unique function is returned.

See *$ent->refs()* on page 3–13 for detailed information.

See Chapter 7, “Entity and Reference Kinds” for lists of reference kinds for each programming language supported by *Understand*. In particular, see *Using Kinds in the Perl API* on page 7–2 for a list of methods that deal with reference kinds.

In addition to getting general entity and metrics information, you can also get information about the references to or from an entity. Again, you obtain a list of all references for a particular entity, and then filter the list to include only those types of references that you are interested in. For example:

```
foreach $var ($db->ents("Global Object ~Static")) {
  print $var->name(),":\n";
  foreach $ref ($var->refs()) {
    printf "  %-8s %-16s %s (%d,%d)\n",
      $ref->kindname(),
      $ref->ent()->name(),
      $ref->file()->name(),
      $ref->line(),
      $ref->column();
  }
  print "\n";
}
```

Getting Associated Comments

If associated comments have been stored in the database, they may be retrieved for an entity. Only the C and Ada versions of *Understand* currently store information about associated comments. See *$ent->comments()* on page 3–6 for detailed information.

```
foreach $func ($db->ents("function ~unresolved ~unknown")) {
   @comments = $func->comments("before");
   if (@comments) {
      print $func->longname(),":\n";
```

```
        foreach $comment (@comments) {print "  ",$comment,"\n";}
         print "\n";
    }
}
```

To associate comments when the C and Ada versions of *Understand*, use the -comment command line option or the Associate Comments setting in the **Options** tab of the **Project->Configure** dialog.

Associated comments are comments that occur near the declaration of an entity in source code. Some entity kinds have different kinds of declarations, which can be explicitly specified. Also, comment position, before or after the declaration, can be specified.

The returned comments can be a formatted string (the default) or may be an array of raw comment strings.

Getting Project Metrics

Metric values associated with the entire database or project are available for Understand::Db objects.

The command $db->metrics() returns a list of all available project metric names. The command $db->metric(@mets) returns a list of values for specific metrics. For example:

```
# loop through all project metrics
foreach $met ($db->metrics()) {
   print $met," = ",$db->metric($met),"\n";
}
```

See *$db->metrics()* on page 3–5 for detailed information.

Getting Entity Metrics

Metric values associated with a specific entity are available for Understand::Ent objects.

The command $ent->metrics() returns a list of all available entity metric names. The command $ent->metric(@mets) returns a list of values for specific metrics.

```
# lookup a specific metric
foreach $func ($db->ents("Function")) {
   $val = $func->metric("Cyclomatic");
   # only if metric is defined for entity
   print $func->name()," = ",$val,"\n" if defined($val);
}
```

See *$ent->metrics()* on page 3–13 for detailed information.

Getting Info Browser Text

The text for Info Browser views of entities may be created using the command $ent->ib().

```
# print Info Browser view of all functions
foreach $func ($db->ents("Function")) {
   print $func->ib(),"\n";
}
```

See *$ent->ib()* on page 3–10 for detailed information.

Saving Graphics Files

Graphical views of entities may be created and saved as jpg or png files, using the command $ent->draw().

```
# loop through all functions
foreach $func ($db->ents("Function")) {
   # save png file of Callby graphs of functions
   $file = "callby_" . $func->name() . ".png";
   print $func->longname(), " -> ", $file,"\n";
   $func->draw("callby",$file);
}
```

On Windows systems that have Visio installed, vsd files may also be created.

See *$ent->draw()* on page 3–7 and *Visio::draw()* on page 3–25 for detailed information.

Lexer

The Lexer class provides several methods that return objects of class Understand::Lexeme. A lexeme is the smallest unit of language in a source file that has a meaning. For example, a comment, a string, a keyword, or an operator.

The $ent->lexer() method returns a lexical stream as an object of the Understand::Lexer class. A lexical stream contains all the lexemes for the specified entity. A lexical stream may be generated for a file entity only if the original file exists and is unchanged from the last database reparse.

Individual lexemes (tokens) may be accessed from a lexer object, either sequentially with $lexer->first() and $lexeme->next(), or as an array with $lexer->lexemes().

Each lexeme object indicates its token kind ($lexeme->token()), its text ($lexeme->text()), its referenced entity ($lexeme->ent()) and its line and column position.

```
# lookup file entity, create lexer
$file = $db->lookup("test.cpp");
$lexer = $file->lexer();
# regenerate source file from lexemes
# add a '@' after each entity name
foreach $lexeme ($lexer->lexemes()) {
   print $lexeme->text();
   if ($lexeme->ent()) {
      print "@";
   }
}
```

See *Understand::Lexeme class* on page 3–20 and *Understand::Lexer class* on page 3–22 for detailed information.

Using the GUI Class

When a script is being run from within the Understand application, the Gui class becomes available. This class gives access to the current open database and information about the cursor position and current selection for the file being edited.

```
# This script is designed to run within the
# Understand application
use Understand;
die "Must run within Gui" if !Understand::Gui::active();
die "Must run with db open" if !Understand::Gui::db();

my $db = Understand::Gui::db();
printf("Database: %s\n",$db->name());
my $filename = Understand::Gui::filename();
my $col      = Understand::Gui::column();
my $line     = Understand::Gui::line();
printf("File '%s' [%d,%d]\n",$filename,$line,$col)
             if ($filename);
my $entity = Understand::Gui::entity();
printf("Entity '%s'\n",$entity->name()) if $entity;
my $selection = Understand::Gui::selection();
my $word = Understand::Gui::word();
printf("Selection '%s'\n",$selection) if $selection;
printf("Word '%s'\n",$word) if $word;
```

See *Understand::Gui class* on page 3–15 for detailed information.

Chapter 3 Perl API Class and Method Reference

This chapter provides detailed references for each Perl API function available to access the *Understand* database. It is recommended that you begin by reading the introductory sections of the proceeding chapter before using this chapter.

This chapter contains the following sections:

Section	Page
Understand Package	3–2
Understand::Db class	3–4
Understand::Ent class	3–6
Understand::Gui class	3–15
Understand::Kind class	3–18
Understand::Lexeme class	3–20
Understand::Lexer class	3–22
Understand::Metric class	3–23
Understand::Ref class	3–23
Understand::Util class	3–25
Understand::Visio class	3–25

Understand Package

The Understand package provides a Perl class-oriented interface to Understand databases. To make this package available, scripts should contain the following command:

```
use Understand;
```

The base Understand class provides the following methods:

- *Understand::license()* on page 3–2
- *Understand::open()* on page 3–2
- *Understand::version()* on page 3–3

Understand::license()

Syntax `Understand::license(name);`

Description Specify a regcode string, or a specific path to an Understand license. This method is usually unnecessary.

See Also *Understand License Requirements* on page 2–2

Understand::open()

Syntax `($db, $status) = Understand::open(path,[display]);`

Description Open a database. Returns ($db, $status).

Arguments The path argument specifies the database filename.

The display argument may be specified to change the display mode for Ada, FORTRAN, Pascal, and JOVIAL. If it is not specified, the default, as stored in the database is used. Display must be one of the following:

- original - The case of all entities will be as originally specified in source code.
- upper - The case of all entities will be changed to all uppercase.
- lower - The case of all entities will be changed to all lowercase.
- first - The case of all entities will be changed so just the first character is capitalized.
- mixed - The case of all entities will be changed so the first character of each word is capitalized.

Return Values

$db is an object of the class Understand::Db.

If no error occurs, $status is undef. Otherwise, $status is:

Return Values	Description
DBAlreadyOpen	Only one database may be open at once
DBCorrupt	Sorry, bad database file
DBOldVersion	Database needs to be rebuilt
DBUnknownVersion	Database needs to be rebuilt
DBUnableOpen	Database is unreadable or does not exist
NoApiLicenseAda	Ada license required
NoApiLicenseC	C license required
NoApiLicenseFtn	FORTRAN license required
NoApiLicenseJava	Java license required
NoApiLicenseJovial	JOVIAL license required
NoApiLicensePascal	Pascal license required

Example

```
($db, $status) = Understand::open("test.udc");
die "Error status: ",$status,"\n" if $status;
```

See Also

Opening a Database on page 2–5

Understand::version()

Syntax

```
$vers = Understand::version();
```

Description

Returns the build number for the current installed uperl module. This can be used by scripts to make sure they are being run on the minimum build necessary, or to change their behavior to support limited functionality in older builds.

See Also

Perl Version on page 2–2

Understand::Db class

The Understand::open() method returns an object, $db, of the Understand::Db class. The Understand::Db class provides a number of methods that operate on such objects.

$db->check_comment_association()

Syntax `$db->check_comment_association()`

Description Returns undef if comment association is disabled. Otherwise, returns the currently selected comment association style (after, before, longest).

See Also *Getting Associated Comments* on page 2–8

$db->close()

Syntax `$db->close()`

Description Closes a database so that another database may be opened.

See Also *Closing a Database* on page 2–6

$db->ents()

Syntax `$db->ents( [$kindstring] )`

Description Returns a list of entities. Each returned entity is an object in the class Understand::Ent.

Arguments If the optional argument $kindstring is not passed, then all entities in the database are returned. Otherwise, $kindstring should be a language-specific entity filter string. See Chapter 7 for lists of kinds.

See Also *Getting a List of Entities* on page 2–6

$db->language()

Syntax `$db->language()`

Description Returns a string indicating the language of the database. Possible return values include “Ada”, “C”, “Fortran”, “Java”, “Pascal”, and “Jovial”.

$db->last_id()

Syntax `$db->last_id()`

Description Returns the last (maximum) entity id for a database.

$db->lookup()

Syntax `$db->lookup($name [,$kindstring] [,$case])`

Description Returns a list of entities that match the specified $name.

Arguments In the $name argument, the special character '?' may be used to indicate a match of any single character and the special character '*' may be used to indicate a match of 0 (zero) or more characters.

If the optional argument $kindstring is passed, it should be a language-specific entity filter string. See Chapter 7 for lists of kinds.

If the optional argument $case is passed, it should be 0 to mean case-insensitive and 1 to mean case-sensitive lookup. The default is case-insensitive.

See Also *Getting a List of Entities* on page 2–6

$db->lookup_uniquename()

Syntax `$db->lookup_uniquename($uniquename)`

Description Returns the entity identified by uniquename, or undef if no entity is found.

Arguments $uniquename is the name returned by $ent->uniquename() (see page 3-14).

$db->metric()

Syntax `$db->metric(@metriclist)`

Description Returns a project metric value for each specified metric name in @metriclist.

See Also *Getting Project Metrics* on page 2–9

$db->metrics()

Syntax `$db->metrics()`

Description Returns a list of all project metric names.

See Also *Getting Project Metrics* on page 2–9

$db->name()

Syntax `$db->name()`

Description Returns the filename of the database.

Understand::Ent class

The $db->ents() and $ent->ents() methods return objects of class Understand::Ent. The Understand::Ent class provides a number of methods that operate on such objects.

Entity objects may be compared using the <,=,>,<=, and >= operators.

$ent->comments()

Syntax

```
$ent->comments( [$style [,$format [,$refkindstring]]] )
```

Description

Returns a formatted string based on the comments associated with an entity. Only the C and Ada versions of *Understand* currently store information about associated comments.

Arguments

The optional argument $style specifies which comments are to be used. By default, comments after the entity declaration are processed. The valid values for $style are:

Argument Values	Description
after	Process comments after the entity declaration. (This is the default.)
before	Process comments before the entity declaration.
longest	Process “before” or “after” comments, whichever are longest.

The optional argument $format is used to specify what kind of formatting, if any, is applied to the comment text. The valid values for $format are shown in the following table:

Argument Values	Description
<no value>	Removes comment characters and certain repeating characters, while retaining the original newlines. (This is the default.)
raw	Return an array of comment strings in original format, including comment characters.

If the optional $refkindstring argument is specified, it should be a language-specific reference filter string. See Chapter 7 for lists of kinds.

Example

```
@funcs = $db->ents("function ~unresolved ~unknown,
                    procedure");
foreach $func (sort {lc($a->longname()) cmp
                     lc($b->longname());} @funcs) {
  if ($func->library() ne "Standard") {
    $comment = $func->comments("before");
    if ($comment) {
        print "-----\n";
        print $func->longname(),":\n";
        print $comment,"\n";
        print "-----\n\n";
    }
  }
}
```

$ent->draw()

Syntax

```
$ent->draw( $kind, $filename [,$options] )
```

Description

Generates a graphics file for an entity. JPG and PNG file formats are supported on all platforms. Visio file output is supported only on Windows systems where Visio is installed.

Arguments

The $kind argument should specify the kind of view to generate for the entity. The valid $kind options match the "Graphical Menu Strings" listed for the -gv and -gvnew command options in the "Server Mode" chapter of the Understand manual for your programming language. For the Perl language, these are the options:

Languages	Values
Ada	Callby, Child Lib Units, Declaration, Declaration Tree, Declared In, Instantiated From, Instantiations, Invocation, Parent Declaration, Parent Lib Unit, Rename Declaration, Type Derived From, Type Tree, With, Withby
C/C++	Base Classes, Callby, Data Members, Declaration, Declaration File, Declaration Type, Derived Classes, Include, Includeby, Invocation, Parent Declaration, Return Type
FORTRAN	Callby, Declaration, Declaration File, Include, Includeby, Invocation, Usedby, Uses
Java	Callby, Contains, Declaration, Declaration File, Declaration Type, Extends, Extended By, Invocation, Parent Declaration, Return Type

Languages	Values
JOVIAL	Callby, Declaration, Declaration File, Declaration Type, Invocation, Parent Declaration, Return Type
Pascal	Callby, Declaration, Declaration File, Include, Includeby, Invocation

The $filename argument must end with and extension supported on the current platform. That is, either .jpg or .png is supported on all platforms. The .vsd extension is supported only on Windows systems where Visio is installed.

The $options argument may be used to specify some aspects of how to generate the graphics. The format of the $options string is "name=value". Multiple options may be separated with a semicolon.

N[illegible] options are available for all programming lang[illegible]ges and all [illegible]aphical view types. For a list of th[illegible] options, op[illegible] *nderstand* [illegible]r y[illegible]ur [illegible], then ch[illegible]se **ptions->Graphical Settings** and go to the **View [illegible]ptions** tab. [illegible] are shown.

An options string may contain options that are specific for different views, using the syntax "[viewname]optionname=value". This permits you to, for instance, turn fullnames on for Invocation Trees, but not for CallBy trees.

Names and values may be abbreviated to any unique prefix of their full names. For example, you could use the following options string:

```
"layout=cross; scale=14"
```

Additionally, you may use “font” as a special option for any graph type. Its value must be the path to a TrueType font (.ttf) for JPG and PNG files, or the name of a system font for Visio .vsd files.

Using the .vsd extension causes Visio to be invoked, to draw the graphics, and to save the drawing to the named file. Visio remains running, but may be closed by calling Understand::Visio::quit().

Return Values

One of the following status strings is returned if an error occurs:

Status String	Description
"NoFont"	No suitable font can be found.
"NoImage"	No image is defined or the image is empty.
"NoVisioSupport"	No Visio .vsd files can be generated on this system.
"TooBig"	JPG does not support a dimension greater than 64 K.

Status String	Description
"UnableCreateFile"	File cannot be opened or created.
"UnsupportedFile"	Only .jpg and .png files are supported.

Additional error messages are also possible when generating a Visio file.

See Also *Understand::Visio class* on page 3–25

$ent->ents()

Syntax

```
$ent->ents( $refkindstring [,$entkindstring] )
```

Description Returns a list of entities that reference, or are referenced by, the entity. Each returned entity is an object in the class Understand::Ent.

Arguments The $refkindstring argument should be a language-specific reference filter string. See Chapter 7 for lists of kinds.

If the optional argument $entkindstring is not passed, then all referenced entities are returned. The $entkindstring argument may be a language-specific entity filter string that specifies what kind of referenced entities are to be returned.

$ent->filerefs()

Syntax

```
$ent->filerefs( [$refkindstring [,$entkindstring
                  [, $unique ]]] )
```

Description Returns a list of all references that occur in the specified file entity. These references will not necessarily have the file entity for their ->scope value. Each returned reference is an object in the Understand::Ref class (see page 3-23).

Arguments If the optional argument $refkindstring is not passed, then all references are returned. Otherwise, $refkindstring should be a language-specific reference filter string.

If the optional argument $entkindstring is not passed, then all references to any kind of entity are returned. Otherwise, $entkindstring may be a language-specific entity filter string that specifies references to what kind of referenced entity are to be returned. See Chapter 7 for lists of kinds.

If the optional argument $unique is passed with value 1, only the first matching reference to each unique entity is returned.

$ent->ib()

Syntax

```
$ent->ib( [,$options] )
```

Description

Returns a list of lines of text, representing the Info Browser information for an entity.

Arguments

An $options string may be used to specify some arguments used to create the text. The format of the options string is "name=value" or "{field-name}name=value". Multiple options may be separated with a semicolon. Spaces are allowed and are significant between multi-word field names, whereas, case is not significant.

An option that specifies a field name is specific to that named field of the Info Browser. The available field names are exactly as they appear in the Info Browser.

When a field is nested within another field, the correct name is the two names combined. For example, in C++, the field Macros within the field Local would be specified as "Local Macros".

A field and its subfields may be disabled by specifying levels=0, or by specifying the field off, without specifying any option. For example, either of the following disables and hides the Metrics field:

```
{Metrics}levels=0;
{Metrics}=off;
```

Field names may be full names or short names. For example, either “c++ metrics” or “metrics” may be used as a field name.

The following option is available only without a field name.

- **Indent** - this specifies the number of indent spaces to output for each level of a line of text. The default is 2.

The following options are available only with a field name. Not all options are available for all field names. The available options are the same as those displayed when you right-click on the field name in the *Understand* software. The defaults differ for each language and each field name.

- **Defnfile** - How to display filenames. Values are “Short”, “Long”, “Relative”, and “Off”.
- **Fullnames** - Whether to display full entity names. Values are “On” and “Off”.
- **Levels** - How to display nested fields. Values are -1 to show all nested fields, 0 to hide the field name and all nested fields, or a positive number to show up to that number of nesting levels.

- **Parameters** - Whether to display formal parameters of functions when appropriate. Values are “On” and “Off”.
- **References** - Whether to display file/line information and multiple references. Values are “On” and “Off”.
- **Returntypes** - Whether to display function return types when appropriate. Values are “On” and “Off”.
- **Sort** - Whether to sort information by name (“On”) or by reference file and line number (“Off”).
- **Types** - Whether to displays types of entities when appropriate. Values are “On” and “Off”.

$ent->id()

Syntax `$ent->id()`

Description Returns a numeric identifier that is unique for each underlying database entity. The identifier is not guaranteed to remain consistent after the database has been updated.

An id can be converted back to an object of the Understand::Ent class with Understand::Ent->new($id). The Ent->new() method is only used in this special case.

$ent->kind()

Syntax `$ent->kind()`

Description Returns a kind object for the entity. The returned object is of the Understand::Kind class (see page 3-18).

$ent->kindname()

Syntax `$ent->kindname()`

Description Returns a simple name for the kind of the entity. This is equivalent to $ent->$kind->name().

$ent->language()

Syntax `$ent->language()`

Description Returns a string indicating the language of the entity. Possible return values include “Ada”, “C”, “Fortran”, “Pascal”, “Java”, and “Jovial”.

$ent->lexer()

Syntax `$ent->lexer()`

Description Returns a lexer object of the Understand::Lexer class (see page 3-22) for the specified file entity. The original source file must be readable and unchanged since the last database parse.

Return Values If called in an array context, this method returns ($lexer, $status). If no error occurs, $status is undef. Otherwise, $status is:

Return Values	Description
FileModified	The file has been modified since the last parse.
FileUnreadable	The file is not readable at the original location.

$ent->library()

Syntax `$ent->library()`

Description Returns the name of the library that the entity belongs to, or undef if it does not belong to a library.

Currently, the only supported library is “Standard”, to which the following items belong:

- Java JDK entities
- Predefined Ada entities such as “text_io”

$ent->longname()

Syntax `$ent->longname()`

Description Returns the long name of the entity.

If there is no long name defined, the regular name ($ent->name()) is returned. Examples of entities with long names include files, C++ members and most Ada entities.

See Also *$ent->name()* on page 3–13

$ent->metric()

Syntax `$ent->metric(@metriclist)`

Description Returns a metric value for each specified metric name in @metriclist.

See Also *Understand::Metric class* on page 3–23

$ent->metrics()

Syntax `$ent->metrics()`

Description Returns a list of all metric names that are defined for the entity.

$ent->name()

Syntax `$ent->name()`

Description Returns the simple (short) name for the entity.

See Also *$ent->longname()* on page 3–12

$ent->parameters()

Syntax `$ent->parameters()`

Description Returns the parameters for an entity, in textual form.

$ent->refs()

Syntax
```
$ent->refs( [$refkindstring [,$entkindstring
            [, $unique]]] )
```

Description Returns a list of references. In a scalar context, only the first reference is returned. Each returned reference is an object in the Understand::Ref class (see page 3-23).

If the optional argument $refkindstring is not passed, then all references for the entity are returned. Otherwise, $refkindstring should be a language-specific reference filter string. If the optional argument $entkindstring is not passed, then all references to any kind of entity are returned.

Otherwise, $entkindstring should be a language-specific entity filter string that specifies references to what kind of referenced entity are to be returned. See Chapter 7 for lists of kinds.

If the optional argument $unique is passed with value 1, only the first matching reference to each unique entity is returned.

See Also *$ent->filerefs()* on page 3–9

$ent->ref()

Syntax `$ent->ref( [$refkindstring [,$entkindstring]] )`

Description Returns the first reference for the entity. This is really the same as calling $ent->refs() from a scalar context. See Chapter 7 for lists of kinds.

See Also *$ent->filerefs()* on page 3–9

$ent->relname()

Syntax `$ent->relname()`

Description Returns the relative name of the file entity. Returns the fullname for the file, minus any root directories that are common for all project files. Returns undef for non-file entities.

$ent->type()

Syntax `$ent->type()`

Description Returns the type string of the entity. This is defined for entity kinds like variables and type, as well as entity kinds that have a return type like functions.

$ent->uniquename()

Syntax `$ent->uniquename()`

Description Returns the uniquename of the entity. This name is not suitable for use by an end user. Rather, it is a means of identifying an entity uniquely in multiple databases, perhaps as the source code changes slightly over time. The uniquename is composed of things like arguments and parent names. So, some code changes will result in new uniquenames for the same intrinsic entity.

Use $db->lookup_uniquename() (see page 3-5) to convert a uniquename back to an object of the Understand::Ent class.

Understand::Gui class

When a script is being run from within the Understand application, the Understand::Gui class becomes available.

This class provides methods that give access to the current open database and information about the cursor position and current selection for the file being edited. There are no objects of this class.

Gui::active()

Syntax

```
Understand::Gui::active()
```

Description

Returns true if the script has been called from within the *Understand* software application. No other functions in this class are available if this is not true.

Gui::column()

Syntax

```
Understand::Gui::column()
```

Description

Returns the column of the cursor in the current file being edited, or returns 0 (zero) if no file is being edited.

Example

```
my $col      = Understand::Gui::column();
```

Gui::db()

Syntax

```
Understand::Gui::db()
```

Description

Returns the current database. This database must not be closed.

Example

```
my $db = Understand::Gui::db();
printf("Database: %s\n",$db->name());
```

Gui::disable_cancel()

Syntax

```
Understand::Gui::disable_cancel()
```

Description

Disables the Cancel dialog when run from the *Understand* GUI.

Example

```
my $db = Understand::Gui::db();
printf("Database: %s\n",$db->name());
```

Gui::entity()

Syntax `Understand::Gui::entity()`

Description Returns the entity at the current cursor position. Returns undef if no file is being edited or if the cursor position does not contain an entity.

Example
```
my $entity = Understand::Gui::entity();
```

Gui::filename()

Syntax `Understand::Gui::filename()`

Description Returns the name of the current file being edited, or undef if no file is being edited.

Example
```
my $filename = Understand::Gui::filename();
```

Gui::line()

Syntax `Understand::Gui::line()`

Description Returns the line of the cursor in the current file being edited, or returns 0 (zero) if no file is being edited.

Example
```
my $line     = Understand::Gui::line();
```

Gui::progress_bar()

Syntax `Understand::Gui::progress_bar(percent [,allow_cancel])`

Description Displays the progress bar, if percent is 0.0 or greater. If it is less than 0.0, the progress bar is hidden, if it is currently displayed. If it is greater than 1.0, then 1.0 is assumed.

Arguments If $allow_cancel is specified, a value of 1 indicates a cancel button should be displayed on the progress bar. If allow_cancel is not specified, the default value is 0, which means no cancel button is displayed.

Return Values If the cancel button is displayed and the user presses it, this call returns 1. Otherwise, it returns 0.

Gui::selection()

Syntax `Understand::Gui::selection()`

Description Returns the selected text in the current file being edited. Returns 0 (zero) if no file is being edited or no text is selected.

Example

```
my $selection = Understand::Gui::selection();
```

Gui::word()

Syntax `Understand::Gui::word()`

Description Returns the word at the cursor position in the current file being edited. Returns 0 (zero) if no file is being edited.

Example

```
my $word = Understand::Gui::word();
```

Gui::yield()

Syntax `Understand::Gui::yield()`

Description Causes a potential yield event in the understand application, if it is needed. Normally, the following functions internally cause a yield:

- $db->ents() (see page 3-4)
- $db->metric() (see page 3-5)
- $ent->ents() (see page 3-9)
- $ent->lexer() (see page 3-12)
- $ent->metric() (see page 3-12)
- $ent->refs() (see page 3-13)

If these functions are not called for long periods of time, it may be desirable to call yield() directly, to allow the *Understand* software application environment to respond to external events, such as window repaints.

Understand::Kind class

You can use this module to extract information about a kind of entity.

The $ent->kind() method returns an object of class Understand::Kind. The Understand::Kind class provides a number of methods that operate on such objects. See *Using Kinds in the Perl API* on page 7–2 for a list of other methods that accept kind filters.

See Chapter 7, "Entity and Reference Kinds" for lists of entity and reference kinds for each programming language supported by *Understand*.

$kind->check()

Syntax `$kind->check($kindstring)`

Description Returns true if the kind matches the filter $kindstring.

$kind->inv()

Syntax `$kind->inv()`

Description Returns the logical inverse of the reference kind. Used for reference kinds only. For example, returns "Callby" if the kind is "Call". Invalid for entity kinds.

Kind::list_entity()

Syntax `@Understand::Kind::list_entity([entkind])`

Description Returns a list of entity longname kinds that match the $entkind filter. For example, the following returns all C function entity kinds:

```
my @kinds = Understand::Kind::list_entity("c function");
```

Kind::list_reference()

Syntax `@Understand::Kind::list_reference([refkind])`

Description Returns a list of reference longname kinds that match the $refkind filter. The following returns all Ada declare reference kinds:

```
my @kinds = Understand::Kind::list_reference("ada declare");
```

$kind->longname()

Syntax `$kind->longname()`

Description Returns the long form of the kind name. This is usually more detailed than desired for human reading.

$kind->name()

Syntax `$kind->name()`

Description Returns the name of the kind. This is equivalent to using $ent->kindname().

Understand::Lexeme class

A lexeme is the smallest unit of language in a source file that has a meaning. For example, a comment, a string, a keyword, or an operator.

You can use this module to extract information about an individual lexeme element. The $ent->lexer() method returns an object of the Understand::Lexer class (see page 3-22). The Lexer class provides several methods that return objects of class Understand::Lexeme.

$lexeme->column_begin()

Syntax `$lexeme->column_begin()`

Description Returns the beginning column of the lexeme.

$lexeme->column_end()

Syntax `$lexeme->column_end()`

Description Returns the ending column of the lexeme.

$lexeme->ent()

Syntax `$lexeme->ent()`

Description Returns the entity associated with the lexeme, or undef if none. The object returned is of the Understand::Ent class.

$lexeme->inactive()

Syntax `$lexeme->inactive()`

Description Returns true if the lexeme is part of inactive code.

$lexeme->line_begin()

Syntax `$lexeme->line_begin()`

Description Returns the beginning line of the lexeme.

$lexeme->line_end()

Syntax `$lexeme->line_end()`

Description Returns the ending line of the lexeme.

$lexeme->next()

Syntax `$lexeme->next()`

Description Returns the next lexeme, or undef if at end of file. The object returned is of the Understand::Lexeme class.

$lexeme->previous()

Syntax `$lexeme->previous()`

Description Returns the previous lexeme, or undef if at beginning of file. The object returned is of the Understand::Lexeme class.

$lexeme->ref()

Syntax `$lexeme->ref()`

Description Returns the reference associated with the lexeme, or undef if none. The object returned is of the Understand::Ref class.

$lexeme->text()

Syntax `$lexeme->text()`

Description Returns the text for the lexeme.

$lexeme->token()

Syntax `$lexeme->token()`

Description Returns the token kind of the lexeme. Values include:

- Comment
- Identifier
- Keyword
- Literal
- Newline
- Operator
- Preprocessor
- Punctuation
- String
- Whitespace

Understand::Lexer class

The Lexer class provides several methods that return objects of class Understand::Lexeme. A lexeme is the smallest unit of language in a source file that has a meaning. For example, a comment, a string, a keyword, or an operator.

The $ent->lexer() method returns an object of the Understand::Lexer class. A lexer object contains all the lexemes for the specified entity.

Using a lexeme and the methods of the Understand::Lexeme class (see page 3-20), you can find information about individual lexemes.

$lexer->first()

Syntax `$lexer->first()`

Description Returns the first lexeme for the lexer. The object returned is of the Understand::Lexeme class.

$lexer->lexeme()

Syntax `$lexer->lexeme($line,$column)`

Description Returns the lexeme that occurs at the specified line and column. The object returned is of the Understand::Lexeme class.

$lexer->lexemes()

Syntax `$lexer->lexemes([$start_line,$end_line])`

Description Returns an array of all lexemes.

Arguments If the optional arguments $start_line and $end_line are specified, only the lexemes within these lines are returned.

$lexer->lines()

Syntax `$lexer->lines()`

Description Returns the number of lines in the lexer.

Understand::Metric class

This class provides methods that give access to information about the metrics available for the programming language. There are no objects of this class.

Metric::description()

Syntax `Understand::Metric::description($metric)`

Description Returns the short description of the specified metric.

Metric::list()

Syntax `Understand::Metric::list([$kindstring])`

Description Returns a list of metric names.

Arguments If the optional argument $kindstring is not passed, then the names of all possible metrics are returned. Otherwise, only the names of metrics defined for entities that match the entity filter $kindstring are returned. See Chapter 7 for lists of kinds.

Understand::Ref class

You can use this module to extract information about references.

The $ent->filerefs(), $ent->ref(), and $lexeme->ref() methods return an object of class Understand::Ref. The Understand::Ref class provides a number of methods that operate on such objects.

$ref->column()

Syntax `$ref->column()`

Description Returns the column in the source file where the reference occurred.

$ref->ent()

Syntax `$ref->ent()`

Description Returns the entity being referenced. The returned entity is an object in the Understand::Ent class.

$ref->file()

Syntax `$ref->file()`

Description Returns the file where the reference occurred. The returned file is an object in the Understand::Ent class.

$ref->kind()

Syntax `$ref->()`

Description Returns a kind object of the Understand::Kind class for the reference.

$ref->kindname()

Syntax `$ref->kindname()`

Description Returns a simple name for the kind of the reference. This is equivalent to $ref->$kind->name().

$ref->line()

Syntax `$ref->line()`

Description Returns the line in the source file where the reference occurred.

$ref->scope()

Syntax `$ref->scope()`

Description Returns the entity performing the reference. The returned entity is an object in the Understand::Ent class.

Understand::Util class

The Understand::Util class provides utility methods.

Util::checksum()

Syntax
```
Understand::Util::checksum($text[,$len])
```

Description
Returns a checksum of the text.

Arguments
The optional argument $len specifies the length of the checksum, which may be between 1 and 32 characters, with 32 being the default.

Understand::Visio class

The Understand::Visio class provides methods that control the Visio environment if it is available.

Visio::draw()

Syntax
```
Understand::Visio::draw( $ent, $kind, $filename
                         [,$options] )
```

Description
On Windows systems only, this method invokes Visio, if it is not already running, generates a graphic for the specified entity, and saves the results to a .vsd file. Visio will continue running after this call (for efficiency reasons), but may be forced to quit using Understand::Visio::quit().

Arguments
For a description of the $kind and $options arguments, see *$ent->draw()* on page 3–7.

Visio::quit()

Syntax
```
Understand::Visio::quit()
```

Description
On Windows systems only, forces Visio to quit, if it is running because of a call to Visio::draw() or $ent->draw().

Chapter 4 Using the C API

This chapter provides an overview of API usage with some sample code. It is recommended that you begin by reading through the following introductory sections of this chapter:

Section	Page
Using the Understand C API	4–2
Getting Started with the C API	4–3
Code Sample Using the C API	4–7

Chapter 5, “C API Functions” contains detailed information for all C API functions and is intended to be used as a reference.

Chapter 6, “C API Code Samples” contains samples that use the C API.

Chapter 7, “Entity and Reference Kinds” contains lists of entity and reference kinds for use with the C API.

Using the *Understand* C API

The C API provides read-only access to the *Understand* database. The analyzed database cannot be changed by API functions. The C Interface to the database is described in this chapter.

This API can be used with any language-specific version of the *Understand* software. It is not only for use with *Understand for C++*.

A quick overview of using the C API is provided in the next section. Some code samples utilizing basic API functions are also provided.

The API Reference Section contains detailed descriptions on each API function. Some functions are specific to certain languages. *Understand* product-specific functions are noted in their descriptions.

Downloading

The C API has a separate installation. To download it, go to http://www.scitools.com/downloadapi.shtml.

To add the API to your installation, install it to the same location as your *Understand* installation location. It is important to make sure the API and the Understand versions are identical.

Licensing

Use of the C API requires an *Understand* license be available for use. You do not need to do anything extra to request the license—it is handled for you by the API.

API Memory Allocation

The API never requires the user to pass in an allocated item. Any API function that does allocation has a corresponding API function which must be called to free that item. The user must never call the system 'free()' to free an item allocated by the API.

Getting Started with the C API

Using the C API to access the *Understand* database is generally done in the following way:

- Open *Understand* database
- Retrieve List of All Entities
- Filter the List of Entities
- Retrieve Desired Info about each Entity from the Filtered List Entity Info includes:
  - names, types, kinds, etc.
  - metric values
  - reference information
- Close database

API Include file

To use the *Understand* API, you will need to include udb.h in any source files that call an *Understand* API function. Reference the .h file with the appropriate relative path from your project source area. For example:

```
#include "../sti/src/udb/udb.h"
```

Since changes may have been made to the C API since the publication of this manual, you can check the udb.h file for the latest reference information about the C API.

Understand Licensing

In order for your application to use the *Understand* API functions, an *Understand* license must first be found. You have several options in specifying where the API should look to find the required *Understand* license file:

- Call ***udbSetLicense()*** to specify the actual path to your *Understand* license. This is usually unnecessary.
- Set the environment variable $STILICENSE.
- If you do neither of the above, the API uses the license found in the location specified by $STIHOME/conf/license.

Note that the license will not be 'checked out' for use until the *Understand* database is opened. Be sure to always check the return status of ***udbDbOpen()*** as it will fail if all licenses are already in use or if a license is not found.

Opening and Closing the Database

As previously noted, use udbDbOpen() to open the *Understand* database for reading. This action also consumes an *Understand* license. Refer to *udbDbOpen* on page 5–11 for detailed information and return statuses.

```
status = udbDbOpen("test.udc");
```

Then close the database when done:

```
udbDbClose();
```

Get a List of Entities

Once the database is open, retrieve all the entities that are of interest to you.

First get the list of all entities:

```
udbListEntity( &allEnts, &allEntsSize);
```

Then filter the list to include only what you are interested in. In this example, the list is filtered to include only functions:

```
udbListEntityFilter (allEnts,
                     udbKindParse("function"),
                     &funcEnts,
                     &funcEntsSize);
```

When done with the entity lists, be sure to free them:

```
udbListEntityFree(allEnts);
udbListEntityFree(funcEnts);
```

Get Information About an Entity

Once you have an entity (or group of entities), you can obtain all kinds of information about it. The following example prints both the short and long names of each function.

```
for (i=0; i<funcEntsSize; i++)   {
       printf ("Shortname: %s, Longname:%s\n",
              udbEntityNameShort(funcEnts[i]),
              udbEntityNameLong(funcEnts[i]) );
}
```

Get Reference Information for an Entity

In addition to getting general entity and metrics information, you can also get information about the references to or from an entity. Again, you obtain a list of all references for a particular entity, and then filter the list to include only those types of references that you are interested in.

The following example reports where an entity is defined showing the parent entity in which it is defined, as well as the file, line, and column location where the definition occurs.

First, get all the references for the entity, then filter the reference list to include only those where the entity is defined , of which there should only be one occurrence.

```
udbListReference(entity, &refs, NULL);
udbListReferenceFilter(refs,
   udbKindParse("definedin,declaredin"), NULL, 0, &defs,
   NULL);
```

Once you have the desired list of references, you can do whatever you want with them. As always, remember to free the Reference Lists when done.

```
if (defs != NULL && defs[0] != 0)  {
   printf ("     %s (%s at line:%d col:%d)\n\n",

udbEntityNameShort(udbReferenceEntity(defs[0])),

udbEntityNameShort(udbReferenceFile(defs[0])),
           udbReferenceLine(defs[0]),
           udbReferenceColumn(defs[0]) );
   udbListReferenceFree(defs);
 }
 udbListReferenceFree(refs);
```

About Library Support

Several API functions utilize libraries. Currently these functions only apply to *Understand for Ada* and *Understand for Java* databases. In *Understand for Ada*, there will always be a library named “standard” and this is the only Ada library supported at this time. In *Understand for Java* the Java JDK entities are in a library named “standard”.

Any entity that is not in the “standard” library, is considered to be in the NULL library. To filter on entities that are not in the standard library, specify “~standard”. For example:

```
udbLibraryFilterEntity(entities,
                       "~standard",
                       &entities, &size);
```

will return a list of all entities not in the standard library (for Ada only).

Compiling & Linking with the API library

Compile the application as you normally would, adding dependencies for the API header file and library to the makefile.

The API libraries are located in the $STIHOME/bin/<architecture> directory.

On Unix/Linux, link with either udb_api.so or udb_api.sl (for dynamic linking) or udb_api.a (for static linking).

On Windows, link with either udb_api.lib (for dynamic linking) or udb_api.obj (for static linking). Define the system libraries needed. Your application may need additional libraries defined.

The following command performs static linking on Windows:

```
link test.obj udb_api.obj /nodefaultlib:libc advapi32.lib
   user32.lib wsock32.lib netapi32.lib
```

The following command performs dynamic linking on Windows:

```
link test.obj udb_api.lib
```

Code Sample Using the C API

The following simple code sample opens an Understand database, reports a couple of project metrics, a list of function names and a reference for where the function is defined, and then closes the database.

The API calls have been bolded for illustration.

```
/* sample program using Understand C API */
#include <stdio.h>
#include <stdlib.h>
#include "udb.h"

main(
  int argc,
  char *argv[])
{
  UdbEntity *ents; /* list of entities */
  int entsSize; /* number of entities */
  UdbReference *refs; /* list of references for entity */
  UdbReference *filterrefs; /* list of filtered references for entity */
  int i; /* counter */
  UdbStatus status; /* return status */
  /* open Understand database */
  status = udbDbOpen("sample.udc");
  if (status) {
    printf ("unable to open valid Understand database: %s ", "sample.udc");
    switch (status) {
      case Udb_statusDBAlreadyOpen:
        printf("database is already open\n"); break;
      case Udb_statusDBOldVersion:
        printf("database is old version\n"); break;
      case Udb_statusDBUnknownVersion:
        printf("database is unknown version\n"); break;
      case Udb_statusDBUnableOpen:
        printf("unable to locate database\n"); break;
      case Udb_statusNoApiLicenseAda:
        printf("no Understand/Ada license available\n"); break;
      case Udb_statusNoApiLicenseC:
        printf("no Understand/C license available\n"); break;
      case Udb_statusNoApiLicenseFtn:
        printf("no Understand/Fortran license available\n"); break;
      default:
        printf("unable to access database\n"); break;
    }
```

```
    exit (EXIT_FAILURE);
  }
  /* get list of all entities */
  udbListEntity(&ents, &entsSize);
  /* filter list to include only non-member object kinds */
  udbListEntityFilter (ents, udbKindParse("object ~member"),
                       &ents, &entsSize);
  /* loop through all the list of filtered entities */
  for (i=0; i<entsSize; i++) {
    printf ("%s\n", udbEntityNameLong(ents[i]));
    /* get references for this entity */
    udbListReference(ents[i], &refs, NULL);
    /* filter references to desired kinds (in this case, where defined) */
    udbListReferenceFilter(refs, udbKindParse("definedin"), NULL,
                           0, &filterrefs, NULL);
    /* output reference info if reference of specified kind is found */
    if (filterrefs != NULL && filterrefs[0] != 0) {
      printf (" %s: %s in %s at line:%d \n\n",
              udbKindShortname(udbReferenceKind(filterrefs[0])),
              udbEntityNameShort(udbReferenceEntity(filterrefs[0])),
              udbEntityNameShort(udbReferenceFile(filterrefs[0])),
              udbReferenceLine(filterrefs[0]) );
      /* free the filtered ref list, if any */
      udbListReferenceFree(filterrefs);
    }
    /* free the ref list for the entity */
    udbListReferenceFree(refs);
  }
  /* free the entity list */
  udbListEntityFree(ents);
  /* close Understand database */
  udbDbClose();
  return (0);
}
```

Chapter 5 C API Functions

This chapter provides detailed references for each C API function available to access the *Understand* database. It is recommended that you begin by reading through the introductory sections of the preceding chapter before using this chapter.

This chapter contains the following API function specifications:

Section	Page
udbComment	5–5
udbCommentRaw	5–7
udbDbClose	5–8
udbDbLanguage	5–9
udbDbName	5–10
udbDbOpen	5–11
udbEntityDraw	5–13
udbEntityId	5–16
udbEntityKind	5–17
udbEntityLanguage	5–18
udbEntityLibrary	5–19
udbEntityNameLong	5–20
udbEntityNameShort	5–21
udbEntityNameSimple	5–22
udbEntityNameUnique	5–23
udbEntityRefs	5–24
udbEntityTypetext	5–25
udbInfoBuild	5–26
udbIsKind	5–27
udbIsKindFile	5–28
udbKindInverse	5–29
udbKindLanguage	5–30
udbKindList	5–31
udbKindListCopy	5–32
udbKindListFree	5–33
udbKindLocate	5–34
udbKindLongname	5–35

Section	Page
udbKindParse	5–36
udbLexemeColumnBegin	5–38
udbLexemeColumnEnd	5–39
udbLexemeEntity	5–40
udbLexemeInactive	5–41
udbLexemeLineBegin	5–42
udbLexemeLineEnd	5–43
udbLexemeNext	5–44
udbLexemePrevious	5–45
udbLexemeReference	5–46
udbLexemeText	5–47
udbLexemeToken	5–48
udbLexerDelete	5–49
udbLexerFirst	5–50
udbLexerLexeme	5–51
udbLexerLexemes	5–52
udbLexerLines	5–53
udbLexerNew	5–54
udbLibraryCheckEntity	5–55
udbLibraryCompare	5–56
udbLibraryFilterEntity	5–57
udbLibraryList	5–58
udbLibraryListFree	5–59
udbLibraryName	5–60
udbLicenseInfo	5–61
udbListEntity	5–62
udbListEntityFilter	5–63
udbListEntityFree	5–64
udbListFile	5–65
udbListKindEntity	5–66
udbListKindFree	5–67
udbListKindReference	5–68
udbListReference	5–69
udbListReferenceFile	5–70
udbListReferenceFilter	5–71

Section	Page
udbListReferenceFree	5–72
udbLookupEntity	5–73
udbLookupEntityByReference	5–74
udbLookupEntityByUniquename	5–75
udbLookupFile	5–76
udbLookupReferenceExists	5–77
udbMetricDescription	5–78
udbMetricIsDefinedEntity	5–79
udbMetricIsDefinedProject	5–80
udbMetricListEntity	5–82
udbMetricListKind	5–83
udbMetricListLanguage	5–84
udbMetricListProject	5–85
udbMetricLookup	5–86
udbMetricName	5–87
udbMetricValue	5–88
udbMetricValueProject	5–89
udbReferenceColumn	5–90
udbReferenceCopy	5–91
udbReferenceCopyFree	5–92
udbReferenceEntity	5–93
udbReferenceFile	5–94
udbReferenceKind	5–95
udbReferenceLine	5–96
udbReferenceScope	5–97

API Function Reference

This reference section contains API functions for all supported language-specific versions of the *Understand* software. Some functions are specific to certain languages. Language-specific functions will be noted in their descriptions.

It is recommended that you read through the introductory sections of the preceding chapter before using the API.

Chapter 7, "Entity and Reference Kinds" details available entity and reference kinds which are utilized by several API functions. Since changes may have been made to the C API since the publication of this manual, you can check the udb.h file for the latest reference information about the C API.

udbComment

Description

Returns a formatted string based on comments associated with a given entity.

Syntax

```
#include "udb/udb.h"
char * udbComment(UdbEntity entity,
                  UdbCommentStyle style,
                  UdbCommentFormat format,
                  UdbKindList kinds)
```

Arguments

Argument	Description
UdbEntity entity	Specify the entity.
UdbCommentStyle style	Specify the comment style.
UdbCommentFormat format	Specify the comment format.
UdbKindList kinds	If not NULL, entity kinds to filter.

UdbEntity specifies the entity for which comments are to be returned.

UdbCommentStyle specifies which comments are to be used. By default, comments after the entity declaration are processed. The valid values are:

Argument Values	Description
Udb_commentStyleDefault or Udb_commentStyleAfter	Process comments after the entity declaration. (This is the default.)
Udb_commentStyleBefore	Process comments before the entity declaration.

UdbCommentFormat specifies what kind of formatting, if any, is applied to the comment text. The valid value is:

Argument Values	Description
Udb_commentFormatDefault	Removes comment characters and certain repeating characters, while retaining the original newlines. (Use udbCommentRaw to return the raw comment string.)

UdbKindList is a language-specific reference filter string to specify which kinds of entities should be matched.

Return Values

Return Values	Description
char *	Pointer to the string of comments returned.

See Also *udbCommentRaw* on page 5–7

udbCommentRaw

Description

Returns a raw comment string based on comments associated with a given entity.

Syntax

```
#include "udb/udb.h"
void udbCommentRaw(UdbEntity entity,
                   UdbCommentStyle style,
                   UdbKindList kinds,
                   char ***commentString,
                   int *)
```

Arguments

Argument	Description
UdbEntity entity	Specify the entity
UdbCommentStyle style	Specify the comment style.
UdbKindList kinds	If not NULL, entity kinds to filter.
char ***commentString	Pointer to the raw comment string returned.
int *len	Returns the number of comments.

UdbEntity specifies the entity for which comments are to be returned.

UdbCommentStyle specifies which comments are to be used. By default, comments after the entity declaration are processed. The valid values are:

Argument Values	Description
Udb_commentStyleDefault or Udb_commentStyleAfter	Process comments after the entity declaration. (This is the default.)
Udb_commentStyleBefore	Process comments before the entity declaration.

UdbKindList is a language-specific reference filter string to specify which kinds of entities should be matched.

Return Values

There is no return value.

See Also

udbComment on page 5–5

udbDbClose

Description	Close the current *Understand* database. Only one database may be open at a time.
Syntax	`#include "udb/udb.h"` `void udbDbClose(void)`
Arguments	There are no arguments to *udbDbClose*.
Return Values	There are no return values from *udbDbClose*.
Example Usage	`udbDbClose();`
See Also	*udbDbOpen* on page 5–11

udbDbLanguage

Description

Return the general language variety of the current (open) Understand database.

Only one Understand database can be open at any time.

Syntax

```
#include "udb/udb.h"
UdbLanguage udbDbLanguage(void)
```

Arguments

There are no arguments to *udbDbLanguage*.

Return Values

UdbDbLanguage returns one of the following values specifying the language of the Understand database.

Return Values	Description
Udb_language_NONE	No language specified for database.
Udb_language_ALL	Project supports all languages.
Udb_language_Ada	Project database conforms to Ada.
Udb_language_C	Project database conforms to C or C++.
Udb_language_Fortran	Project database conforms to FORTRAN.
Udb_language_Java	Project database conforms to Java.
Udb_language_Jovial	Project database conforms to JOVIAL.
Udb_language_Pascal	Project database conforms to Pascal.

See Also

udbEntityLanguage on page 5–18

udbKindLanguage on page 5–30

udbMetricIsDefinedProject on page 5–80

udbMetricListLanguage on page 5–84

udbMetricListProject on page 5–85

udbDbName

Description Return the (non-allocated) filename of the current *Understand* database. Returns NULL if no database is currently open.

Syntax

```
#include "udb/udb.h"
char *udbDbName(void)
```

Arguments There are no arguments for *udbDbName*.

Return Values

Return Values	Description
char *	Returns a non-allocated string of the filename of the *Understand* database.
NULL	NULL is returned when there is not a currently open database.

Example Usage

```
printf ("Database name: %s\n", udbDbName());
```

See Also *udbDbOpen* on page 5–11

udbDbOpen

Description

Open the specified *Understand* database.

The specified database file is located, opened, and the license checked. If the specified file cannot be found, Udb_statusDBUnableOpen is returned. A valid *Understand* license is required in order for a database to be successfully opened. If a license is not found or not available for use, the language specific value Udb_statusNoApiLicense[C|Ada|Ftn|Java|Jovial|Pascal] is returned. Only read access to the database is allowed from the API.

Some versions of *Understand* will automatically update an older database version to the current database version. This API function will not perform that database version upgrade. Use *Understand* to upgrade or re-analyze your project database.

Only one database may be open at a time.

Syntax

```
#include "udb/udb.h"
UdbStatus  udbDbOpen(char *filename);
```

Arguments

Argument	Description
char *filename	Specify the filename of the *Understand* database to open. Non-allocated.

Return Values

Return Values	Description
Udb_statusOkay	Database opened successfully
Udb_statusDBAlreadyOpen	Database is already open
Udb_statusDBChanged	Database has been changed
Udb_statusDBCorrupt	Database is corrupt
Udb_statusDBOldVersion	Database has old version
Udb_statusDBUnknownVersion	Database has unknown version
Udb_statusDBUnableCreate	Unable to create database
Udb_statusDBUnableDelete	Unable to delete database
Udb_statusDBUnableModify	Unable to modify database
Udb_statusDBUnableOpen	Unable to locate the database file

Return Values	Description
Udb_statusDBUnableWrite	Unable to write database
Udb_statusNoApiLicense	No license available for the API.
Udb_statusNoApiLicenseAda	No *Understand for Ada* license available for the API.
Udb_statusNoApiLicenseC	No *Understand for C* license available for the API.
Udb_statusNoApiLicenseFtn	No *Understand for FORTRAN* license available for the API.
Udb_statusNoApiLicenseJava	No *Understand for Java* license available for the API.
Udb_statusNoApiLicenseJovial	No *Understand for JOVIAL* license available for the API.
Udb_statusNoApiLicensePascal	No *Understand for Pascal* license available for the API.

Example Usage

```
status = udbDbOpen("test.udc")
```

See Also

udbDbClose on page 5–8

udbEntityDraw

Description

Create a graphic view of the specified entity in a file.

The kind of graphic views available are dependent on the language and the kind of entity specified. For example, a function entity may have view kinds of “Invocation”, “Callby”, “Declaration” and “Declaration File”.

Graphical view options may optionally be specified as a name and value pair. These options are also dependent on the language and the kind of view specified. For example, the Callby view of a function could optionally show parameters as well as set the font size used, among others.

Syntax

```
#include "udb/udb.h"
UdbStatus udbEntityDraw(char *view,
                        UdbEntity entity,
                        char *options,
                         char *file)
```

Arguments

Argument	Description
char * view	Specify the kind of graphic view to draw. Non-allocated. The kind of views available depend on the language and the kind of entity specified. A list of the available view kinds for each language follows.
UdbEntity entity	Specify the entity to draw.
char * options	Specify NULL or a string of drawing options. Non-allocated. The format of the options string is "name=value". Multiple options are separated with a semicolon. The valid names and values are the same as those that appear in the Understand Options menus. They may be abbreviated to any unique prefix of their full names. A the list of possible option names and their values for each language follows. Additionally, the special option 'font' may be specified. Its value must be the path to a TrueType font (.ttf).
char *file	Specify the output file name (.png or .jpg only). Non-allocated.

The View Kinds are the same as those supported in Understand. As of the print date, these are the options:

Languages	Values
Ada	Callby, Child Lib Units, Declaration, Declaration Tree, Declared In, Instantiated From, Instantiations, Invocation, Parent Declaration, Parent Lib Unit, Rename Declaration, Type Derived From, Type Tree, With, Withby
C/C++	Base Classes, Callby, Data Members, Declaration, Declaration File, Declaration Type, Derived Classes, Include, Includeby, Invocation, Parent Declaration, Return Type
FORTRAN	Callby, Declaration, Declaration File, Include, Includeby, Invocation, Usedby, Uses
Java	Callby, Contains, Declaration, Declaration File, Declaration Type, Extends, Extended By, Invocation, Parent Declaration, Return Type
JOVIAL	Callby, Declaration, Declaration File, Declaration Type, Invocation, Parent Declaration, Return Type
Pascal	Callby, Declaration, Declaration File, Include, Includeby, Invocation

The drawing options may be used to specify parameters used to generate the graphics. The format of the options string is "name=value". Multiple options are separated with a semicolon.

Not all options are available for all programming languages and all graphical view types. For a list of the valid options, open *Understand* for your programming language, then choose **Options->Graphical Settings** and go to the **View Options** tab. When you select a view, the options for that graph type are shown.

An options string may contain options that are specific for different views, using the syntax "[viewname]optionname=value". This permits you to, for instance, turn fullnames on for Invocation Trees, but not for CallBy trees.

Names and values may be abbreviated to any unique prefix of their full names. For example, you could use the following options string:

```
"layout=cross; scale=14"
```

Additionally, you may use “font” as a special option for any graph type. Its value must be the path to a TrueType font (.ttf) for JPG and PNG files.

```
"font=c:\winnt\fonts\times.ttf"
```

Return Values

Return Values	Description
Udb_statusOkay	Draw successful.
Udb_statusDrawNoFont	No suitable font can be found.
Udb_statusDrawNoImage	No valid image view is defined or the specified entity does not have a view of the specified kind.
Udb_statusDrawTooBig	The selected view is too large. Note the jpg format does not support a dimension larger than 64K
Udb_statusDrawUnableCreateFile	Unable to create or open file.
Udb_statusDrawUnsupportedFile	File type requested is not supported. Specify only .png or .jpg.

Example Usage

```
status = udbEntityDraw ("Callby",
                        funtionEntity,
                        "scale=14; level=1",
                         "callby.png");
```

See Also *udbEntityKind* on page 5–17

udbEntityId

Description

Return a unique id for the specified entity. The intent of this function is to provide a perfect hash value for entities.

Ids are unique for each entity and are valid only for the currently open database in which they are retrieved. Closing and re-opening the database or reparsing of the database (either partially or in full) invalidates any entity ids obtained previously.

Syntax

```
#include "udb/udb.h"
int  udbEntityId(UdbEntity entity)
```

Arguments

Argument	Description
UdbEntity entity	Specify the entity

Return Values

Return Values	Description
int	The unique id for this entity as assigned at database opening. Not valid between subsequent opens of the database or for different analysis runs.

Example Usage

```
printf("name of entity %4d is %s\n",
              udbEntityId(entity),
               udbEntityNameShort(entity) );
```

See Also

udbEntityNameLong on page 5–20

udbEntityNameShort on page 5–21

udbEntityNameSimple on page 5–22

udbEntityKind

Description

Returns the kind of the specified entity.

There are numerous different defined kinds of entities for each language. Refer to the language-specific listings of entity kinds in Chapter 7, “Entity and Reference Kinds”

Syntax

```
#include "udb/udb.h"
UdbKind   udbEntityKind(UdbEntity entity)
```

Arguments

Argument	Description
UdbEntity entity	Specify the entity

Return Values

Return Values	Description
UdbKind	Refer to the language-specific listings of entity kinds for each language in Chapter 7, “Entity and Reference Kinds”.

Example Usage

```
printf ("entity %s is of kind %s\n",
            udbEntityNameShort(entity),
            udbKindShortname(udbEntityKind(entity)) );
```

See Also

udbIsKind on page 5–27

udbIsKindFile on page 5–28

udbKindInverse on page 5–29

udbKindLanguage on page 5–30

udbKindList on page 5–31

udbKindLocate on page 5–34

udbKindLongname on page 5–35

udbKindParse on page 5–36

udbKindShortname on page 5–37

udbEntityLanguage

Description Returns the general language of the specified entity, which is based on the entity kind.

Syntax

```
#include "udb/udb.h"
UdbLanguage  udbEntityLanguage(UdbEntity entity)
```

Arguments

Argument	Description
UdbEntity entity	Specify the entity. The entity kind is examined to determine the language.

Return Values UdbEntityLanguage returns one of the following values specifying the general language of the specified entity, based on the entity kind.

Return Values	Description
Udb_language_Ada	Conforms to Ada.
Udb_language_C	Conforms to C or C++.
Udb_language_Fortran	Conforms to FORTRAN.
Udb_language_Java	Conforms to Java.
Udb_language_Jovial	Conforms to JOVIAL.
Udb_language_Pascal	Conforms to Pascal.

Example Usage

```
language = udbEntityLanguage(entity);
switch (language) {
   case Udb_language_C : ...
   case Udb_language_Fortran : ...
   case Udb_language_Ada : ...
   }
```

See Also

udbDbLanguage on page 5–9

udbKindLanguage on page 5–30

udbMetricIsDefinedProject on page 5–80

udbMetricListLanguage on page 5–84

udbMetricListProject on page 5–85

udbEntityLibrary

Description

Return the library that an entity belongs to.

Applies only to *Understand for Ada* and *Understand for Java* databases. For Ada, there will always be a library named “standard” which is the only Ada library supported at this time. For Java, the Java JDK entities are in a library named “standard”.

Syntax

```
#include "udb/udb.h"
UdbLibrary   udbEntityLibrary(UdbEntity entity);
```

Arguments

Argument	Description
UdbEntity entity	Specify the entity

Return Values

Return Values	Description
UdbLibrary	The library to which this entity belongs or NULL if the entity does not belong to any library.

Example Usage

```
library = udbEntityLibrary(entity);
if (library) {
    printf ("%s is in library %s\n",
                  udbEntityNameLong(entity),
                  udbLibraryName(library) );
}
```

See Also

udbLibraryCheckEntity on page 5–55

udbLibraryCompare on page 5–56

udbLibraryFilterEntity on page 5–57

udbLibraryList on page 5–58

udbLibraryListFree on page 5–59

udbLibraryName on page 5–60

udbEntityNameLong

Description

Return the long name of the specified entity.

If there is no long name defined for this entity, then the short name is returned.

Examples of entities with long names include files, C++ members and most Ada entities. Long names include the name of the compilation unit for entity and function names (Ada), the class name for class members (C++), and the full file path for file names.

Syntax

```
#include "udb/udb.h"
char   *udbEntityNameLong(UdbEntity entity)
```

Arguments

Argument	Description
UdbEntity entity	Specify the entity

Return Values

Return Values	Description
char *	The long name of the entity. This is permanent, non-allocated.

Example Usage

```
printf ("Fullname: %s, Shortname: %s\n",
          udbEntityNameLong(entity),
          udbEntityNameShort(entity) );
```

See Also

udbEntityId on page 5–16

udbEntityNameShort on page 5–21

udbEntityNameSimple on page 5–22

udbEntityNameShort

Description

Return the short name of the specified entity.

The short name does not include the compilation unit (Ada), member's class names (C++) or the full file path of filenames. For Java, the short name includes one class name level (for example, a.b).

Use *udbEntityNameLong()* to obtain the longname (also known as fullname) of an entity.

Syntax

```
#include "udb/udb.h"
char  *udbEntityNameShort(UdbEntity entity)
```

Arguments

Argument	Description
UdbEntity entity	Specify the entity

Return Values

Return Values	Description
char *	The short name of the entity. This is permanent, non-allocated.

Example Usage

```
printf ("Fullname: %s, Shortname: %s\n",
             udbEntityNameLong(entity),
             udbEntityNameShort(entity) );
```

See Also

udbEntityId on page 5–16

udbEntityNameLong on page 5–20

udbEntityNameSimple on page 5–22

udbEntityNameSimple

Description

Returns the simple name of the specified entity.

The simple name does not include anything other than the name of the entity. No prefixes concerning the parent or container of the entity are included.

Syntax

```
#include "udb/udb.h"
char * udbEntityNameSimple(UdbEntity entity)
```

Arguments

Argument	Description
UdbEntity entity	Specify the entity

Return Values

Return Value	Description
char *	The simple name of the entity.

Example Usage

```
printf ("Fullname: %s, Simplename: %s\n",
         udbEntityNameLong(entity),
         udbEntityNameSimple(entity) );
```

See Also

udbEntityId on page 5–16

udbEntityNameLong on page 5–20

udbEntityNameShort on page 5–21

udbEntityNameUnique

Description

Returns a unique name for the specified entity.

The unique name is a temporary, unallocated string that can be used with udbLookupEntityByUniquename.

The uniquename is fully specified, so that it may be looked up again later. The uniquename for an entity is useful when an entity needs to be restored after a database is closed and reopened, for example.

Syntax

```
#include "udb/udb.h"
char * udbEntityNameUnique(UdbEntity entity)
```

Arguments

Argument	Description
UdbEntity entity	Specify the entity

Return Values

Return Value	Description
char *	A temporary unique name for the entity.

Example Usage

```
uniquename = udbEntityNameUnique(entity));
...
refentity = udbLookupEntityByUniquename(uniquename);
```

See Also

udbEntityId on page 5–16

udbLookupEntityByUniquename on page 5–75

udbEntityRefs

Description

Returns a temporary list of references for the specified entity. This is a convenience function that combines several common tasks. By using this function, you do not need to create lists of type UdbKindList.

Syntax

```
#include "udb/udb.h"
int  udbEntityRefs(UdbEntity entity,
                   char *refKindString,
                   char *entKindString,
                   int unique,
                   UdbReference **refs)
```

Arguments

Argument	Description
UdbEntity entity	Specify the entity
char *refKinds	Pass NULL or an unallocated reference kind string.
char *entKinds	Pass NULL or an unallocated entity kind string.
int unique	Set to 1 to return only the first matching reference to each unique entity. Set to 0 otherwise.
UdbReference **refs	Location to return list of references.

Return Values

Return Values	Description
int	Returns the length of the list of references.

See Also

udbLexemeReference on page 5–46

udbListReference on page 5–69

udbLookupEntityByReference on page 5–74

udbLookupReferenceExists on page 5–77

udbReferenceEntity on page 5–93

udbEntityTypetext

Description

Return the type text associated with the specified entity or NULL if there is no type for the entity.

An example of a typetext is “unsigned long int” or “char *”. For a function entity, this is the return type. Not all entities have a type, therefore not all entities have a type text. Files, for example, are not language types, and therefore a file entity will return NULL when *udbEntityTypetext* is queried.

Syntax

```
#include "udb/udb.h"
char  *udbEntityTypeText(UdbEntity entity)
```

Arguments

Argument	Description
UdbEntity entity	Specify the entity

Return Values

Return Values	Description
char *	The type text of the entity. Non-allocated.

Example Usage

```
printf ("%s is of type  %s\n",
           udbEntityNameShort(entity),
           udbEntityTypetext(entity) );
```

See Also

udbEntityKind on page 5–17

udbEntityNameLong on page 5–20

udbEntityNameShort on page 5–21

udbInfoBuild

Description

Returns the build number for the current installed C API. This can be used to make sure the program is being run on the minimum build necessary, or to change program behavior to support limited functionality in older builds.

Syntax

```
#include "udb/udb.h"
char * udbInfoBuild()
```

Arguments

There are no arguments.

Return Values

Return Values	Description
char *	A string describing the current build.

See Also

udbDbName on page 5–10

udbLicenseInfo on page 5–61

udbIsKind

Description

Return true if the specified kind matches the kind name text.

In order to match the criteria of a list of kind names, a kind must have in its fullname, every name listed in the list of names, and must not have in its fullname, any names listed in the list of names that begin with '~'. An explanation of kind names with examples is provided in *Kind Name Filters* on page 7–4.

Refer to the language-specific listings of entity and reference kinds in Chapter 7, “Entity and Reference Kinds”.

Syntax

```
#include "udb/udb.h"
int udbIsKind( UdbKind kind, char *text)
```

Arguments

Argument	Description
UdbKind kind	Specify the entity or reference kind
char *text	Specify the kind text as a static string

Return Values

Return Values	Description
int	Returns true if the specified kind matches the kind text; false otherwise.

Example Usage

udbIsKind can be used with entity kinds:

```
if (udbIsKind(udbEntityKind(entity),
  "c typedef ~member ~unresolved, c object ~member"))...
```

or reference kinds:

```
if (udbIsKind(udbReferenceKind(ref), "call ~inactive")...
```

See Also

udbEntityKind on page 5–17
udbIsKindFile on page 5–28
udbKindInverse on page 5–29
udbKindList on page 5–31
udbKindLocate on page 5–34
udbKindLongname on page 5–35
udbKindParse on page 5–36
udbKindShortname on page 5–37

udbIsKindFile

Description

Return true if the specified kind refers to a “file” kind entity.

Refer to the language-specific listings of entity kinds in Chapter 7, “Entity and Reference Kinds”.

Syntax

```
#include "udb/udb.h"
int udbIsKindFile( UdbKind kind)
```

Arguments

Argument	Description
UdbKind kind	Specify the entity kind

Return Values

Return Values	Description
int	Returns true if the specified kind is a kind of file; false otherwise.

Example Usage

```
if (udbIsKindFile(udbEntityKind(entity))
{ ... }
```

See Also

udbListFile on page 5–65

udbListReferenceFile on page 5–70

udbLookupFile on page 5–76

udbReferenceFile on page 5–94

udbEntityKind on page 5–17

udbIsKind on page 5–27

udbKindInverse on page 5–29

udbKindLanguage on page 5–30

udbKindList on page 5–31

udbKindLocate on page 5–34

udbKindLongname on page 5–35

udbKindParse on page 5–36

udbKindShortname on page 5–37

udbKindInverse

Description

Return the inverse kind of the specified kind. This is valid only for reference kinds. Returns 0 if an invalid kind is specified. For example, the inverse kind of call is callby, and the inverse of setby is set.

Refer to the language-specific listings of reference kinds in Chapter 7, “Entity and Reference Kinds”.

Syntax

```
#include "udb/udb.h"
UdbKind  udbKindInverse( UdbKind  kind)
```

Arguments

Argument	Description
UdbKind kind	Specify the entity kind

Return Values

Return Value	Description
UdbKind	Returns the inverse kind or 0 if an invalid kind is specified.

Example Usage

```
inv_kind =  udbKindInverse(udbReferenceKind(ref));
```

See Also

udbEntityKind on page 5–17

udbIsKind on page 5–27

udbIsKindFile on page 5–28

udbKindLanguage on page 5–30

udbKindList on page 5–31

udbKindLocate on page 5–34

udbKindLongname on page 5–35

udbKindParse on page 5–36

udbKindShortname on page 5–37

udbKindLanguage

Description

Returns the general language of the specified entity or reference kind. Refer to the language-specific listings of reference kinds in Chapter 7, “Entity and Reference Kinds”.

Syntax

```
#include "udb/udb.h"
UdbLanguage  udbKindLanguage( UdbKind  kind)
```

Arguments

Argument	Description
UdbKind kind	Specify the entity or reference kind.

Return Values

Return Value	Description
UdbLanguage	Language of the specified kind.

UdbLanguage specifies a programming language supported by *Understand* databases. Valid values are:

Return Values	Description
Udb_language_Ada	Project database conforms to Ada.
Udb_language_C	Project database conforms to C or C++.
Udb_language_Fortran	Project database conforms to FORTRAN.
Udb_language_Java	Project database conforms to Java.
Udb_language_Jovial	Project database conforms to JOVIAL.
Udb_language_Pascal	Project database conforms to Pascal.

Example Usage

```
language = udbKindLanguage(udbReferenceKind(ref));
```

See Also

udbEntityKind on page 5–17
udbIsKind on page 5–27
udbKindList on page 5–31
udbKindLocate on page 5–34
udbKindParse on page 5–36
udbDbLanguage on page 5–9
udbEntityLanguage on page 5–18
udbMetricIsDefinedProject on page 5–80
udbMetricListLanguage on page 5–84
udbMetricListProject on page 5–85

udbKindList

Description

Add a kind to the specified kindlist and return the reallocated kindlist.

Use *udbKindListFree()* to free the kindlist when done.

In order to match the criteria of a list of kind names, a kind must have in its fullname, every name listed in the list of names, and must not have in its fullname, any names listed in the list of names that begin with '~'. An explanation of kind names with examples is provided in *Kind Name Filters* on page 7–4

Refer to the language-specific listings of entity and reference kinds in Chapter 7, “Entity and Reference Kinds”.

Syntax

```
#include "udb/udb.h"
void    udbKindList(UdbKind kind, UdbKindList *kindlist)
```

Arguments

Argument	Description
UdbKind kind	Specify the kind
UdbKindList *kindlist	*kindlist must be an existing kind list to add to or NULL to create a new list.

Return Values

There is no function return value.

Example Usage

```
udbKindList( kind, &kindlist);
```

or

```
udbKindList(udbEntityKind (entity), &kindlist );
```

The resulting kindlist can then be passed to other API functions:

```
udbListEntityFilter(ents, kindlist, &newents, NULL);
```

See Also

udbKindListCopy on page 5–32

udbKindListFree on page 5–33

udbKindLocate on page 5–34

udbKindParse on page 5–36

udbKindListCopy

Description

Return an allocated copy of the specified kindlist.

Use *udbKindListFree()* to free the kindlist when done.

Refer to the language-specific listings of entity and reference kinds in Chapter 7, “Entity and Reference Kinds”.

Syntax

```
#include "udb/udb.h"
UdbKindList  udbKindListCopy(UdbKindList);
```

Arguments

Argument	Description
UdbKindList *kindlist	The kindlist to copy

Return Values

Return Value	Description
UdbKindList *	Allocated copy of the specified kindlist

Example Usage

```
newkindlist = udbKindListCopy(kindlist);
```

See Also

udbKindList on page 5–31

udbKindListFree on page 5–33

udbKindListFree

Description

Free the specified kindlist.

Kindlists created or copied by *udbKindList()* or *udbKindListCopy()* must be freed with *udbKindListFree()*.

Syntax

```
#include "udb/udb.h"
void   udbKindListFree(UdbKindList kindlist);
```

Arguments

Argument	Description
UdbKindList kindlist	The allocated kindlist

Return Values

There is no function return value.

Example Usage

```
udbKindListFree( kindlist);
```

See Also

udbKindList on page 5–31

udbKindListCopy on page 5–32

udbKindLocate

Description

Returns true if the specified kind is in the kind list, or if the list is NULL.

Refer to the language-specific listings of entity and reference kinds in Chapter 7, “Entity and Reference Kinds”.

Syntax

```
#include "udb/udb.h"
int  udbKindLocate(UdbKind kind, UdbKindList kindlist);
```

Arguments

Argument	Description
UdbKind kind	The kind to find in the kindlist
UdbKindList *kindlist	The kindlist

Return Values

Return Value	Description
int	Return true if the specified kind is in the kindlist or if the kindlist is NULL; false otherwise.

Example Usage

```
if ( udbKindLocate(kind, kinds) ) {...}
```

See Also

udbComment on page 5–5

udbCommentRaw on page 5–7

udbEntityKind on page 5–17

udbKindList on page 5–31

udbKindListCopy on page 5–32

udbKindListFree on page 5–33

udbKindParse on page 5–36

udbListEntityFilter on page 5–63

udbListReferenceFilter on page 5–71

udbLookupReferenceExists on page 5–77

udbReferenceKind on page 5–95

udbKindLongname

Description

Return the longname associated with the specified kind. Return NULL if an invalid kind is specified. The returned name is Static, Non-Allocated.

The longname is the fullname of the kind. The shortname is an informal name for the kind.

Syntax

```
#include "udb/udb.h"
char   *udbKindLongname(UdbKind kind);
```

Arguments

Argument	Description
UdbKind kind	Specify the kind

Return Values

Return Value	Description
char *	The longname associated with the specified kind. The returned name is Static, Non-Allocated.

Example Usage

```
printf(" (%s)\n", udbKindLongname(
udbEntityKind(entity)) );
```

See Also

udbEntityKind on page 5–17

udbKindShortname on page 5–37

udbReferenceKind on page 5–95

udbKindParse

Description

Parse the specified text containing one or more comma separated groups of one or more kind names. Return an allocated kindlist of all kinds that match the specified text. Returns NULL if no kinds match.

A kind must match the criteria specified within any comma separated list of kind names. In order to match the criteria of a list of kind names, a kind must have in its fullname, every name listed in the list of names, and must not have in its fullname, any names listed in the list of names that begin with '~'. An explanation of kind names with examples is provided in *Kind Name Filters* on page 7–4

Refer to the language-specific listings of entity and reference kinds in Chapter 7, “Entity and Reference Kinds”.

Syntax

```
#include "udb/udb.h"
UdbKindList  udbKindParse(char *text)
```

Arguments

Argument	Description
char *text	Specify one or more comma separated groups of one or more kind names.

Return Values

Return Value	Description
UdbKindList	Return an allocated kindlist of all kinds that match the specified text.

Example Usage

```
kinds = udbKindParse("c global object ~static");
```

See Also

udbKindList on page 5–31

udbKindListCopy on page 5–32

udbKindListFree on page 5–33

udbKindLocate on page 5–34

udbListEntityFilter on page 5–63

udbListReferenceFilter on page 5–71

udbLookupReferenceExists on page 5–77

udbKindShortname

Description

Return the shortname associated with the specified kind. Return NULL if an invalid kind is specified. The returned name is Static, Non-Allocated.

The shortname is an informal name for the kind. In contrast, the longname is the fullname of the kind.

Syntax

```
#include "udb/udb.h"
char *udbKindShortname(UdbKind)
```

Arguments

Argument	Description
UdbKind kind	Specify the kind

Return Values

Return Value	Description
char *	The shortname associated with the specified kind. The returned name is Static, Non-Allocated.

Example Usage

```
printf("%s\n",
udbKindShortname(udbEntityKind(entity)));
```

See Also

udbEntityKind on page 5–17

udbKindLongname on page 5–35

udbReferenceKind on page 5–95

udbLexemeColumnBegin

Description Returns the beginning column of the lexeme.

Syntax

```
#include "udb/udb.h"
int udbLexemeColumnBegin(UdbLexeme lexeme)
```

Arguments

Argument	Description
UdbLexeme lexeme	Specify the lexeme.

Return Values

Return Value	Description
int	The column of the source code file in which the specified lexeme begins.

Example Usage

```
colstart = udbLexemeColumnBegin(udbLexemeNext(lexeme));
```

See Also

udbLexemeColumnEnd on page 5–39

udbLexemeLineBegin on page 5–42

udbLexemeLineEnd on page 5–43

udbLexemeColumnEnd

Description Returns the ending column of the lexeme.

Syntax

```
#include "udb/udb.h"
int udbLexemeColumnEnd(UdbLexeme lexeme)
```

Arguments

Argument	Description
UdbLexeme lexeme	Specify the lexeme.

Return Values

Return Value	Description
int	The column of the source code file at which the specified lexeme ends.

Example Usage

```
colend = udbLexemeColumnEnd(udbLexemeNext(lexeme));
```

See Also

udbLexemeColumnBegin on page 5–38

udbLexemeLineBegin on page 5–42

udbLexemeLineEnd on page 5–43

udbLexemeEntity

Description

Returns the entity associated with the lexeme. Returns NULL if there is no associated entity.

Syntax

```
#include "udb/udb.h"
UdbEntity udbLexemeEntity(UdbLexeme lexeme)
```

Arguments

Argument	Description
UdbLexeme lexeme	Specify the lexeme.

Return Values

Return Value	Description
UdbEntity	The entity associated with the specified lexeme.

Example Usage

```
entity = udbLexemeEntity(udbLexemeNext(lexeme));
```

See Also

udbLexemeInactive on page 5–41

udbLexemeReference on page 5–46

udbLexemeText on page 5–47

udbLexemeToken on page 5–48

udbLexemeInactive

Description Returns true if the lexeme is part of inactive code.

Syntax

```
#include "udb/udb.h"
int udbLexemeInactive(UdbLexeme lexeme)
```

Arguments

Argument	Description
UdbLexeme lexeme	Specify the lexeme.

Return Values

Return Value	Description
int	Returns true if the specified lexeme is part of inactive code; false otherwise.

Example Usage

```
inactive = udbLexemeInactive(udbLexemeNext(lexeme));
```

See Also

udbLexemeEntity on page 5–40

udbLexemeReference on page 5–46

udbLexemeText on page 5–47

udbLexemeToken on page 5–48

udbLexemeLineBegin

Description Returns the beginning line of the lexeme.

Syntax

```
#include "udb/udb.h"
int udbLexemeLineBegin(UdbLexeme lexeme)
```

Arguments

Argument	Description
UdbLexeme lexeme	Specify the lexeme.

Return Values

Return Value	Description
int	The line of the source code file in which the specified lexeme begins.

Example Usage

```
linestart = udbLexemeLineBegin(udbLexemeNext(lexeme));
```

See Also

udbLexemeColumnBegin on page 5–38

udbLexemeColumnEnd on page 5–39

udbLexemeLineEnd on page 5–43

udbLexerLines on page 5–53

udbLexemeLineEnd

Description Returns the ending line of the lexeme.

Syntax

```
#include "udb/udb.h"
int udbLexemeLineEnd(UdbLexeme lexeme)
```

Arguments

Argument	Description
UdbLexeme lexeme	Specify the lexeme.

Return Values

Return Value	Description
int	The line of the source code file at which the specified lexeme ends.

Example Usage

```
lineend = udbLexemeLineEnd(udbLexemeNext(lexeme));
```

See Also

udbLexemeColumnBegin on page 5–38

udbLexemeColumnEnd on page 5–39

udbLexemeLineBegin on page 5–42

udbLexerLines on page 5–53

udbLexemeNext

Description

Returns the next lexeme. Returns NULL if there is no next lexeme.

Syntax

```
#include "udb/udb.h"
int udbLexemeNext(UdbLexeme lexeme)
```

Arguments

Argument	Description
UdbLexeme lexeme	Specify the lexeme.

Return Values

Return Value	Description
UdbLexeme	The next lexeme after the specified lexeme.

Example Usage

```
nextLexeme = udbLexemeNext(lexeme);
```

See Also

udbLexemePrevious on page 5–45

udbLexerFirst on page 5–50

udbLexerLexeme on page 5–51

udbLexerLexemes on page 5–52

udbLexemePrevious

Description Returns the next lexeme. Returns NULL if there is no next lexeme.

Syntax

```
#include "udb/udb.h"
int udbLexemePrevious(UdbLexeme lexeme)
```

Arguments

Argument	Description
UdbLexeme lexeme	Specify the lexeme.

Return Values

Return Value	Description
UdbLexeme	The lexeme before the specified lexeme.

Example Usage

```
prevLexeme = udbLexemePrevious(lexeme);
```

See Also

udbLexemeNext on page 5–44

udbLexerFirst on page 5–50

udbLexerLexeme on page 5–51

udbLexerLexemes on page 5–52

udbLexemeReference

Description Returns the reference associated with the lexeme. Returns NULL if there is no associated reference.

Syntax

```
#include "udb/udb.h"
UdbReference udbLexemeReference(UdbLexeme lexeme)
```

Arguments

Argument	Description
UdbLexeme lexeme	Specify the lexeme.

Return Values

Return Value	Description
UdbReference	The reference associated with the specified lexeme.

Example Usage

```
entity = udbLexemeReference(udbLexemeNext(lexeme));
```

See Also

udbEntityRefs on page 5–24

udbListReference on page 5–69

udbLookupEntityByReference on page 5–74

udbLookupReferenceExists on page 5–77

udbReferenceEntity on page 5–93

udbLexemeText

Description Returns the reference associated with the lexeme. Returns NULL if there is no associated reference.

Syntax

```
#include "udb/udb.h"
char * udbLexemeText(UdbLexeme lexeme)
```

Arguments

Argument	Description
UdbLexeme lexeme	Specify the lexeme.

Return Values

Return Value	Description
char *	The text of the specified lexeme.

Example Usage

```
printf ("Line %i: %s\n", udbLexemeLineBegin(lexeme),
                         udbLexemeText(lexeme) );
```

See Also

udbLexemeEntity on page 5–40

udbLexemeInactive on page 5–41

udbLexemeReference on page 5–46

udbLexemeToken on page 5–48

udbLexemeToken

Description

Returns the token kind of the lexeme.

Syntax

```
#include "udb/udb.h"
UdbToken udbLexemeToken(UdbLexeme lexeme)
```

Arguments

Argument	Description
UdbLexeme lexeme	Specify the lexeme.

Return Values

Return Value	Description
UdbToken	The reference associated with the specified lexeme.

UdbToken indicates the type of lexeme. Valid values are:

Return Values	Description
Udb_tokenEOF	Lexeme is an End of File marker or does not exist.
Udb_tokenComment	Lexeme is a comment.
Udb_tokenIdentifier	Lexeme is an identifier.
Udb_tokenKeyword	Lexeme is a programming language keyword.
Udb_tokenLiteral	Lexeme is a literal.
Udb_tokenNewline	Lexeme is a newline.
Udb_tokenOperator	Lexeme is an operator.
Udb_tokenPreprocessor	Lexeme is a preprocessor command.
Udb_tokenPunctuation	Lexeme is a form of punctuation.
Udb_tokenString	Lexeme is a string.
Udb_tokenWhitespace	Lexeme is whitespace.

Example Usage

```
lexType = udbLexemeToken(udbLexemeNext(lexeme));
```

See Also

udbLexemeEntity on page 5–40

udbLexemeInactive on page 5–41

udbLexemeReference on page 5–46

udbLexemeText on page 5–47

udbLexerDelete

Description Deletes and frees the specified lexer.

Syntax

```
#include "udb/udb.h"
void udbLexerDelete(UdbLexer lexer)
```

Arguments

Argument	Description
UdbLexer lexer	Specify the lexer.

Return Values There is no return value.

Example Usage

```
udbLexerDelete(lexer);
```

See Also *udbLexerNew* on page 5–54

udbLexerFirst

Description

Returns the first lexeme in the specified lexer.

Syntax

```
#include "udb/udb.h"
UdbLexeme udbLexerFirst(UdbLexer lexer)
```

Arguments

Argument	Description
UdbLexer lexer	Specify the lexer.

Return Values

Return Value	Description
UdbLexeme	The first lexeme in the specified lexer.

Example Usage

```
lexeme = udbLexerFirst(lexer);
```

See Also

udbLexemeNext on page 5–44

udbLexemePrevious on page 5–45

udbLexerLexeme on page 5–51

udbLexerLexemes on page 5–52

udbLexerLexeme

Description Returns the lexeme that occurs at the specified line and column in the lexer.

Syntax

```
#include "udb/udb.h"
UdbLexeme udbLexerLexeme(UdbLexer lexer,
                         int line, int col)
```

Arguments

Argument	Description
UdbLexer lexer	Specify the lexer.
int line	Specify the line to look for a lexeme.
int col	Specify the column to look for a lexeme.

Return Values

Return Value	Description
UdbLexeme	The lexeme found at the specified location.

Example Usage

```
lexeme = udbLexerLexeme(lexer, line, col);
```

See Also

udbLexemeNext on page 5–44

udbLexemePrevious on page 5–45

udbLexerFirst on page 5–50

udbLexerLexemes on page 5–52

udbLexerLexemes

Description Returns an array of all lexemes in the lexer. If startLine and endLine are non-zero, returns an array of all lexemes in that range of the lexer.

Syntax

```
#include "udb/udb.h"
UdbLexeme udbLexerLexemes(UdbLexer lexer,
                          int startLine, int endLine,
                          UdbLexeme **lexeme)
```

Arguments

Argument	Description
UdbLexer lexer	Specify the lexer.
int startLine	Specify the first line in the lexer at which to look for lexemes.
int endLine	Specify the last line in the lexer at which to look for lexemes.
UdbLexeme **lexeme	Points to the array of lexemes returned.

Return Values

Return Value	Description
int	Returns true if lexemes were found at the specified location; false otherwise.

Example Usage

```
status = udbLexerLexemes(lexer, start, end, &lexList);
```

See Also

udbLexemeNext on page 5–44

udbLexemePrevious on page 5–45

udbLexerFirst on page 5–50

udbLexerLexeme on page 5–51

udbLexerLines

Description Returns the number of lines in the specified lexer.

Syntax

```
#include "udb/udb.h"
int udbLexerLines(UdbLexer lexer)
```

Arguments

Argument	Description
UdbLexer lexer	Specify the lexer.

Return Values

Return Value	Description
int	The number of lines in the specified lexer.

Example Usage

```
lines = udbLexerLines(lexer);
```

See Also

udbLexemeLineBegin on page 5–42

udbLexemeLineEnd on page 5–43

udbLexerNew

Description

Creates a lexer for the specified entity. The original source file must be readable and unchanged since the last database parse.

Syntax

```
#include "udb/udb.h"
UdbStatus udbLexerNew(UdbEntity entity,
                      int ents,
                       UdbLexer lexer)
```

Arguments

Argument	Description
UdbEntity entity	Specify the entity for which to create a lexer.
int ents	If TRUE, associate identifiers with entities.
UdbLexer *lexer	Location at which to create the lexer.

Return Values

Return Values	Description
Udb_statusOkay	Lexer created successfully.
Udb_statusLexerFileModified	Lexer cannot be created because the file was modified since the database was analyzed.
Udb_statusLexerFileUnreadable	Lexer cannot be created because the file is unreadable.
Udb_statusLexerUnsupportedLanguage	Lexer cannot be created because the file is in an unsupported language.

Example Usage

```
status = udbLexerNew(entity, numLexemes, *lexer);
```

See Also

udbLexerDelete on page 5–49

udbLibraryCheckEntity

Description

Return true if the specified entity is in the specified list of libraries.

Applies only to *Understand for Ada* and *Understand for Java* databases and only the "standard" library is supported.

Syntax

```
#include "udb/udb.h"
int udbLibraryCheckEntity(UdbEntity entity,
                          UdbLibrary *liblist)
```

Arguments

Argument	Description
UdbEntity entity	The specified entity
UdbLibrary *liblist	The specified library list

Return Values

Return Value	Description
int	Returns true if the specified entity is in the specified list of libraries; false otherwise.

Example Usage

```
udbLibraryList ("~standard", &libraries, &size);
if (udbLibraryCheckEntity (entity, libraries)) {...}
udbLibraryListFree(libraries);
```

See Also

udbEntityLibrary on page 5–19

udbLibraryCompare on page 5–56

udbLibraryFilterEntity on page 5–57

udbLibraryList on page 5–58

udbLibraryListFree on page 5–59

udbLibraryName on page 5–60

udbLibraryCompare

Description

Compare two library structures. Return a value suitable for typical comparison operations.

Applies only to Understand for Ada databases and only the "standard" Ada library is supported.

Syntax

```
#include "udb/udb.h"
int  udbLibraryCompare(UdbLibrary lib1, UdbLibrary lib2)
```

Arguments

Argument	Description
UdbLibrary lib1	The first library to compare
UdbLibrary lib2	The second library to compare

Return Values

Return Value	Description
int	Returns: 0 if lib1 is the same as lib2 <0 if lib1 < lib2 >0 if lib1 > lib2

Example Usage

```
if (! udbLibraryCompare (lib1, lib2) ) {...}
```

See Also

udbEntityLibrary on page 5–19

udbLibraryCheckEntity on page 5–55

udbLibraryFilterEntity on page 5–57

udbLibraryList on page 5–58

udbLibraryListFree on page 5–59

udbLibraryName on page 5–60

udbLibraryFilterEntity

Description

Filter the specified list of entities with the specified library filter and return a new, allocated list. Use *udbListEntityFree* to free the allocated list when done.

Applies only to Understand for Ada databases and only the "standard" Ada library is supported.

Syntax

```
#include "udb/udb.h"
void udbLibraryFilterEntity(UdbEntity *ents,
                            char *filter,
                            UdbEntity **newents,
                            int *num);
```

Arguments

Argument	Description
UdbEntity ents	Specify the list of entities
char *filter	Specify the library filter
UdbEntity **newents	Returns the allocated list of new entities
int *num	If not NULL, returns the size of the new list

Return Values

There is no function return value.

Example Usage

```
udbLibraryFilterEntity(entities, "~standard",
&entities,  &size);
```

See Also

udbEntityLibrary on page 5–19

udbLibraryCheckEntity on page 5–55

udbLibraryCompare on page 5–56

udbLibraryList on page 5–58

udbLibraryListFree on page 5–59

udbLibraryName on page 5–60

udbLibraryList

Description

Returns an allocated list of libraries. Use *udbLibraryListFree()* to free the list.

Applies only to Understand for Ada databases and only the "standard" Ada library is supported.

Syntax

```
#include "udb/udb.h"
void udbLibraryList(char *filter,
                    UdbLibrary **liblist,
                    int *num)
```

Arguments

Argument	Description
char *filter	Specify the library filter
UdbLibrary **liblist	Returns the allocated list of libraries
int *num	If not NULL, returns the size of the library list

Return Values

There is no function return value.

Example Usage

```
udbLibraryList ("~standard", &libraries, &size);
```

See Also

udbEntityLibrary on page 5–19

udbLibraryCheckEntity on page 5–55

udbLibraryCompare on page 5–56

udbLibraryFilterEntity on page 5–57

udbLibraryListFree on page 5–59

udbLibraryName on page 5–60

udbLibraryListFree

Description

Free the list of libraries created with *udbLibraryList()*.

Applies only to Understand for Ada databases and only the "standard" Ada library is supported.

Syntax

```
#include "udb/udb.h"
void   udbLibraryListFree(UdbLibrary *liblist)
```

Arguments

Argument	Description
UdbLibrary *liblist	Specify the allocated library list to free

Return Values

There is no function return value.

Example Usage

```
udbLibraryList ("~standard", &libraries, &size);
```

See Also

udbEntityLibrary on page 5–19

udbLibraryCheckEntity on page 5–55

udbLibraryCompare on page 5–56

udbLibraryFilterEntity on page 5–57

udbLibraryList on page 5–58

udbLibraryName on page 5–60

udbLibraryName

Description

Return the name associated with the specified library. The name is temporary.

Applies only to Understand for Ada databases and only the "standard" Ada library is supported.

Syntax

```
#include "udb/udb.h"
char  *udbLibraryName(UdbLibrary)
```

Arguments

Argument	Description
UdbLibrary library	Specify the library

Return Values

Return Value	Description
char *	Returns a temporary name associated with the specified library or NULL if the entity is not associated with a library.

Example Usage

```
library = udbEntityLibrary(entity);
if (library) {
    printf ("%s is in library %s\n",
                  udbEntityNameLong(entity),
                  udbLibraryName(library) );
}
```

See Also

udbEntityLibrary on page 5–19

udbLibraryCheckEntity on page 5–55

udbLibraryCompare on page 5–56

udbLibraryFilterEntity on page 5–57

udbLibraryList on page 5–58

udbLibraryListFree on page 5–59

udbLicenseInfo

Description

Returns information about the current *Understand* license. The database must be opened (or have been attempted to be opened) in order for the license information to be available. All strings returned are temporary, non-allocated.

Syntax

```
#include "udb/udb.h"
void udbLicenseInfo(char **code, char **expire,
                    char **file, char **hostid);
```

Arguments

Argument	Description
char **code	Return a temporary, non-allocated string containing the license or registration code, or NULL if none.
char **expire	Return a temporary, non-allocated string containing the time until expiration of trial, in seconds, or NULL if there is no expiration of the license.
char **file	Return a temporary, non-allocated string containing the license file path, or NULL if no license file.
char **hostid	Return a temporary, non-allocated string containing the hostid, or NULL if the hostid cannot be determined.

Return Values

There is no function return value.

Example Usage

```
udbLicenseInfo(&licCode, &licExpire, &licFile,
&licHostid);
printf ("Using license ");
if (licFile)
   printf ("in file %s ", licFile);
if (licHostid)
  printf ("from host %s ", licHostid);
if (licCode)
  printf ("with license code %s ", licCode);
if (licExpire)
  printf ("that expires in %s.\n", licExpire);
else
  printf ("that has no expiration date.");
```

See Also

udbDbOpen on page 5–11

udbInfoBuild on page 5–26

udbListEntity

Description Return an allocated list of all entities. After a database update, the list is invalid and must be retrieved again.

Syntax

```
#include "udb/udb.h"
void  udbListEntity(UdbEntity **list, int *items);
```

Arguments

Argument	Description
UdbEntity **list	If not NULL, return a non-allocated list of entities
int *items	If not NULL, return size of list.

Return Values There are no function return values.

Example Usage

```
/* get list of all entities */
udbListEntity(&all_ents, &num_ents);

/* output name of each entity in list */
for (i=0; i<num_ents; i++)   {
   printf ("%s\n",  udbEntityNameLong(all_ents[i]) );
}
udbListEntityFree (all_ents);
```

See Also *udbListEntityFilter* on page 5–63

udbListEntityFree on page 5–64

udbListEntityFilter

Description

Filter the specified list of entities, using the kinds specified, and return a new, allocated, array. Use *udbListEntityFree()* to free this list.

The original list is automatically freed, so the most common usage of this call is to specify the same entity list for both the input and output entity lists. For example:

```
udbListEntityFilter(myEnts, kinds, &myEnts,
&myEntsSize);
```

Syntax

```
#include "udb/udb.h"
void  udbListEntityFilter(UdbEntity *ents,
                          UdbKindList kinds,
                         UdbEntity **newents, int *items)
```

Arguments

Argument	Description
UdbEntity *ents	Original list of entities; allocated
UdbKindList kinds	If not NULL, entity kinds to filter; allocated
UdbEntity **newents	Return allocated array of entities
int *items	If not NULL, return size of list.

Return Values

There are no function return values.

Example Usage

```
udbListEntity(&list, NULL);
udbKindList ("function", &kinds );
/* filter entity list to only specified kinds */
udbListEntityFilter(list, kinds, &funclist, &size);
...
udbListEntityFree(funclist);
```

See Also

udbEntityKind on page 5–17
udbLibraryFilterEntity on page 5–57
udbListEntity on page 5–62
udbListEntityFree on page 5–64
udbKindList on page 5–31
udbKindListCopy on page 5–32
udbKindListFree on page 5–33
udbKindLocate on page 5–34
udbKindParse on page 5–36
udbListReferenceFilter on page 5–71

udbListEntityFree

Description Free the list of entities that was returned from *udbLibraryFilterEntity()*, *udbListEntity()*, or *udbListEntityFilter()*.

Syntax

```
#include "udb/udb.h"
void  udbListEntityFree(UdbEntity *list)
```

Arguments

Argument	Description
UdbEntity *list	Specify entity list to free

Return Values There are no function return values.

Example Usage

```
udbListEntityFilter(list, kinds, &newlist, &size);
...
udbListEntityFree(newlist);
```

See Also

udbListEntity on page 5–62
udbListEntityFilter on page 5–63
udbLibraryListFree on page 5–59

udbListFile

Description Return a non-allocated, temporary, list of all analyzed file entities.

Syntax

```
#include "udb/udb.h"
void udbListFile(UdbEntity **list, int *items)
```

Arguments

Argument	Description
UdbEntity **list	Return a temporary list of all analyzed file entities (not allocated).
int *items	Return size of list.

Return Values There are no function return values.

Example Usage

```
udbListFile(&list, &size);
```

See Also

udbIsKindFile on page 5–28

udbListEntity on page 5–62

udbListEntityFilter on page 5–63

udbListReferenceFile on page 5–70

udbLookupFile on page 5–76

udbReferenceFile on page 5–94

udbListKindEntity

Description

Return an allocated list of all entity kinds. Call *udbListKindFree()* to free this list.

Refer to the language-specific listings of entity kinds in Chapter 7, “Entity and Reference Kinds”.

Syntax

```
#include "udb/udb.h"
void  udbListKindEntity(UdbKind **list, int *items)
```

Arguments

Argument	Description
UdbKind **list	Return allocated array of all entity kinds.
int *items	If not NULL, return size of array.

Return Values

There are no function return values.

Example Usage

This example gets the list of entity kinds, prints the long name of each entity kind, and frees the list of kinds.

```
udbListKindEntity(&kinds, &size);
printf ("All Entity Kinds: \n");
for (i=0; i<size; i++) {
    printf ("  %s\n", udbKindLongname(kinds[i]) );
}
udbListKindFree(kinds);
```

See Also

udbListKindFree on page 5–67

udbListKindReference on page 5–68

udbListKindFree

Description Free the kind list created using *udbListKindEntity()* or *udbListKindReference()*.

Syntax

```
#include "udb/udb.h"
void  udbListKindFree(UdbKind *list)
```

Arguments

Argument	Description
UdbKind *list	The specified kind list

Return Values There are no function return values.

Example Usage

```
udbListKindFree(kinds);
```

See Also *udbListKindEntity* on page 5–66

udbListKindReference on page 5–68

udbListKindReference

Description

Return allocated list of all reference kinds. Call *udbListKindFree()* to free this list.

Refer to the language-specific listings of reference kinds in Chapter 7, “Entity and Reference Kinds”.

Syntax

```
#include "udb/udb.h"
void   udbListKindReference(UdbKind **list, int *items)
```

Arguments

Argument	Description
UdbEntity **list	Return allocated array of all reference kinds.
int *items	If not NULL, return size of array.

Return Values

There are no function return values.

Example Usage

This example gets the list of reference kinds, prints the long name of each reference kind, and frees the list of reference kinds.

```
udbListKindReference(&refkinds, &size);
printf ("All Reference Kinds: \n");
for (i=0; i<size; i++) {
    printf ("  %s\n", udbKindLongname(refkinds[i]) );
}
udbListKindFree(refkinds);
```

See Also

udbListKindFree on page 5–67

udbListKindEntity on page 5–66

udbListReference

Description

Lookup the list of references for the specified entity. Call *udbListReferenceFree()* to free this list.

Syntax

```
#include "udb/udb.h"
void udbListReference(UdbEntity entity,
                      UdbReference **refs,
                      int *items)
```

Arguments

Argument	Description
UdbEntity entity	Specify the entity to obtain references for.
UdbReference **refs	Return allocated array of references.
int *items	If not NULL, return size of references array.

Return Values

There are no function return values.

Example Usage

This example gets all references for the entity and prints out the reference information, and then frees the reference list.

```
udbListReference(entity, &refs, &size);
printf ("References found for %s: \n",
udbEntityNameLong(entity) );
for (i=0; i<size; i++) {
    printf ("  %s: %s[%d]\n",

udbKindShortname(udbReferenceKind(refs[i])),

udbEntityNameShort(udbReferenceFile(refs[i])),
              udbReferenceLine(refs[i]));
}
udbListReferenceFree(refs);
```

See Also

udbEntityRefs on page 5–24

udbListReferenceFile on page 5–70

udbListReferenceFilter on page 5–71

udbListReferenceFree on page 5–72

udbLookupEntityByReference on page 5–74

udbLookupReferenceExists on page 5–77

udbReferenceEntity on page 5–93

udbListReferenceFile

Description Lookup the list of all references within the specified file. Call *udbListReferenceFree()* to free this list.

Syntax

```
#include "udb/udb.h"
void  udbListReferenceFile(UdbEntity file,
                           UdbReference **refs,
                           int *items)
```

Arguments

Argument	Description
UdbEntity file	Specify the file entity to obtain references for.
UdbReference **refs	Return allocated array of references.
int *items	If not NULL, return size of references array.

Return Values There are no function return values.

Example Usage

```
udbListReferenceFile(fileent, &refs, &size);
...
udbListReferenceFree(refs);
```

See Also

udbIsKindFile on page 5–28

udbListFile on page 5–65

udbListReference on page 5–69

udbListReferenceFilter on page 5–71

udbListReferenceFree on page 5–72

udbLookupFile on page 5–76

udbReferenceFile on page 5–94

udbListReferenceFilter

Description

Filter the specified list of references, using the reference kinds and/or the entity kinds specified, and return a new allocated array of references. If unique is specified, the newrefs array will only contain the first reference for each unique entity.

Call *udbListReferenceFree()* to free this list.

Syntax

```
#include "udb/udb.h"
void  udbListReferenceFilter(UdbReference *refs,
                             UdbKindList refkinds,
                             UdbKindList entkinds,
                             int unique,
                             UdbReference **newrefs,
                             int *items)
```

Arguments

Argument	Description
UdbReference *refs	Original list of references; non-allocated.
UdbKindList refkinds	If not NULL, reference kinds to filter; allocated.
UdbKindList entkinds	If not NULL, entity kinds to filter; allocated.
int unique	If specified, only one reference per unique entity is returned
UdbReference **newrefs	Return allocated array of references.
int *items	If not NULL, return size of newrefs array.

Return Values

There are no function return values.

Example Usage

This example filters an existing list of reference to include only those references where there is an active callby or useby reference, or where the entity is declared or defined.

```
udbListReferenceFilter(refs,
        udbKindParse("callby
~inactive,declarein,definein, useby ~inactive"),
         NULL, 0, &xrefs, &items);
```

See Also

udbListReference on page 5–69

udbListReferenceFree on page 5–72

udbListEntityFilter on page 5–63

udbListReferenceFree

Description Free the list of references that was returned from *udbListReference()*, *udbListReferenceFile()*, or *udbListReferenceFilter()*.

Syntax

```
#include "udb/udb.h"
void udbListReferenceFree(UdbReference *refs)
```

Arguments

Argument	Description
UdbReference *refs	Reference list to free.

Return Values There are no function return values.

Example Usage

```
udbListReferenceFree(refs);
```

See Also

udbListReference on page 5–69

udbListReferenceFile on page 5–70

udbListReferenceFilter on page 5–71

udbLookupEntity

Description Returns a list of entities that match the specified name, kind, and/or shortname.

Syntax

```
#include "udb/udb.h"
void  udbLookupEntity( char *name, char *kind,
                       int shortname,
                       UdbEntity **entities,
                       int *matchline);
```

Arguments

Argument	Description
char *name	Name of entity to lookup.
char *kind	Kind of entity to lookup.
int shortname	If true, allow shortname matches. Longname matches are always allowed.
UdbEntity **entities	Returns allocated list of entities. This list may be freed with udbListEntityFree.
int *matchline	If not NULL, returns the size of the entities list.

Return Values There are no return values.

Example Usage

```
udbLookupEntity("tool", NULL, NULL, &entity, match);
```

See Also

udbLookupEntityByReference on page 5–74

udbLookupFile on page 5–76

udbLookupReferenceExists on page 5–77

udbLookupEntityByReference

Description

Lookup the specified entity by name, based on the source file, line, and column that an entity reference occurs at.

Syntax

```
#include "udb/udb.h"
UdbEntity udbLookupEntityByReference(UdbEntity file,
                                     char *name,
                                    int line, int col,
                                      int *matchline)
```

Arguments

Argument	Description
UdbEntity file	File entity.
char *name	Name of entity to lookup.
int line	Line reference occurs on
int col	Column reference occurs on
int *matchline	If not NULL, return matched line, or 0

Return Values

Return Value	Description
UdbEntity	Return the referenced entity.

Example Usage

```
refentity = udbLookupEntityByReference(
                        udbLookupFile(filename),
                        entityname,
                        refline, refcol,
                        NULL);
```

See Also

udbEntityRefs on page 5–24

udbReferenceEntity on page 5–93

udbLookupEntity on page 5–73

udbLookupFile on page 5–76

udbLookupReferenceExists on page 5–77

udbLookupEntityByUniquename

Description Look up the specified entity by a uniquename provided by the udbEntityNameUnique function. Returns NULL if the entity is not found.

Syntax

```
#include "udb/udb.h"
UdbEntity udbLookupEntityByUniquename(
                                        char *uniquename)
```

Arguments

Argument	Description
char *name	Uniquename of entity to look up.

Return Values

Return Value	Description
UdbEntity	Return the referenced entity.

Example Usage

```
uniquename = udbEntityNameUnique(entity));
...
refentity = udbLookupEntityByUniquename(uniquename);
```

See Also *udbEntityNameUnique* on page 5–23

udbLookupFile

Description Lookup the specified file entity by name.

Syntax

```
#include "udb/udb.h"
UdbEntity   udbLookupFile(char *name)
```

Arguments

Argument	Description
char *name	Name of file entity to lookup.

Return Values

Return Value	Description
UdbEntity	Return the file entity.

Example Usage

```
fileentity = udbLookupFile(filename);
```

See Also

udbIsKindFile on page 5–28

udbListFile on page 5–65

udbListReferenceFile on page 5–70

udbReferenceFile on page 5–94

udbLookupEntity on page 5–73

udbLookupEntityByReference on page 5–74

udbLookupReferenceExists on page 5–77

udbLookupReferenceExists

Description Return 1 if the specified entity has any references of the general kind specified by the list of reference kinds. Returns 1 if the list is NULL.

Syntax

```
#include "udb/udb.h"
int  udbLookupReferenceExists(UdbEntity entity,
                              UdbKindList kindlist)
```

Arguments

Argument	Description
UdbEntity entity	Specified entity to lookup
UdbKindList kindlist	Allocated kind list

Return Values

Return Value	Description
int	1 if the specified entity has any reference of the general kind specified by the list of references or if the reference kind list is NULL.

Example Usage

```
if (udbLookupReferenceExists(entity, refkinds) ) {...}
```

See Also

udbEntityRefs on page 5–24

udbListReference on page 5–69

udbLookupEntity on page 5–73

udbLookupEntityByReference on page 5–74

udbLookupFile on page 5–76

udbReferenceEntity on page 5–93

udbMetricDescription

Description

Returns the short description of the specified metric.

Syntax

```
#include "udb/udb.h"
char * udbMetricDescription(UdbMetric metric)
```

Arguments

Argument	Description
UdbMetric metric	Specified metric to return description for

Return Values

Return Value	Description
char *	Short description of specified metric.

See Also

udbMetricKind on page 5–81

udbMetricListKind on page 5–83

udbMetricLookup on page 5–86

udbMetricName on page 5–87

udbMetricValue on page 5–88

udbMetricValueProject on page 5–89

udbMetricIsDefinedEntity

Description Return true if the specified metric is defined for the specified entity (kind).

Syntax

```
#include "udb/udb.h"
int udbMetricIsDefinedEntity(UdbMetric metric,
                             UdbEntity entity)
```

Arguments

Argument	Description
UdbMetric metric	Specify metric to check for.
UdbEntity entity	Specify entity to check for.

Return Values

Return Value	Description
int	True if the specified metric is defined for the kind of the specified entity.

Example Usage

```
if (udbMetricIsDefinedEntity(Udb_cMetricCountLineCode,
                             entity ) )
   ...
}
```

See Also

udbMetricIsDefinedProject on page 5–80

udbMetricListEntity on page 5–82

udbMetricLookup on page 5–86

udbMetricName on page 5–87

udbMetricValue on page 5–88

udbMetricIsDefinedProject

Description Return true if the specified metric is defined for the specified programming language.

Syntax

```
#include "udb/udb.h"
int udbMetricIsDefinedProject(UdbMetric metric,
                              UdbLanguage language)
```

Arguments

Argument	Description
UdbMetric metric	Specified metric to check for
UdbLanguage language	Specified project language

UdbLanguage specifies a programming language supported by *Understand* databases. Valid values are:

Return Values	Description
Udb_language_Ada	Project database conforms to Ada.
Udb_language_C	Project database conforms to C or C++.
Udb_language_Fortran	Project database conforms to FORTRAN.
Udb_language_Java	Project database conforms to Java.
Udb_language_Jovial	Project database conforms to JOVIAL.
Udb_language_Pascal	Project database conforms to Pascal.

Return Values

Return Value	Description
int	true if the specified metric is defined for the kind of the specified entity.

Example Usage

```
if (udbMetricIsDefinedProject(Udb_cMetricCountLineCode,
                              Udb_language_C ) )
   ...
}
```

See Also

udbMetricIsDefinedEntity on page 5–79

udbMetricListProject on page 5–85

udbMetricValueProject on page 5–89

udbDbLanguage on page 5–9

udbMetricKind

Description Returns the kind of the specified metric. This can be used to determine what formatting character to use when printing a value with printf.

Syntax

```
#include "udb/udb.h"
UdbMetricKind udbMetricKind(UdbMetric metric)
```

Arguments

Argument	Description
UdbMetric metric	Specify metric.

Return Values

Return Values	Description
Udb_mkind_NONE	No such metric.
Udb_mkind_Integer	Metric returns an integer value.
Udb_mkind_Real	Metric returns a floating point value.

Example Usage

```
kind = udbMetricKind(metric);
```

See Also

udbMetricDescription on page 5–78

udbMetricIsDefinedEntity on page 5–79

udbMetricIsDefinedProject on page 5–80

udbMetricListKind on page 5–83

udbMetricLookup on page 5–86

udbMetricListEntity

Description

Returns all metrics defined for entities that match a specified entity string.

Syntax

```
#include "udb/udb.h"
int udbMetricListEntity(UdbEntity entity
                        UdbMetric **metrics)
```

Arguments

Argument	Description
UdbEntity entity	Specified entity.
UdbMetric **metrics	List of returned metrics.

Return Values

Return Value	Description
int	True if successful. False otherwise.

Example Usage

```
status = udbMetricListEntity(entity, &metrics);
```

See Also

udbMetricIsDefinedEntity on page 5–79

udbMetricListKind on page 5–83

udbMetricListLanguage on page 5–84

udbMetricListProject on page 5–85

udbMetricListKind

Description

Returns all metrics that are defined for entities that match a specified kind string.

Syntax

```
#include "udb/udb.h"
int  udbMetricListKind(char *, UdbMetric metrics)
```

Arguments

Argument	Description
char *name	Kind string to specify which metrics to return.
UdbMetric **metrics	List of returned metrics.

Return Values

Return Value	Description
int	True if successful. False otherwise.

Example Usage

```
status = udbMetricListKind(entity, &metrics);
```

See Also

udbMetricKind on page 5–81

udbMetricListEntity on page 5–82

udbMetricListLanguage on page 5–84

udbMetricListProject on page 5–85

udbMetricListLanguage

Description

Returns all entity metrics defined for the specified language.

Syntax

```
#include "udb/udb.h"
int  udbMetricListLanguage(UdbLanguage language,
                           UdbMetric **metrics)
```

Arguments

Argument	Description
UdbLanguage language	Specify a programming language using one of the values in the following table.
UdbMetric **metrics	Points to list of returned entity metrics.

UdbLanguage may be one of the following values specifying the language of the Understand database:

UdbLanguage Values	Description
Udb_language_ALL	Project supports all languages.
Udb_language_Ada	Project database conforms to Ada.
Udb_language_C	Project database conforms to C or C++.
Udb_language_Fortran	Project database conforms to FORTRAN.
Udb_language_Java	Project database conforms to Java.
Udb_language_Jovial	Project database conforms to JOVIAL.
Udb_language_Pascal	Project database conforms to Pascal.

Return Values

Return Value	Description
int	True if successful. False otherwise.

Example Usage

```
res = udbMetricListLanguage(Udb_language_C, &metrics);
```

See Also

udbDbLanguage on page 5–9

udbEntityLanguage on page 5–18

udbKindLanguage on page 5–30

udbMetricListEntity on page 5–82

udbMetricListKind on page 5–83

udbMetricListProject on page 5–85

udbMetricListProject

Description

Returns all project metrics defined for the specified language.

Syntax

```
#include "udb/udb.h"
int  udbMetricListProject(UdbLanguage language,
                          UdbMetric **metrics)
```

Arguments

Argument	Description
UdbLanguage language	Specify a programming language using one of the values in the following table.
UdbMetric **metrics	Points to list of returned project metrics.

UdbLanguage may be one of the following values specifying the language of the Understand database:

UdbLanguage Values	Description
Udb_language_ALL	Project supports all languages.
Udb_language_Ada	Project database conforms to Ada.
Udb_language_C	Project database conforms to C or C++.
Udb_language_Fortran	Project database conforms to FORTRAN.
Udb_language_Java	Project database conforms to Java.
Udb_language_Jovial	Project database conforms to JOVIAL.
Udb_language_Pascal	Project database conforms to Pascal.

Return Values

Return Value	Description
int	True if successful. False otherwise.

Example Usage

```
res = udbMetricListProject(Udb_language_C, &metrics);
```

See Also

udbDbLanguage on page 5–9
udbMetricIsDefinedProject on page 5–80
udbMetricListEntity on page 5–82
udbMetricListKind on page 5–83
udbMetricListLanguage on page 5–84
udbMetricValueProject on page 5–89

udbMetricLookup

Description Returns the UdbMetric literal for the specified character string.

Syntax

```
#include "udb/udb.h"
UdbMetric udbMetricLookup(char *name)
```

Arguments

Argument	Description
char *name	Text name of metric.

Return Values

Return Value	Description
UdbMetric	Metric that matches the character string.

Example Usage

```
metric = udbMetricLookup("Cyclomatic");
```

See Also

udbLookupEntity on page 5–73

udbLookupEntityByReference on page 5–74

udbLookupFile on page 5–76

udbLookupReferenceExists on page 5–77

udbMetricIsDefinedProject on page 5–80

udbMetricListEntity on page 5–82

udbMetricListKind on page 5–83

udbMetricListProject on page 5–85

udbMetricName

Description Returns the name of the metric as a string when a UdbMetric literal is specified.

Syntax

```
#include "udb/udb.h"
 char * udbMetricName(UdbMetric metric)
```

Arguments

Argument	Description
UdbMetric metric	Specify a metric using the defined literals.

Return Values

Return Value	Description
char *name	Returns text name of metric.

Example Usage

```
metricName = udbMetricName(metric);
```

See Also

udbMetricDescription on page 5–78

udbMetricLookup on page 5–86

udbMetricValue

Description Returns the value of the specified entity metric.

Syntax

```
#include "udb/udb.h"
 float udbMetricValue(UdbEntity entity,
                      UdbMetric metric)
```

Arguments

Argument	Description
UdbEntity entity	Specify the entity for which a metric value should be returned.
UdbMetric metric	Specify the metric for which a value should be returned

Return Values

Return Value	Description
float	Value of specified metric returned as a real number value.

Example Usage

```
metricValue = udbMetricValue(entity, metric);
```

See Also *udbMetricValueProject* on page 5–89

udbMetricValueProject

Description Returns the value of the specified project metric.

Syntax

```
#include "udb/udb.h"
 float udbMetricValueProject(UdbMetric metric)
```

Arguments

Argument	Description
UdbMetric metric	Specify the metric for which a value should be returned

Return Values

Return Value	Description
float	Value of specified metric returned as a real number value.

Example Usage

```
metricValue = udbMetricValueProject(metric);
```

See Also *udbMetricValue* on page 5–88

udbReferenceColumn

Description Return the column position of the specified reference.

Syntax

```
#include "udb/udb.h"
int  udbReferenceColumn(UdbReference ref)
```

Arguments

Argument	Description
UdbReference ref	Specified reference

Return Values

Return Value	Description
int	The column position of specified reference.

Example Usage

```
printf ("%s: %s (%s [%d,%d])\n",

udbKindShortname(udbReferenceKind(ref)),

udbEntityNameShort(udbReferenceEntity(ref)),

udbEntityNameShort(udbReferenceFile(ref)),
                        udbReferenceLine(ref),
                        udbReferenceColumn(ref) );
```

See Also

udbLexemeColumnBegin on page 5–38

udbLexemeColumnEnd on page 5–39

udbListReference on page 5–69

udbReferenceEntity on page 5–93

udbReferenceFile on page 5–94

udbReferenceKind on page 5–95

udbReferenceLine on page 5–96

udbReferenceScope on page 5–97

udbReferenceCopy

Description

Create an allocated copy of the specified reference. This copy must be freed with *udbReferenceCopyFree()*.

Syntax

```
#include "udb/udb.h"
UdbReference  udbReferenceCopy(UdbReference ref)
```

Arguments

Argument	Description
UdbReference ref	Specified reference to make a copy of.

Return Values

Return Value	Description
UdbReference	The new reference copy.

Example Usage

```
newref = udbReferenceCopy(ref);
```

See Also

udbReferenceCopyFree on page 5–92

udbReferenceCopyFree

Description — Free an allocated reference created with *udbReferenceCopy()*.

Syntax

```
#include "udb/udb.h"
void  udbReferenceCopyFree(UdbReference ref)
```

Arguments

Argument	Description
UdbReference ref	Specified reference to free.

Return Values — There is no return value.

Example Usage

```
udbReferenceCopyFree(ref);
```

See Also — *udbReferenceCopy* on page 5–91

udbReferenceEntity

Description Return the entity of the specified reference. This is the entity being referenced.

Syntax

```
#include "udb/udb.h"
UdbEntity  udbReferenceEntity(UdbReference ref)
```

Arguments

Argument	Description
UdbReference ref	Specified reference.

Return Values

Return Value	Description
UdbEntity	Entity Being Referenced.

Example Usage

```
printf ("%s: %s (%s [%d,%d])\n",

udbKindShortname(udbReferenceKind(ref)),

udbEntityNameShort(udbReferenceEntity(ref)),

udbEntityNameShort(udbReferenceFile(ref)),
                        udbReferenceLine(ref),
                        udbReferenceColumn(ref) );
```

See Also

udbEntityRefs on page 5–24

udbLexemeReference on page 5–46

udbListReference on page 5–69

udbLookupEntityByReference on page 5–74

udbLookupReferenceExists on page 5–77

udbReferenceColumn on page 5–90

udbReferenceFile on page 5–94

udbReferenceKind on page 5–95

udbReferenceLine on page 5–96

udbReferenceScope on page 5–97

udbReferenceFile

Description Return the file of the specified reference.

Syntax

```
#include "udb/udb.h"
UdbEntity  udbReferenceFile(UdbReference ref)
```

Arguments

Argument	Description
UdbReference ref	Specified reference.

Return Values

Return Value	Description
UdbEntity	File entity being referenced.

Example Usage

```
printf ("%s: %s (%s [%d,%d])\n",

udbKindShortname(udbReferenceKind(ref)),

udbEntityNameShort(udbReferenceEntity(ref)),

udbEntityNameShort(udbReferenceFile(ref)),
                        udbReferenceLine(ref),
                        udbReferenceColumn(ref) );
```

See Also

udbIsKindFile on page 5–28

udbListFile on page 5–65

udbListReferenceFile on page 5–70

udbLookupFile on page 5–76

udbReferenceColumn on page 5–90

udbReferenceEntity on page 5–93

udbReferenceKind on page 5–95

udbReferenceLine on page 5–96

udbReferenceScope on page 5–97

udbReferenceKind

Description Return the kind of the specified reference.

Syntax

```
#include "udb/udb.h"
UdbKind  udbReferenceKind(UdbReference ref)
```

Arguments

Argument	Description
UdbReference ref	Specified reference.

Return Values

Return Value	Description
UdbKind	Kind of Reference

Example Usage

```
printf ("%s: %s (%s [%d,%d])\n",

udbKindShortname(udbReferenceKind(ref)),

udbEntityNameShort(udbReferenceEntity(ref)),

udbEntityNameShort(udbReferenceFile(ref)),
                        udbReferenceLine(ref),
                        udbReferenceColumn(ref) );
```

See Also

udbEntityKind on page 5–17

udbKindLongname on page 5–35

udbKindShortname on page 5–37

udbReferenceColumn on page 5–90

udbReferenceEntity on page 5–93

udbReferenceFile on page 5–94

udbReferenceLine on page 5–96

udbReferenceScope on page 5–97

udbReferenceLine

Description Return the line of the specified reference.

Syntax

```
#include "udb/udb.h"
int  udbReferenceLine(UdbReference ref)
```

Arguments

Argument	Description
UdbReference ref	Specified reference

Return Values

Return Value	Description
int	The line position of specified reference.

Example Usage

```
printf ("%s: %s (%s [%d,%d])\n",

udbKindShortname(udbReferenceKind(ref)),

udbEntityNameShort(udbReferenceEntity(ref)),

udbEntityNameShort(udbReferenceFile(ref)),
                        udbReferenceLine(ref),
                        udbReferenceColumn(ref) );
```

See Also

udbLexemeLineBegin on page 5–42

udbLexemeLineEnd on page 5–43

udbLexerLines on page 5–53

udbReferenceColumn on page 5–90

udbReferenceEntity on page 5–93

udbReferenceFile on page 5–94

udbReferenceKind on page 5–95

udbReferenceScope on page 5–97

udbReferenceScope

Description Return the scope entity of the specified reference. This is the entity that is performing the reference.

Syntax

```
#include "udb/udb.h"
UdbEntity  udbReferenceScope(UdbReference ref)
```

Arguments

Argument	Description
UdbReference ref	Specified reference

Return Values

Return Value	Description
UdbEntity	Return the scope entity (the entity that is performing the reference).

Example Usage

```
refscope = udbReferenceScope(ref);
```

See Also

udbReferenceColumn on page 5–90

udbReferenceEntity on page 5–93

udbReferenceFile on page 5–94

udbReferenceKind on page 5–95

udbReferenceLine on page 5–96

Chapter 6 C API Code Samples

This chapter provides simple, sample code for use as examples that can also be used as a starting point for creating your own application. All examples use an *Understand for C++* database. API usage with other Understand databases/languages is similar.

The following sections show each different example as a separate function within a larger module. The first section shows the common elements within a "main" function, from which any of the other example functions could be called.

Each example shows the C code, an explanation of the code, and sample output produced by the code.

This chapter contains the following:

Section	Page
Open Database and Get All Entities	6–2
Report All Entities	6–4
Report All Files	6–6
Report Functions with their Parameters and Types	6–7
Report Global Objects	6–9
Report All Structs and their Member Types	6–11
Find All References To and From an Entity	6–13

Open Database and Get All Entities

Description

This example shows a sample main() function, where the sample database is opened and closed.

This main() function can be used as a starting point for any of the other functional examples provided.

Sample Code

```
#include <stdio.h>
#include <stdlib.h>
#include "udb.h"

static char         *dbFilename = "test.udc";

main ( int argc,  char *argv[])    {
   status = udbDbOpen(dbFilename);
   if (status)  {
      printf ("unable to open  Understand database: %s \n", dbFilename);
      switch (status)  {
         case Udb_statusDBAlreadyOpen:
                        printf("database already open\n");                    break;
         case Udb_statusDBChanged:
                        printf("database has been changed\n");                break;
         case Udb_statusDBCorrupt:
                        printf("database is corrupt\n");                      break;
         case Udb_statusDBOldVersion:
                        printf("database is old version\n");                  break;
         case Udb_statusDBUnknownVersion:
                        printf("unknown version\n");                          break;
         case Udb_statusDBUnableOpen:
                        printf("unable to locate file\n");                    break;
         case Udb_statusNoApiLicense:
                        printf("no Understand license available\n");          break;
         case Udb_statusNoApiLicenseAda:
                        printf("no Understand Ada license available\n");      break;
         case Udb_statusNoApiLicenseC:
                        printf("no Understand C license available\n");        break;
         case Udb_statusNoApiLicenseFtn:
                        printf("no Undertstand Fortran license available\n"); break;
         case Udb_statusNoApiLicenseJava:
                        printf("no Understand Java license available\n");     break;
         case Udb_statusNoApiLicenseJovial:
                        printf("no Understand Jovial license available\n");   break;
         case Udb_statusNoApiLicensePascal:
                        printf("no Understand Pascal license available\n");   break;
         default:
                        printf("unable to access database\n");                break;
      }
```

```
40       exit (EXIT_FAILURE);
41    }
42    /* call any of the example functions here */
43    udbDbClose();
```

Explanation of Sample Code

This example shows what is needed in order to call the API functions and how to open and close a database.

line 3**: Include "udb.h"** in order to use the *Understand* C API.

line 4: For demonstration purposes, the Understand database name being used is "test.udc". Change this name to **specify the name of the *Understand* database** that you have already created and analyzed.

line 7: Call udbSetLicense() to **specify directory path to the *Understand* license file** to use. The license is not consumed until the call to udbDbOpen() is performed. The call to udbSetLicense() is optional. The API will attempt to locate a license file in the following order:

```
1. udbSetLicense()
2. environment variable STILICENSE
3. For Windows, the STILICENSE key in
     HKEY_CURRENT_USER\Software\ Scientific Toolworks,
Inc.
4. conf/license inside sti home directory
```

lines 8-21: **Open the *Understand* database** and check the return status. The return status will indicate if the database cannot be opened for any reason, including if a license cannot be obtained.

line 24: Call any of the other example functions provided here or one of your own creation.

line 25: **Close the database**.

Report All Entities

Description This example retrieves a list of all project entities and reports the short name for each entity and its entity kind.

Sample Code

```
1  static void reportAllEntities ()    {
2     UdbEntity *ents;
3     int       entsSize;
4     int      i;
5
6     udbListEntity(&ents, &entsSize);
7     printf ("\nEntity Name (kind)\n");
8     for (i=0; i<entsSize; i++)
9        printf (" %s (%s)\n",
10                udbEntityNameShort(ents[i]),
11                udbKindShortname(udbEntityKind(ents[i])) );
12    udbListEntityFree(ents);
13 }
```

Explanation of Sample Code

lines 2-3: Declaration for the list of all project entities and the number of entities in the list.

line 5: udbListEntity() **obtains a list of all entities** in the project. The second parameter in the call returns the number of entities in the list. You may also pass a NULL parameter for either argument.

line 6: Just prints a header. We will display the short name of the entity followed by the entity kind in parentheses.

line 7: **Loop through the entity list**. Note that the number of entities in the list is returned from udbListEntity(), and the loop through that list of entities is indexed from 0..n-1.

lines 8-10: **Print the name and the kind name for each entity**. We have chosen to print only the short name of the entity, but as an alternative, we could print the long name of an entity. Long names include the name of the compilation unit for entity and function names (Ada), the class name for class members (C++), and the full file path of file names. The shortname of the entity kind is sufficient.

line 11: **Free the entity list** when done.

Sample Output

```
Entity Name (kind)
  PIXEL_SIZE (Macro)
  RGBAPIXEL (Typedef)
  RGBA_BLUE (Macro)
  RGBA_GREEN (Macro)
```

```
RGBA_RED (Macro)
RGBA_ALPHA (Macro)
CBmp (Class)
DECLARE_DYNAMIC (Private Member Function)
CBmp (Public Member Function)
~CBmp (Public Member Function)
```

Report All Files

Description This example prints the (long) name of each analyzed file in the project. The long name of a file is the full path of the file.

Sample Code

```
1  static void reportAllFiles() {
2   UdbEntity  *fileEnts;
3   int        fileEntsSize;
4   int       i;
5
6     udbListFile (&fileEnts, &fileEntsSize);
7     printf ("\nProject Files:\n");
8     for (i=0; i<fileEntsSize; i++)
9        printf (" %s \n", udbEntityNameLong(fileEnts[i]) );
10    udbListEntityFree(fileEnts);
11 }
```

Explanation of Sample Code

line 2-3: Declare variables for the list of entities and the list size.

line 5: **Retrieve the list of analyzed file entities**.

line 6: Print a header message.

line 7: **Loop through all the file entities** in the list.

line 8: **Print the long name**, which is the full path, of the file entity. Alternatively, the short name could be printed, which would show only the file name and not the full path.

line 9: **Free the file entity list** when done.

Sample Output

```
Project Files:
   D:\examples\cpp\Paintlib\code\src\stdpch.h
   D:\examples\cpp\Paintlib\code\src\bitmap.h
   D:\examples\cpp\Paintlib\code\src\datasrc.h
   D:\examples\cpp\Paintlib\code\src\picdec.h
   D:\examples\cpp\Paintlib\code\src\config.h
   D:\examples\cpp\Paintlib\code\src\tga.h

D:\examples\cpp\Paintlib\code\djgpp\testdec\testdec.cpp
   D:\examples\cpp\Paintlib\code\src\anybmp.cpp
```

Report Functions with their Parameters and Types

Description This example reports all Functions, with their parameters and types, if any. The return type of the function (if any) is also reported.

Sample Code

```
1  static void reportFunctionParameters()  {
2      UdbEntity    *ents;
3      int          entsSize;
4      UdbReference *refs;
5      int          refsSize;
6      int         i,j;
7
8     udbListEntity(&ents, &entsSize);
9     udbListEntityFilter (ents, udbKindParse("function"), &ents, &entsSize);
10    printf ("\nFunction (return type) and its Parameters:\n");
11    for (i=0; i<entsSize; i++) {
12       printf ("\n %s ", udbEntityNameLong(ents[i]) );
13       if (udbEntityTypetext(ents[i]))
14          printf ("(%s)", udbEntityTypetext(ents[i]) );
15       printf ("\n");
16       udbListReference(ents[i], &refs, &refsSize);
17       udbListReferenceFilter(refs,
18                              udbKindParse("define"), udbKindParse("parameter"), 0,
19                              &refs, &refsSize);
20       for (j=0; j<refsSize; j++) {
21          printf ("   %s %s \n",
22                   udbEntityTypetext(udbReferenceEntity(refs[j])),
23                   udbEntityNameShort(udbReferenceEntity(refs[j])) );
24       }
25      udbListReferenceFree(refs);
26    }
27    udbListEntityFree(ents);
28 }
```

Explanation of Sample Code

lines 2-3: Declare variables for the list of entities and the list size. In this case, the list will be filtered to include only functions.

lines 4-5: Declare variables for references of a particular entity, in this case a function entity.

line 7: **Retrieve the list of all entities**.

line 8: **Filter the list of all entities to contain only functions,** replacing the original entity list with the new filtered list.

line 9: Print the header.

line 10: **Loop through all the function entities**.

line 11: **Print the function name**. The longname is printed here which will include the class name (C+) or compilation unit (Ada) where applicable. Alternatively, the short name of the function could be specified instead.

lines 12-13: **Print the return type of the function**, if any. The return type is the text returned from *udbEntityTypetext()*.

line 15: **Retrieve the list of all References** for the current function entity.

lines 16-18: **Filter the list of References** to include only those that define parameters (i.e. contain "define" reference types and "parameter" entity types). The same reference list is used.

line 19: **Loop through references**, which have been filtered to include only those where parameters are defined.

lines 20-22: **Print the parameter type and name**.

line 24: **Free the list of References.**

line 26: **Free the list of entities**.

Sample Output

```
Function (return type) and its Parameters:
 Trace (void)
 CBmp::DECLARE_DYNAMIC
 CBmp::CBmp
 CBmp::~CBmp
 CBmp::CreateCopy (void)
   CBmp & rSrcBmp
   int BPPWanted
 CBmp::Create (void)
   LONG Width
   LONG Height
   WORD BitsPerPixel
   BOOL bAlphaChannel
 CBmp::SetPaletteEntry (void)
   BYTE Entry
   BYTE r
   BYTE g
   BYTE b
   BYTE a
CBmp::GetWidth (int)
CBmp::GetHeight (int)
CBmp::GetMemUsed (virtual long)
```

Report Global Objects

Description This example reports usages of all global objects.

Sample Code

```
1  static void reportGlobalObjects()  {
2      UdbEntity    *ents;
3      int          entsSize;
4      UdbReference *refs;
5      int          refsSize;
6      int         i,j;
7
8     udbListEntity(&ents, &entsSize);
9     udbListEntityFilter (ents, udbKindParse("global object ~static"),
10                         &ents, &entsSize);
11    printf ("\nGlobal Objects:\n");
12    for (i=0; i<entsSize; i++) {
13       printf (" %s \n", udbEntityNameLong(ents[i]) );
14       udbListReference(ents[i], &refs, &refsSize);
15       for (j=0; j<refsSize; j++) {
16          printf ("   %s: %s [%s (%d)] \n",
17               udbKindShortname(udbReferenceKind(refs[j])),
18               udbEntityNameShort(udbReferenceEntity(refs[j])),
19               udbEntityNameShort(udbReferenceFile(refs[j])),
20               udbReferenceLine(refs[j]) );
21       }
22       udbListReferenceFree(refs);
23    }
24    udbListEntityFree(ents);
25 }
```

Explanation of Sample Code

lines 2-3: Declare variables for list of entities and the list size. In this case, the list will be filtered to include only global objects.

lines 4-5: Declare variables for references for a particular entity, in this case a global object.

line 7: **Retrieve list of all entities**.

line 8-9: **Filter the entity list** to contain only global objects, replacing the original entity list with the filtered list.

line 10: Print the header.

line 11: **Loop through the list** of global objects.

lines 12: **Print the name** of the global object.

line 13: **Retrieve the references** for the current global object.

line 14: **Loop through all references** of the global object.

lines 15-19: **Print reference information** of the global object, including the kind of reference, the name of the entity being referenced, and the file and line number where the reference occurs.

line 21: **Free the Reference list**.

line 23: **Free the entity list**.

Sample Output

```
Global Objects:
 _TIFFerrorHandler
   Define: tif_msrc.c [tif_msrc.c (180)]
   Type: TIFFErrorHandler [tif_msrc.c (180)]
   Set: tif_msrc.c [tif_msrc.c (180)]
 PaintX_ProxyFileInfo
   Define: PaintX_p.c [PaintX_p.c (369)]
   Type: ExtendedProxyFileInfo [PaintX_p.c (369)]
   Set: PaintX_p.c [PaintX_p.c (369)]
 INTERNALBPP
   Define: PictureDecoder.cpp [PictureDecoder.cpp (7)]
   Set: PictureDecoder.cpp [PictureDecoder.cpp (7)]
   Use: LoadPicture [PictureDecoder.cpp (41)]
   Use: LoadResPicture [PictureDecoder.cpp (283)]
```

Report All Structs and their Member Types

Description This example reports all structs and their members.

Sample Code

```
1  static void reportAllStructs() {
2      UdbEntity    *structEnts;
3      int          structEntsSize;
4      UdbReference *refs;
5      int          refsSize;
6      int          i,j;
7      UdbStatus   status;
8
9     udbListEntity(&structEnts, &structEntsSize);
10    udbListEntityFilter (structEnts, udbKindParse("struct"),
11                         &structEnts, &structEntsSize);
12    for (i=0; i<structEntsSize; i++) {
13       printf ("\n %s \n", udbEntityNameLong(structEnts[i]) );
14       udbListReference(structEnts[i], &refs, &refsSize);
15       udbListReferenceFilter(refs,
16               udbKindParse("define, declare"), NULL,
17                1, &refs, &refsSize);
18       for (j=0; j<refsSize; j++) {
19           printf ("   %s %s \n",
20                udbEntityTypetext(udbReferenceEntity(refs[j])),
21                udbEntityNameShort(udbReferenceEntity(refs[j])) );
22       }
23       udbListReferenceFree(refs);
24    }
25    udbListEntityFree(structEnts);
26 }
```

Explanation of Sample Code

lines 2-3: Declare variables for a filtered list of entities and the list size. In this case, the list will be filtered to include only structs.

lines 4-5: Declare variables for references for a particular entity, in this case, a struct.

line 8: **Get the list of all project entities**.

lines 9-10: **Filter the entity list** to contain only structs, replacing the original list of entities, with the list containing only structs.

line 11: **Loop through all the structs.**

line 12: **Print the name of the struct**.

line 13: **Retrieve the references** for the current struct.

lines 14-16: **Filter the list of references** to include only “define” or “declare” references. The unique flag is also set to avoid

duplication in the case that a member is both defined and declared within the struct.

line 17: **Loop through the references**.

lines 18-20: **Print the type and the name of the struct** member being defined or declared.

line 22: **Free the list of references**.

line 24: **Free the list of entities**.

Sample Output

```
MacPattern
  BYTE [] pix

jpeg_source_mgr

OpDef
  char * name
  int len
  char * description

[unnamed]
  unsigned char * pData
  int FileSize
  int CurPos

IPictureDecoder
  virtual HRESULT LoadPicture
  virtual HRESULT LoadResPicture
```

Find All References To and From an Entity

Description This example reports all references used by a specified entity and also all references where a specified entity is used. This example is different from the others in that a specific entity is needed when calling the example function. Choosing the desired entity to report is left as an exercise for the user.

Sample Code

```
1  static void reportRefs(UdbEntity entity)  {
2     UdbReference   *refs;
3     int            refsSize;
4     UdbReference   *filterRefs;
5     int            filterRefsSize;
6     int            i,j;
7
8     udbListReference(entity, &refs, &refsSize);
9
10    udbListReferenceFilter(refs,
11       udbKindParse("call,declare,define,friend,use,include,modify,base,set,typed"),
12       NULL, 0, &filterRefs, &filterRefsSize);
13    printf ("\nWho (%s) %s references: \n",
14                udbKindShortname(udbEntityKind(entity)),
15                udbEntityNameLong(entity) );
16    for (j=0; j<filterRefsSize; j++) {
17       printf ("   %s: %s [%s (%d)] \n",
18                 udbKindShortname(udbReferenceKind(filterRefs[j])),
19                 udbEntityNameShort(udbReferenceEntity(filterRefs[j])),
20                 udbEntityNameShort(udbReferenceFile(filterRefs[j])),
21                 udbReferenceLine(filterRefs[j]) );
22       }
23    udbListReferenceFree(filterRefs);
24    udbListReferenceFilter(refs,
25        udbKindParse("callby,declarein,definein,friendby,useby,includeby,modifyby,
derive,setby,typedby"),
26        NULL,  0,  &filterRefs, &filterRefsSize);
27     printf ("\nWho references (%s) %s: \n",
28                udbKindShortname(udbEntityKind(entity)),
29                udbEntityNameLong(entity));
30     for (j=0; j<filterRefsSize; j++) {
31        printf ("   %s: %s [%s (%d)] \n",
32                     udbKindShortname(udbReferenceKind(filterRefs[j])),
33                     udbEntityNameShort(udbReferenceEntity(filterRefs[j])),
34                   udbEntityNameShort(udbReferenceFile(filterRefs[j])),
35                   udbReferenceLine(filterRefs[j]) );
36      }
37      udbListReferenceFree(filterRefs);
38      udbListReferenceFree(refs);
39 }
```

Explanation of Sample Code

lines 2-5: Declare variables for all references and the filtered list of references and their corresponding list sizes. The list of all references is retrieved once and then filtered repeatedly to obtain the list of references desired.

line 8: **Retrieve the list of all references** for the specified entity.

lines 10-12: **Filter the list of references to include 'forward' types of references**. What the specified entity calls, declares, defines, friends, uses, includes, modifies, derived from, sets, or types. In this example, the entity kindlist is NULL, indicating that we want these types of references to all kinds of entities. Alternatively, the entity kind could have been specified to include references only to functions, or classes, for example.

lines 13-15: Print header with kind and name of the entity.

line 16: **Loop through list of forward references**.

line 17-21: **Report forward reference info**; the kind of reference, the name of the entity being referenced, and the file and line where the reference occurs.

line 23: **Free the list of forward references**.

line 24-26: **Filter the list of references to include 'backward' types of references**. Where the specified entity is called by, declared in, defined in, is friend by, used by, included by, modified by, a base of, set by, or is typed by. In this example, the entity kind is NULL, indicating that we want these types of references to all kinds of entities. Alternatively, the entity kind could have been specified to include references only to functions, or classes, for example.

lines 27--29: Print header with kind and name of the entity.

line 30: **Loop through list of backward references**.

line 31-35: **Report backward reference info**; the kind of reference, the name of the entity being referenced, and the file and line where the reference occurs.

line 37: **Free the list of backward references**.

line 38: **Free the list of all references**.

Sample Output

```
Who (Function) AppendFilterSuffix references:
   Define: filter [DocMan.cpp (22)]
   Define: ofn [DocMan.cpp (22)]
   Define: pTemplate [DocMan.cpp (23)]
   Define: pstrDefaultExt [DocMan.cpp (23)]
```

```
      Use: pTemplate [DocMan.cpp (29)]
      Call: GetDocString [DocMan.cpp (29)]
      Use: filterExt [DocMan.cpp (29)]
      Call: IsEmpty [DocMan.cpp (30)]
      Use: pTemplate [DocMan.cpp (31)]
      Call: GetDocString [DocMan.cpp (31)]
      Use: filterName [DocMan.cpp (31)]
      Call: IsEmpty [DocMan.cpp (32)]
      Use: pstrDefaultExt [DocMan.cpp (38)]
      Use: LPCTSTR [DocMan.cpp (42)]
      Set: pstrDefaultExt [DocMan.cpp (42)]
      Inactive Use: pstrDefaultExt [DocMan.cpp (44)]
      Use: LPTSTR [DocMan.cpp (46)]
      Use: LPCTSTR [DocMan.cpp (46)]
      Use: pstrDefaultExt [DocMan.cpp (46)]
      Set: ofn [DocMan.cpp (46)]
      Set: lpstrDefExt [DocMan.cpp (46)]
      Use: ofn [DocMan.cpp (47)]
      Use: nMaxCustFilter [DocMan.cpp (47)]
      Set: ofn [DocMan.cpp (47)]
      Set: nFilterIndex [DocMan.cpp (47)]
      Modify: filter [DocMan.cpp (51)]
      Use: TCHAR [DocMan.cpp (53)]
      Modify: filter [DocMan.cpp (53)]
      Use: TCHAR [DocMan.cpp (55)]
      Modify: filter [DocMan.cpp (55)]
      Modify: filter [DocMan.cpp (57)]
      Use: TCHAR [DocMan.cpp (58)]
      Modify: filter [DocMan.cpp (58)]
      Use: ofn [DocMan.cpp (59)]
      Use: nMaxCustFilter [DocMan.cpp (59)]
      Modify: nMaxCustFilter [DocMan.cpp (59)]

   Who references (Function) AppendFilterSuffix:
      Define: DocMan.cpp [DocMan.cpp (22)]
      Call: DoPromptFileName [DocMan.cpp (82)]
      Call: DoPromptFileName [DocMan.cpp (92)]
```

Chapter 7 Entity and Reference Kinds

This chapter provides reference information about entity and reference kinds for each language and for both the Perl and C APIs.

This chapter contains the following sections:

Section	Page
Kind Name Usage	7–2
Ada Entity Kinds	7–6
Ada Reference Kinds	7–19
C/C++ Entity Kinds	7–39
C/C++ Reference Kinds	7–52
FORTRAN Entity Kinds	7–62
FORTRAN Reference Kinds	7–68
Java Entity Kinds	7–77
Java Reference Kinds	7–87
JOVIAL Entity Kinds	7–96
JOVIAL Reference Kinds	7–106
Pascal Entity Kinds	7–116
Pascal Reference Kinds	7–127

Kind Name Usage

Entity and reference kind names are used with both the Perl and C APIs.

Using Kinds in the Perl API

A number of methods provided with the Perl API deal with entity and reference kinds.

The $ent->kind() method returns an object of the Understand::Kind class. The following methods provided by that class allow you to obtain information about a kind:

- **$kind->check()** — Checks to see if the kind matches a kindstring filter.
- **$kind->inv()** — Returns the inverse of a reference kind.
- **$kind->longname()** — Returns the long form of the kind name.
- **$kind->name()** — Returns the name of the kind as a string. You can also use $ent->kindname() to get the name of a kind as a string.

Methods that accept a $kindstring or $entkindstring argument allow you to filter the results using an entity filter. The following methods accept such arguments:

- **$db->ents()** — Return a list of entities that match a kind.
- **$db->lookup()** — Return a list of entities that match a name and kind.
- **$ent->ents()** — Returns a list of entities that reference, or are referenced by, the entity.
- **$ent->filerefs()** — Returns a list of all references that occur in the specified file entity.
- **$ent->refs()** — Returns a list of references.
- **$ent->ref()** — Returns the first reference for the entity.
- **Metric::list()** — Returns a list of metric names for the specified kinds.

Methods that accept a $refkindstring argument allow you to filter the results using a reference filter. The following methods accept such a argument:

- **$ent->comments()** — Returns a formatted string based on the comments associated with an entity.
- **$ent->ents()** — Returns a list of entities that reference, or are referenced by, the entity.
- **$ent->filerefs()** — Returns a list of all references that occur in the specified file entity.
- **$ent->refs()** — Returns a list of references.
- **$ent->ref()** — Returns the first reference for the entity.

The following methods allow you to test kindname filters:

- **Kind::list_entity()** — Returns a list of entity longname kinds that match the $entkind filter.
- **Kind::list_reference()** — Returns a list of reference longname kinds that match the $refkind filter.

Using Kinds in the C API

The C API provides a number of functions that work with kinds and kind filters. For details, see Chapter 5, “C API Functions”. For example, some of these functions are:

- udbEntityKind returns the kind of the specified entity. For example:

```
if (udbIsKind(udbEntityKind(entity),"c abstract class type") {
    ...
}
```

- udbReferenceKind returns the kind of the specified reference.
- udbIsKind returns true if two kinds match.
- udbKindInverse returns the inverse kind of the specified kind.
- udbKindParse returns a kindlist of all kinds that match the specified text.
- udbKindList add a kind to the specified kindlist.
- udbKindLocate returns true if the specified kind is in the kind list.
- udbListEntityFilter returns a list of entities using the kind filter specified.
- udbListReferenceFilter returns a list of references using the kind filter specified.

Kind Name Filters

Kind filters are conceptually fairly simple, and in practice are also fairly easy to use. However, there are many details involved that can make documenting them daunting.

There are approximately 75 to 150 different defined kinds of entities and references, depending on the language. Some concepts of kind are simple to describe in some languages. For example, there is a single kind that represents file entities in Ada. However, in C++ there are three different kinds that represent different kinds of files. (Actually, there is a fourth kind, but it is used internally only and should not occur in usage of this API).

Each distinct kind is represented by a string of tokens that together read something like a sentence. A Kind string always has a token representing the language (C, Ada, Fortran or Java) and one or more tokens which describe the kind. The tokens have been chosen to be common, when appropriate, among several similar kinds. For example, in C++, the three kinds of files are “C Code File”, “C Header File” and “C Unknown Header File”. Notice how the token “File” is common to all three kinds and the token “Header” is common to two of the kinds? This is very important when specifying a filter.

A filter string is used to match one or more related or unrelated kinds, for purposes of selecting entities or references from the database. In order for a filter string to match a kind, each token in the filter string must be present as a token in the kind string. This can be thought of as an “and” relationship. For example, the filter “File” matches all three C file kinds, since all three have the token “File” in their strings. The filter “Header File” matches the two C file kinds that have both “Header” and “File” in their strings.

A filter string may use the symbol “~” to indicate the absence of a token. So, again for example, the filter string “File ~Unknown” matches the two C file kinds that have the token “File” in their string but do not have the token “Unknown” in their string.

In addition to “and” filters, “or” filters can also be constructed with the “,” separator. Groups of tokens separated by a comma are essentially treated as different filters. When each filter is calculated the results are combined with duplicates discarded. So, the filter string “Code File, Header File” matches all three of the C file kinds.

With proper knowledge of all the kinds available, kind filters can provide a powerful mechanism for selecting entities and references.

On the one hand, specifying “File” matches all file kinds; on the other hand, “Undefined” matches undefined files in addition to all other entity kinds that represent the concept “undefined”.

The following sections detail the name strings (used in the Perl API) and their respective C API literals for both Entity Kinds and Reference Kinds of each language.

Examples

Example 1: “c file ~unresolved” translates to c files that are not unresolved.

This results in a match with names of: C Code File, C Header File, and C Unknown Header File

Example 2: “ada function ~generic” translates to ada functions except those that are generic. This results in a match with names of:

```
Ada Abstract Function
Ada Abstract Function Local
Ada Abstract Function Operator
Ada Abstract Function Operator Local
Ada Function
Ada Function External
Ada Function Local
Ada Function Operator
Ada Function Operator Local
```

Example 3: Kind names may also specify a list of different kinds, by separating them with commas, as shown in this example.

“c typedef ~member ~unresolved, c object ~member” translates to c typedefs that are neither members nor unresolved, plus any c objects that are not members. This results in a match with names of:

```
C Object Global
C Object Global Static
C Object Local
C Object Local Static
C Typedef Type
C Unknown Object
C Unresolved Object Global
C Unresolved Object Global Static
```

Ada Entity Kinds

This section lists the general categories of Ada entity kinds and the specific kind Perl and C API names associated with them.

About "Local" Kinds

"Local" as part of an entity kind indicates that the entity was declared inside a package or task body, within a subroutine or block, or within the private part of a package specification or protected declaration.

Conversely, entities that do not have "local" as part of the entity kind are entities that are library units, or are declared within the visible part of a package or protected declaration, or within a task declaration.

Note: Some entity kinds do not have both a "Local" and non-local version. For example, there are no Local private types since the private declaration must appear in the visible part of a package specification.

About Derived Types and SubTypes

Derived types and subtypes are given an entity kind that corresponds to the entity kind of the root type. For example:

```
type color is (red, orange, yellow, green, blue, violet);
type rainbow_color is new color;
type paint_color is new rainbow_color;
```

The kind for all three types will be Ada Enumeration.

Ada Component Kinds

Use "ada component" to match all Ada component kinds.

Kind Name
Ada Component
Ada Component Local
Ada Component Variant
Ada Component Variant Local
Ada Component Discriminant
Ada Component Discriminant Local

A component is a record component. It may be local or non-local. For example:

```
type rec_type is record
    component : integer;   -- Ada Component
end record;
```

A record component may be declared in a variant part. For example:

```
type rec_type(d : integer) is record
   case d is
      when 1 =>
         component : integer;   -- a 'variant component'
      when others =>
          null;
   end case;
end record;
```

A discriminant component. For example:

```
type rec_type(d : integer) is record
           -- d is an Ada Component Discriminant
   ...
end record;
```

Ada Constant Kinds

Use “ada constant” to match all Ada constant kinds.

Kind Name
Ada Constant Object
Ada Constant Object Local
Ada Constant Object Deferred

A constant is a constant object or named number. It may be local or non-local. For example:

```
const1 : constant integer := 5; -- Ada Constant Object
     or
const2 : constant := 5;           -- Ada Constant Object
```

A constant may have a deferred constant declaration.

```
package some_pack is
   const : constant integer;
private
   const : constant integer := 5; -- Ada Constant Deferred
end;
```

Ada Entry Kinds

Use “ada entry” to match all Ada entry kinds.

Kind Name
Ada Entry
Ada Entry Body

An entry is an entry for a task or protected unit or type.

```
task some_task is
    entry e;            -- Ada Entry
```

An entry body may be in a protected unit or type.

```
protected body some_prot_unit is
   entry e when true is   -- e is Ada Entry Body
   begin
      ...
   end;
end;
```

Ada Enumeration Literal Kinds

Use “ada enumeration Literal” to match all Ada enumeration literals.

Kind Name
Ada Enumeration Literal

This kind is an enumeration literal for an enumeration type.

```
type light_color is (green, yellow, red);
   -- all Ada Enumeration Literal
```

Ada Exception Kinds

Use “ada exception” to match all Ada exception kinds.

Kind Name
Ada Exception
Ada Exception Local
Ada Exception Object Local
Ada Exception Others

An exception is a declared exception and may local or non-local.

```
bad_value : exception;   -- Ada Exception
```

An exception object local is a choice parameter from an exception handler.

```
when the_error : bad_value =>
      -- the_error is Ada Exception Object Local
```

Ada File Kinds

Use “ada file” to match all Ada file kinds.

Kind Name
Ada File

Any file in an ada project.

Ada Function Kinds

Use “ada function” to match all Ada function kinds.

Kind Name
Ada Abstract Function
Ada Abstract Function Local
Ada Abstract Function Operator
Ada Abstract Function Operator Local
Ada Function
Ada Function External
Ada Function Local
Ada Function Operator
Ada Function Operator Local
Ada Generic Function
Ada Generic Function Local

A function may be local or non-local.

```
function sum (a, b : integer) return integer;
                   -- sum is Ada Function
```

A function may be generic.

```
generic
   type t is private;
 function sum (a, b : t) return t;
                   -- Ada Generic Function
```

A function may be abstract.

```
function Get return abs_type is abstract;
                    -- Ada Abstract Function
```

A function may have an operator symbol as the name.

```
function "+"(p1 : abs_type; p2 : abs_type)
  return abs_type is abstract;
                    -- Ada Abstract Function Operator
```

Ada Implicit Kinds

Use “ada implicit” to match all Ada implicit kinds.

Kind Name
Ada Implicit

An implicit function may be referenced by a rename.

```
package some_pack is
   type t is new integer;
end;

with some_pack;
package other_pack is
   function "=" (a, b : integer) return boolean renames
some_pack."=";
end;
```

In this example, an "ada implicit" entity is created for some_pack."=". Normally no entity is created for these implicit functions. They are only created if they are renamed.

Ada Object Kinds

Use “ada object” to match all Ada object kinds.

Kind Name
Ada Constant Object
Ada Constant Object Local
Ada Constant Object Deferred
Ada Exception Object Local
Ada Loop Object Local
Ada Object
Ada Object Local

Kind Name
Ada Protected Object
Ada Protected Object Local
Ada Task Object
Ada Task Object Local

An object is a non-constant object and may be local or non-local.

```
size : integer range 1..10 := 1;  -- Ada Object
```

An object may be a loop parameter and is local.

```
for j in 1..10 loop  -- j is Ada Loop Object Local
   ...
end loop;
```

A protected object is declared as an object of a protected type and may be local or non-local.

```
obj : some_protected_type;    -- Ada Protected Object
```

A task object is declared as an object of a task type.

```
obj : some_task_type;         -- Ada Task Object
```

Ada Package Kinds

Use “ada package” to match all Ada package kinds.

Kind Name
Ada Generic Package
Ada Generic Package Local
Ada Package
Ada Package Local

A non-generic package may be local or non-local.

```
package list_pkg is     -- Ada Package
   ...
end;
```

A local or non-local package may also be generic.

```
generic
   type t is private;
package set_pack is      -- Ada Generic Package
   ...
end;
```

Ada Parameter Kinds

Use “ada parameter” to match all Ada parameter kinds.

Kind Name
Ada Parameter

A parameter is a subroutine or entry parameter.

```
procedure some_proc (a, b : integer);
           -- a and b are both Ada Parameter
```

Ada Procedure Kinds

Use “ada procedure” to match all Ada procedure kinds.

Kind Name
Ada Abstract Procedure
Ada Abstract Procedure Local
Ada Generic Procedure
Ada Generic Procedure Local
Ada Procedure
Ada Procedure External
Ada Procedure Local

A non-generic procedure may be local or non-local.

```
procedure max (a, b : in integer; m : out integer);
          -- max is Ada Procedure
```

A local or non-local procedure may be generic.

```
generic
   type t is private;
procedure switch (a, b: in out t);
                          -- Ada Generic Procedure
```

A local or non-local procedure may be abstract.

```
procedure Put(val : in abs_type) is abstract;
                                 --Ada Abstract Procedure
```

Ada Protected Kinds

Use “ada protected” to match all Ada protected kinds.

Kind Name
Ada Protected
Ada Protected Local
Ada Protected Object
Ada Protected Object Local
Ada Protected Type
Ada Protected Type Local
Ada Protected Type Private
Ada Protected Type Limited Private

A protected kind of object is declared with a single protected unit declaration.

```
protected shared_data is
    ...
private
   data : integer;        -- Ada Protected
end;
```

A protected object is declared as an object of a protected type.

```
obj : some_protected_type;  -- Ada Protected Object
```

A type may be a protected type, either local or non-local.

```
protected type some_protected_type is
                                  -- Ada Protected Type
   ...
private
   ...
end;
```

A protected type may be declared as private.

```
package some_pack is
   type some_protected_type is limited private;
                    -- Ada Protected Type Limited Private
private
   protected type some_protected_type is
                    -- Ada Protected Type Private
      ...
   private
      ...
   end;
end;
```

Ada Task Kinds

Use “ada task” to match all Ada task kinds.

Kind Name
Ada Task
Ada Task Local
Ada Task Object
Ada Task Object Local
Ada Task Type
Ada Task Type Local
Ada Task Type Private
Ada Task Type Limited Private

A task is a task object declared with a single task unit declaration.

```
task buffer is                 -- Ada Task
   ...
end;
```

A task object is declared as an object of a task type.

```
obj : some_task_type;            -- Ada Task Object
```

A task type.

```
task type some_task_type is   -- Ada Task Type
   ...
private
   ...
end;
```

A task type may be declared as private.

```
package some_pack is
   type some_task_type is private;
private
   task type some_task_type is -- Ada Task Type Private
      ...
   private
      ...
   end;
end;
```

Ada Type Kinds

Use “ada type” to match all Ada type kinds.

Kind Name
Ada Abstract Tagged Type Record
Ada Abstract Tagged Type Record Local
Ada Abstract Tagged Type Record Private
Ada Abstract Tagged Type Record Limited Private
Ada Protected Type
Ada Protected Type Local
Ada Protected Type Private
Ada Protected Type Limited Private
Ada Tagged Type Record
Ada Tagged Type Record Local
Ada Tagged Type Record Private
Ada Tagged Type Record Limited Private
Ada Task Type
Ada Task Type Local
Ada Task Type Private
Ada Task Type Limited Private
Ada Type
Ada Type Local
Ada Type Access
Ada Type Access Local
Ada Type Access Private
Ada Type Access Private Limited
Ada Type Access Subprogram
Ada Type Access Subprogram Local
Ada Type Access Subprogram Private
Ada Type Access Subprogram Limited Private
Ada Type Enumeration
Ada Type Enumeration Local
Ada Type Enumeration Private
Ada Type Enumeration Limited Private

Kind Name
Ada Type Private
Ada Type Limited Private
Ada Type Record
Ada Type Record Local
Ada Type Record Private
Ada Type Record Limited Private

A normal ada type is any type or subtype that is not an access, enumeration, or record type. It may be local or non-local.

```
type some_int is range 1 .. 2000;   -- Ada Type
```

A private type is any type or subtype that is not an access, enumeration, or record type and that is declared as private.

```
package some_pack is
   type some_int is private;
private
   type some_int is range 1 .. 2000;
        -- Ada Type Private
end;
```

An access to an object type may be local or non-local.

```
type int_access is access integer;  -- Ada Type Access
```

An access to object type may be declared as private.

```
package some_pack is
   type int_access is private;
private
   type int_access is access integer;
                             -- Ada Type Access Private
end;
```

An access to a subprogram type may be local or non-local.

```
type proc_access is access procedure(a : integer);
                            -- Ada Type Access Subprogram
```

An access to a subprogram type may be declared as private.

```
package some_pack is
   type proc_access is private;
private
   type proc_access is access procedure(a : integer);
                -- Ada Type Access Subprogram Private
end;
```

A type may be an enumeration type.

```
type light_color is (green, yellow, red);
      -- light_color is of kind Ada Type Enumeration
```

An enumeration type may be declared as private.

```
package some_pack is
   type light_color is private;
private
   type light_color is (green, yellow, red);
     -- light_color is Ada Type Enumeration Private
end;
```

A record type is a non-abstract, non-tagged, record type. It may be local or non-local.

```
type rec_type is record          -- Ada Type Record
   component : integer;
end record;
```

A non-abstract, non-tagged, record type may be declared as private.

```
package some_pack is
   type rec_type is private;
private
    type rec_type is record  -- Ada Type Record Private
       component : integer;
   end record;
end;
```

A record type may be a non-abstract tagged type. It may be local or non-local.

```
type tagged_type is tagged null record;
                         -- Ada Tagged Type Record
```

A tagged type may be declared as private.

```
package some_pack is
   type tagged_priv_type is private;
private
    type tagged_priv_type is tagged null record;
                        -- Ada Tagged Type Record Private
 end;
```

An abstract tagged type may be local or non-local.

```
type abs_type is abstract tagged null record;
                        -- Ada Abstract Tagged Type Record
```

An abstract tagged type may be declared as private.

```
package some_pack is
   type abs_priv_type is limited private;
     -- Ada Abstract Tagged Type Record Private Limited
private
   type abs_priv_type is abstract tagged null record;
     -- Ada Abstract Tagged Type Record Private
end;
```

Ada Unknown Kinds

Use “ada unknown” to match all Ada unknown kinds.

Kind Name
Ada Unknown

An unknown entity is one that is used but not declared. Currently, "Unknown" entities are only created in cases where the missing entity is withed or renamed.

```
with unknown_pack;  -- this package is not in project
procedure some_proc is
...
```

Ada Unresolved Kinds

Use “ada unresolved” to match all Ada unresolved kinds.

Kind Name
Ada Unresolved
Ada Unresolved External Function
Ada Unresolved External Procedure

Ada Reference Kinds

This section lists the general categories of Ada reference kinds (and inverse relations) and the specific Perl and C API names associated with them.

Ada Abort and Abortby Kinds

Use “ada abort” to match all Ada abort reference kinds.

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Abort	some_proc	some_task
Ada Abortby	some_task	some_proc

These kinds indicate a task is aborted by another program unit.

```
procedure some_proc is
    task some_task is
       ...
    end;
    task body some_task is separate;
begin
    abort some_task;
end;
```

Ada AccessAttrTyped and AccessAttrTypedby Kinds

Use “ada accessattrtype” to match all these reference kinds.

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada AccessAttrTyped	function_y	a_f
Ada AccessAttrTypedby	a_f	function_y

These kinds indicate an access attribute was used on a subprogram to generate the identified access type.

```
package p is
   type a_f is access function(x : in integer) return integer;
   function function_y(x : integer) return integer;
end;

with p; use p;
procedure main is
  local : a_f;
begin
  local := function_y'access;
end;
```

Ada Association and Associationby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Association	main	param
Ada Associationby	param	main

Association and Associationby reference kinds indicate a reference to a formal parameter in a named parameter association.

```
procedure main is
begin
   some_proc(param => 5);
end;
```

Ada Call and Callby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Call	main	some_proc
Ada Callby	some_proc	main
Ada Call Indirect	main	some_proc
Ada Callby Indirect	some_proc	main
Ada Call Dispatch	other_proc	some_proc
Ada Callby Dispatch	some_proc	other_proc
Ada Call Dispatch Indirect	other_proc	some_proc
Ada Callby Dispatch Indirect	some_proc	other_proc
Ada Call Ptr	some_pack	some_proc
Ada Callby Ptr	some_proc	some_pack

Call and Callby reference kinds indicate an invocation of a subprogram or entry.

```
procedure main is
begin
   some_proc(5);
end;
```

Call Indirect and Callby Indirect indicate an invocation of a subprogram or entry made via a renaming declaration. A regular call relation is also created to the entity that is a rename.

```
package some_pack is
   procedure some_proc;
end;

with some_pack;
procedure main is
   procedure my_proc renames some_pack.some_proc;
begin
   my_proc;
end;
```

Call Dispatch and Callby Dispatch indicate a dispatching call of a subprogram or entry.

```
package some_pack is
   type some_type is tagged null record;
   procedure some_proc(p : some_type);
end;

with some_pack;
procedure other_proc(p : some_pack.some_type'class) is
begin
   some_pack.some_proc(p);
end;
```

Call Dispatch Indirect and Callby Dispatch Indirect indicate a dispatching invocation of a subprogram or entry made via a renaming declaration. A regular call dispatch relation is also created to the entity that is a rename.

Call Ptr and Callby Ptr indicate access to a subprogram has been taken so the subprogram may be called later via a subprogram access variable.

```
package some_pack is
  type access_proc_type is access procedure;
  procedure some_proc;
  access_obj : access_proc_type := some_proc'access;
                    // Ada Call Ptr
end;
```

Ada CallParamFormal and CallParamFormalfor Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada CallParamFormal	obj	param
Ada CallParamFormalfor	param	obj

Ada CallParamFormal and Ada CallParamFormalfor indicate an object is used as an actual parameter for the indicated formal parameter.

```
package p1 is
   procedure some_proc(param : integer);
end;

with p1;
procedure p2 is
   obj : integer := 5;
begin
   p1.some_proc(obj);
end;
```

Ada Child and Parent Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Child Libunit	parent_pack	child_pack
Ada Parent Libunit	child_pack	parent_pack

Child and Parent Libunits references indicate that an entity is a child library unit of another entity.

```
package parent_pack is
    ...
end;
package parent_pack.child_pack is
   ...
end;
```

Ada Declare and Declarein Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Declare	some_pack	some_obj
Ada Declarein	some_obj	some_pack
Ada Declare Body	some_pack	some_proc
Ada Declarein Body	some_proc	some_pack
Ada Declare Body File	some_file	some_pack
Ada Declarein Body File	some_pack	some_file
Ada Declare Formal	gen_pack	item
Ada Declarein Formal	item	gen_pack
Ada Declare Incomplete	some_pack	item
Ada Declarein Incomplete	item	some_pack
Ada Declare Instance	some_proc	my_pack
Ada Declarein Instance	my_pack	some_proc
Ada Declare Instance File	some_file	my_pack
Ada Declarein Instance File	my_pack	some_file
Ada Declare Private	line 2: some_pack	line 2: t
Ada Declarein Private	line 2: t	line 2: some_pack
Ada Declare Spec	some_pack	some_proc
Ada Declarein Spec	some_proc	some_pack
Ada Declare Spec File	some_pack	some_proc
Ada Declarein Spec File	some_proc	some_pack
Ada Declare Stub	some_pack and some_proc	some_proc and param1
Ada Declarein Stub	some_proc and param1	some_pack and some_proc

Declare and Declarein reference kinds indicate that an entity declares a non program unit entity.

```
package some_pack is
   some_obj : integer;
end;
```

Declare Body and Declarein Body reference kinds indicate that an entity declares a program unit body or a parameter in a subprogram body.

```
package body some_pack is
   procedure some_proc is
   begin
      null;
   end;
end;
```

Declare Body File and Declarein Body File indicate a file declares a program unit body. Note that for separate subunits, there will be a Declare Body relation from the parent unit to the subunit and a Declare Body File relation from the file to the subunit. These relations represent the logical and physical declaration structures.

```
some_file contains
  package body some_pack is
end;
```

Declare Formal and Declarein Formal reference kinds indicate that an entity declares a generic formal parameter.

```
generic
   type item is private;
package gen_pack is
   ...
end;
```

Declare Incomplete and Declarein Incomplete reference kinds indicate that an entity declares an incomplete type.

```
package some_pack is
type item;
   ...
end;
```

Declare Instance and Declarein Instance reference kinds indicate that an entity declares an instance of a generic.

```
generic
package gen_pack is
   ...
end;

with gen_pack;
procedure some_proc is
   package my_pack is new gen_pack;
begin
   null;
end;
```

Declare Private and Declarein Private reference kinds indicate a declaration of a private type that occurs in the private part of a package specification.

```
 1   package some_pack is
 2      type t is private;
 3   private
 4      type t is new integer;
 5   end;
```

Declare Spec and Declarein Spec reference kinds indicate that an entity declares a program unit spec.

```
package some_pack is
   procedure some_proc;
end;
```

Declare Spec File and Declarein Spec File reference kinds indicate that a file entity declares a program unit spec.

```
some_file contains
   package some_pack is
   end;
```

Declare Stub and Declarein Stub reference kinds indicate that an entity declares a program unit stub or a parameter in a subprogram stub.

```
package body some_pack is
   procedure some_proc(param1 : integer) is separate;
end;
```

Ada Derive and Derivefrom Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Derive	some_type	derived_type
Ada Derivefrom	derived_type	some_type

Derive and Derivefrom reference kinds indicate the parent type for a derived type.

```
package some_pack is
   type some_type is range 1..10;
   type derived_type is new some_type;
end;
```

Ada Dot and Dotby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Dot	some_proc	some_pack
Ada Dotby	some_pack	some_proc

Dot and Dotby reference kinds indicate a reference to an entity as a prefix in an expanded name.

```
package some_pack is
   a : integer;
end;
with some_pack;
procedure some_proc is
begin
   some_pack.a := 5;
end;
```

Ada End and Endby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada End	some_pack	some_pack
Ada Endby	some_pack	some_pack
Ada End Unnamed	some_pack	some_pack
Ada Endby Unnamed	some_pack	some_pack

End and Endby reference kinds indicate a reference to an entity in its end statement.

```
package some_pack is
   ...
end some_pack;
```

End Unnamed and Endby Unnamed reference kinds indicate an end of a program unit entity where the end statement did not have a name reference

```
package some_pack is
   ...
end;
```

Ada Elaborate Body and Elaborate Bodyby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada ElaborateBody Implicit	some_pack	some_pack
Ada ElaborateBodyby Implicit	some_pack	some_pack
Ada ElaborateBody Ref	some_pack	other_pack
Ada ElaborateBodyby Refby	other_pack	some_pack

ElaborateBody Implicit and ElaborateBodyby Implicit reference kinds indicate a “pragma elaborate_body” applies to a program unit without the program unit name appearing in the pragma statement.

```
package some_pack is
   pragma elaborate_body;
end;
```

ElaborateBody Ref and ElaborateBodyby Refby reference kinds indicate a "pragma elaborate_body" applies to a program unit and the program unit name appears in the pragma statement.

```
with other_pack;
package some_pack is
   pragma elaborate_body(other_pack);
end;
```

Ada Handle and Handleby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Handle	some_proc	some_exception
Ada Handleby	some_exception	some_proc

Handle and Handleby reference kinds indicate a reference to an exception entity in an exception handler.

```
procedure some_proc is
begin
  ...
exception
   when some_exception =>
         ...
end;
```

Ada Instance and Instanceof Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Instance	my_pack	gen_pack
Ada Instanceof	gen_pack	my_pack
Ada Instance Copy	my_pack.nested_proc	gen_pack.nested_proc
Ada Instanceof Copy	gen_pack.nested_proc	my_pack.nested_proc
Ada InstanceActual	my_pack	integer
Ada InstanceActualfor	integer	my_pack
Ada InstanceParamFormal	integer	t
Ada InstanceParamFormalfor	t	integer

Instance and Instanceof references indicate that an instantiated entity is an instance of a generic entity.

```
generic
package gen_pack is
   ...
end;
with gen_pack;
procedure some_proc is
   package my_pack is new gen_pack;
begin
   ...
end;
```

Instance Copy and Instanceof Copy references indicate that an entity was created as a copy of an entity in a generic package.

At the point of an instantiation, a copy of each entity declared inside the generic is created. These entities are linked to the corresponding entity in the generic with an "instanceof copy" relation.

```
generic
package gen_pack is
   procedure nested_proc;
end;
with gen_pack;
procedure some_proc is
   package my_pack is new gen_pack;
begin
   my_pack.nested_proc;
end;
```

InstanceActual and InstanceActualfor references indicate that an entity was used as an actual parameter in a generic instantiation.

```
generic
   type t is (<>);
package gen_pack is
   ...
end;
with gen_pack;
package my_pack is new gen_pack(integer);
```

InstanceParamFormal and InstanceParamFormalfor references links generic formal and actual parameters.

```
generic
   type t is (<>);
package gen_pack is
   ...
end;
with gen_pack;
package my_pack is new gen_pack(integer);
```

Ada Operation and Operationfor Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Operation	some_type	some_proc
Ada Operationfor	some_proc	some_type

Operation and Operationfor references indicate that a subprogram is a primitive operation for the indicated type.

```
package some_pack is
   type some_type is tagged null record;
   procedure some_proc(x : some_type);
end;
```

Ada Override and Overrideby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Override	some_proc(x:other_type)	some_proc(x:some_type)
Ada Overrideby	some_proc(x:some_type)	some_proc(x:other_type)

Override and Overrideby references indicate that an operation subroutine (see Ada Operation and Operationfor Kinds), overrides another operation declared for the parent type.

```
package some_pack is
   type some_type is tagged null record;
   procedure some_proc(x : some_type);

   type other_type is new some_type with null record;
   procedure some_proc(x : other_type);
end;
```

Ada Raise and Raiseby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Raise	some_proc	some_exception
Ada Raiseby	some_exception	some_proc
Ada Raise Implicit	some_proc	some_exception
Ada Raiseby Implicit	some_exception	some_proc

Raise and Raiseby reference kinds indicate a program unit explicitly raised an exception.

```
procedure some_proc is
begin
   raise some_exception;
end;
```

Ada Ref and Refby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Ref	array_type	some_type
Ada Refby	some_type	array_type
Ada Import Ref	some_pack	some_proc
Ada Importby Refby	some_proc	some_pack

Ref and Refby reference kinds indicate a reference to an entity that does not fall under any other category.

All occurrences of an entity name have an associated reference. If the reference is not one of the other kinds described, it will be of type Ada Ref. Examples are the use of a type name in an array bounds declaration, and the use of an entity in a representation clause.

```
package some_pack is
   type some_type is 1..10;
   type array_type is array(some_type) of integer;
end;
```

Import Ref and Import Refby reference kinds indicate the subroutine is named in an import pragma.

```
package some_pack is
   procedure some_proc;
   pragma import(C, some_proc);
end;
```

Ada Rename and Renameby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Rename	sp	some_pack
Ada Renameby	some_pack	sp

Rename and Renameby reference kinds indicate an entity is a renaming declaration of another entity.

```
package some_pack is
   ...
end;
with some_pack;
procedure some_proc is
   package sp renames some_pack;
begin
   ...
end;
```

Ada Root and Rootin Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Root	some_file.ada	some_pack
Ada Rootin	some_pack	some_file.ada

Root and Rootin reference kinds indicate that an entity is a library unit or library unit body in a file. There is a "root" link from a file entity to each library unit and library unit body declared in the file.

```
file some_file.ada contains
   package some_pack is
      ...
   end;
```

Ada Separatefrom and Separate Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Separatefrom	some_proc	some_pack
Ada Separate	some_pack	some_proc

Separatefrom and Separate reference kinds indicate that a program unit entity is declared as separate from another program unit.

```
separate(some_pack)
procedure some_proc is
begin
  ...
end;
```

Ada Set and Setby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Set	some_proc	some_object
Ada Setby	some_object	some_proc
Ada Set Init	some_proc	some_object
Ada Setby Init	some_object	some_proc

Set and Setby reference kinds indicate that a program unit assigns a value to an object.

```
procedure some_proc is
   some_object : integer;
begin
   some_object := 0;
end;
```

Set Init and Setby Init reference kinds indicate that a program unit assigns a value to an object in an initialization.

```
procedure some_proc is
   some_object : integer := 0;
begin
   null;
end;
```

Ada Subtype and Ada Subtypefrom Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Subtype	some_type	some_subtype
Ada Subtypefrom	some_subtype	some_type

Subtype and Subtypefrom reference kinds indicate the parent type for a subtype.

```
package some_pack is
   type some_type is range 1..10;
   subtype some_subtype is some_type range 1..5;
end;
```

Ada Typed and Typedby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Typed	some_type	some_object
Ada Typedby	some_object	some_type
Ada Typed Implicit	integer	gen_func
Ada Typedby Implicit	my_func	integer
Ada Type Convert	some_proc	A_Form
Ada Type Convertby	A_Form	some_proc

Typed and Typedby reference kinds indicate that an object or component is of the specified type or that a function returns the specified type.

```
package some_pack is
   type some_type is range 1..10;
   some_object : some_type;
end;
```

Typed Implicit and Typedby Implicit reference kinds is used to specify the type of an entity where the entity type is not explicitly declared. Specifically, enumeration literals, types of instantiated functions, and loop parameters use this relation.

```
generic
   type t is private;
function gen_func return t;
with gen_func;
procedure some_proc is
   function my_func is new gen_func(integer);
begin
   ...
end;
```

Type Convert and Type Convertby reference kinds indicate that a routine uses a type in a type conversion.

```
procedure some_proc is
   type B_Form is new integer;
   type A_Form is new B_Form;
   X : A_Form;
   Y : B_Form;
begin
   X := A_Form(Y);
end;
```

Ada Use and Useby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Use	ex 1: some_proc	ex 1: some_object
	ex 2: some_proc	ex 2: some_task
Ada Useby	ex 1: some_object	ex 1: some_proc
	ex 2: some_task	ex 2: some_proc

Use and Useby reference kinds indicate the read of an object, or use of a task or protected object in an expanded name.

Example 1:

```
package some_pack is
   some_object : integer;
end;
with some_pack;
procedure some_proc is
   local_object : integer;
begin
   local_object := some_object;
end;
```

Example 2:

```
procedure some_proc is
   task some_task is
         entry e;
   end;
 begin
       some_task.e;
 end;
```

Ada Usepackage and Usepackageby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Usepackage	some_proc	some_pack
Ada Usepackageby	some_pack	some_proc

Usepackage and Usepackageby reference kinds indicate a reference to a package in a use clause.

```
package some_pack is
   ...
end;
with some_pack; use some_pack;
procedure some_proc is
begin
   ...
end;
```

Ada UseType and UseTypeby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada Usetype	some_proc	some_type
Ada Usetypeby	some_type	some_proc

Usetype and Usetypeby reference kinds indicate a reference to a type in a use type clause.

```
package some_pack is
   type some_type is range 1..10;
end;
with some_pack;
procedure some_proc is
   use type some_pack.some_type;
begin
   ...
end;
```

Ada With and Withby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
Ada With Body	some_proc	some_pack
Ada Withby Body	some_pack	some_proc
Ada With Spec	some_proc	some_pack
Ada Withby Spec	some_pack	some_proc
Ada With Needed Body	some_proc	some_pack
Ada Withby Needed Body	some_pack	some_proc
Ada With Needed Spec	some_proc	some_pack
Ada Withby Needed Spec	some_pack	some_proc
Ada Withaccess	some_proc	some_pack
Ada Withaccessby	some_pack	some_proc

Relations ending with Body are for withs from program unit bodies. Relations ending with Spec are for withs from program unit specs.

With and Withby reference kinds indicate a reference to a program unit in with clause, when the with is not actually needed. In cases where the with is needed, the reference kind is Ada With Needed.

```
package some_pack is
   type some_type is range 1..10;
end;
with some_pack;
procedure some_proc is
begin
   null;
end;
```

WithNeeded and WithbyNeeded indicate a reference to a program unit in with clause, when the with is needed. In cases where the with is not actually needed, the reference kind is Ada With.

```
package some_pack is
   type some_type is range 1..10;
end;
with some_pack;
procedure some_proc is
   some_obj : some_pack.some_type;  -- with of some_pack
is needed
begin
   null;
end;
```

Withaccess and Withaccessby reference kinds indicate that a program unit is utilizing a with link to a program unit that was not directly withed it. Typically, with links are either "With" or "WithNeeded".

These relations indicate what program unit is withed and whether or not the with clause was actually necessary. There are times however, when a compilation unit may access a withed unit even though it does not itself contain a with clause for the unit. This may occur in subunits, and in child library units. In these cases, a withaccess relation to the accessed program unit is generated.

```
package some_pack is
   type some_type is range 1..10;
end;
with some_pack; use some_pack;
procedure parent_proc is
   procedure some_proc is separate;
begin
   null;
end;
separate(parent_proc)
procedure some_proc is
   x : some_type;
begin
   null;
end;
```

C/C++ Entity Kinds

This section lists the general categories of C entity kinds and the specific names associated with them for use in the Perl and C APIs.

C Class Kinds

Use “c class” to match all C Class entity kinds.

Kind Name
C Abstract Class Type
C Class Type
C Private Member Class Type
C Protected Member Class Type
C Public Member Class Type
C Unknown Class Type
C Unnamed Class Type
C Unnamed Private Member Class Type
C Unnamed Protected Member Class Type
C Unnamed Public Member Class Type
C Unresolved Class Type
C Unresolved Private Member Class Type
C Unresolved Protected Member Class Type
C Unresolved Public Member Class Type
C Unresolved Unnamed Class Type

An ordinary class is a class that is not a member in another class, and is not unnamed or unresolved.

```
class class_normal {                    // C Class Type
  int mem1;
};
```

An abstract class is a class with at least one pure virtual member function.

```
class c {                       // C Abstract Class Type
  int virtual func1()=0;
};
```

A class may itself be a member of another class. In this case it may be private, protected, or public.

```
class c {
  private: class c_private {};
                          // C Private Member Class Type
};
```

A member class may be unnamed.

```
class A {
  class {                 // C Unnamed Class Type
    int a;
    } b;
};
```

An unresolved class is a class that is known to exist but who's definition is unknown. An unknown class is an unresolved class, but is also lacking any formal declaration.

```
class c_unresolved var_1;       // C Unresolved Class
class c_unknown    var_2;       // C Unknown Class
```

See Also: Enum Kinds, Function Kinds, Object Kinds, Struct Kinds, Typedef Kinds, and Union Kinds for entity kinds of Class Members.

C Enum Kinds

Use "c enum" to match all C Enum Type kinds

Kind Name
C Enum Type
C Private Member Enum Type
C Protected Member Enum Type
C Public Member Enum Type
C Unknown Enum Type
C Unnamed Enum Type
C Unnamed Private Member Enum Type
C Unnamed Protected Member Enum Type
C Unnamed Public Member Enum Type
C Unresolved Enum Type
C Unresolved Private Member Enum Type
C Unresolved Protected Member Enum Type
C Unresolved Public Member Enum Type

An ordinary enum type is an enumerated data type which is not a member of a class.

```
enum etype1 { val1, val2 };    // C Enum Type
```

An enum type may be a member of a class. In this case it may be private, protected, or public.

```
class c {
  protected: enum etype1 {val1, val2};
                          // C Protected Member Enum Type
};
```

A member or non-member enum type may be unnamed.

```
enum { val1, val2 } var_1;    // C Unnamed Enum Type
```

An unresolved enum type is an enumerator type that is that is referenced but for which no definition has been found. This can happen when an include file isn't found. An unknown enum type had neither a definition nor a declaration.

```
enum etype1;

etype1    var_1;      // C Unresolved Enum Type
unk_type  var_2;      // C Unknown Enum Type
```

See Also: Enumerator Kinds for the enumerator values of enumerated data types.

C Enumerator Kinds

Use “c enumerator” to match all C Enumerator kinds

Kind Name
C Enumerator
C Unresolved Enumerator

An enumerator is the identifier assigned a value in an enum type.

```
enum etype {
   e_val_1, e_val_2, e_val_3     // C Enumerator
};
```

An unresolved enumerator is an enumerator identifier that belongs to an enum type that is declared in an included file which is not part of the project.

```
#include “enumTypes.h”      // C Unknown Header File
...
etype1  var_1;
var_1 = e_val_2;         // C Unresolved Enumerator
```

See Also: Enum Kinds for enumerator type kinds.

C File Kinds

Use “c file” to match all C File kinds

Kind Name
C Code File
C Header File
C Unknown Header File
C Unresolved Header File

Code files are project files which are not included by any other files (typically *.c, *.cpp). Header files are project files which are included by at least one other file (typically *.h, *.hpp).

An unknown header file is a header file that has been included by another file but cannot be located.

```
#include "unk_file.h"            // C Unknown Header File
```

An unresolved header file is an intermediate entity kind used by the parser and should never be visible at the API level. Any unresolved header file will either become a known header file (if the file can be located and parsed) or an unknown header file (if the file cannot be located) before the completion of the analysis run.

C Function Kinds

Use “c function” to match all C Function kinds

Kind Name
C Function
C Function Static
C Private Member Const Function
C Private Member Const Function Virtual
C Private Member Const Function Virtual Pure
C Private Member Function
C Private Member Function Static
C Private Member Function Virtual
C Private Member Function Virtual Pure
C Protected Member Const Function
C Protected Member Const Function Virtual
C Protected Member Const Function Virtual Pure

Kind Name
C Protected Member Function
C Protected Member Function Static
C Protected Member Function Virtual
C Protected Member Function Virtual Pure
C Public Member Const Function
C Public Member Const Function Virtual
C Public Member Const Function Virtual Pure
C Public Member Function
C Public Member Function Static
C Public Member Function Virtual
C Public Member Function Virtual Pure
C Unknown Function
C Unknown Member Function
C Unresolved Function
C Unresolved Function Static
C Unresolved Private Member Const Function
C Unresolved Private Member Const Function Virtual
C Unresolved Private Member Const Function Virtual Pure
C Unresolved Private Member Function
C Unresolved Private Member Function Static
C Unresolved Private Member Function Virtual
C Unresolved Private Member Function Virtual Pure
C Unresolved Protected Member Const Function
C Unresolved Protected Member Const Function Virtual
C Unresolved Protected Member Const Function Virtual Pure
C Unresolved Protected Member Function
C Unresolved Protected Member Function Static
C Unresolved Protected Member Function Virtual
C Protected Member Function Virtual Pure
C Unresolved Public Member Const Function
C Unresolved Public Member Const Function Virtual
C Unresolved Public Member Const Function Virtual Pure
C Unresolved Public Member Function

Kind Name
C Unresolved Public Member Function Implicit
C Unresolved Public Member Function Static
C Unresolved Public Member Function Virtual
C Unresolved Public Member Function Virtual Pure

A function may be an ordinary function that is not a static function and not a member of a class.

```
int func1(void);                    // C Function
```

A non-member function that is declared static can only be referenced from within the same file that contains the function.

```
static int func1 (void) { }      // C Function Static
```

A function may be a member of a class. In this case the member function may be private, protected, or public.

```
class c {
  public:
    void func1 (void) {...}; // C Public Member Function
};
```

A member function declared as static exists once for the whole class, not in each class instance.

```
class c {
  private:
    static void func1 () {...};
                      // C Private Member Function Static
};
```

A member function may also be declared const, which indicates that it will not alter the state of the object on which it is invoked.

```
class c {
  protected:
    int func1 () const; // C Protected Member Const Function
};
```

A pure virtual member function is a virtual member function that does not have an implementation. A pure virtual function cannot be called. It must be overridden in a derived class in order to be used.

```
class c {
 public:
   virtual int foobar() = 0; // C Public Member Function Pure
};
```

An unresolved function or unresolved member function is a function that is known to exist but who's definition is unknown. Typically this occurs when the function is defined in a file that is not part of the project. An unknown function is an unresolved function that is also lacking any formal declaration.

```
void func1(int);

int func() {
   func1(1);         // C Unresolved Function
   func2(1);         // C Unknown Function
}
```

C Macro Kinds

Use "c macro" to match all C Macro kinds

Kind Name
C Inactive Macro
C Macro
C Macro Functional
C Macro Project
C Unknown Macro
C Unresolved Macro

A macro which is defined in and used in an active region of code is C Macro. For example:

```
#define MACRO_ACTIVE
```

An inactive macro appears in an inactive region of code.

```
#if 0
#ifdef MACRO_INACTIVE
#endif
#endif
```

An unresolved macro is one that is known to exist but who's definition is not available in the current scope. An unknown macro is a macro whose definition is not known. Typically this occurs when the macro is defined in an included file that is not part of the project.

```
#include "my_macros.h"      // C Unknown Header File
#ifdef  MY_MACRO            // C Unknown Macro
...
#endif
```

C Namespace Kinds

Use “c namespace” to match all C Namespace kinds

Kind Name
C Namespace

For example:

```
namespace MyNamespace {
   int object1;    // MyNamespace::object1
}
```

C Object Kinds

Use “c object” to match all C Object kinds

Kind Name
C Object Global
C Object Global Static
C Object Local
C Object Local Static
C Private Member Object
C Private Member Object Static
C Protected Member Object
C Protected Member Object Static
C Public Member Object
C Public Member Object Static
C Unknown Member Object
C Unknown Object
C Unresolved Object Global
C Unresolved Object Global Static
C Unresolved Private Member Object
C Unresolved Private Member Object Static
C Unresolved Protected Member Object
C Unresolved Protected Member Object Static
C Unresolved Public Member Object
C Unresolved Public Member Object Static

A global object is a variable which is not a member of a class, is not declared within a function, and is not declared as static.

```
int global_var;          // C Object Global
```

A global static object is a variable which is not a member of a class and is not declared within a function, but is declared static, and is therefore only accessible from the file in which it is defined.

```
static int static_global_var;  // C Object Global Static
```

A local object is a variable which is not a member of a class and is defined within a (non-member) function.

```
void func1() {
  int local_var_1;          // C Object Local
  static int local_var_2;   // C Object Local
}
```

An object may be a member of a class. In this case the object may be private, protected, or public.

```
class A {
  public:
    int obj1;    // C Public Member Object
};
```

An unresolved object is a variable which is known to exist, but who's definition is unknown. This typically occurs when a header file is not part of the project. An unknown object is a variable with no known definition or declaration.

```
extern var_1;

int func() {
   var_2 = var_1;     // var_1 is C Unresolved Object
                      // var_2 is C Unknown Object
}
```

C Parameter Kinds

Use "c parameter" to match all C Parameter kinds

Kind Name
C Parameter
C Unnamed Parameter

A parameter is a formal parameter to any function or member function.

```
extern int func_ext(int param_ext); // Not parameter entity
void func (int param_1) {};         // C Parameter
```

C Struct Kinds

Use “c struct” to match all C Struct kinds

Kind Name
C Abstract Struct Type
C Private Member Struct Type
C Protected Member Struct Type
C Public Member Struct Type
C Struct Type
C Unknown Struct Type
C Unnamed Private Member Struct Type
C Unnamed Protected Member Struct Type
C Unnamed Public Member Struct Type
C Unnamed Struct Type
C Unresolved Struct Type
C Unresolved Private Member Struct Type
C Unresolved Protected Member Struct Type
C Unresolved Public Member Struct Type
C Unresolved Unnamed Struct Type

A struct type may be an ordinary struct, that is, not a member of a class.

```
struct mystruct {          // C Struct Type
  int field1;
  int field2;
  };
```

A struct may be unnamed. The following example shows an array of 10 elements containing a structure consisting of two int members

```
struct {                   // C Unnamed Struct Type
   int field1;
   int field2;
}myarray[10];
```

A struct may be a member of a class. In this case the struct may be private, protected, or public.

```
class A {
  public:
    struct mystruct{       // C Public Member Struct Type
        int field1;
        int field2;
     };
}
```

A struct may be an abstract struct, which is a struct with at least one pure virtual member function.

```
struct struct_abstract {    // C Abstract Struct Type
  int virtual mem1() = 0;
};
```

An unresolved struct is a struct that is known to exist but who's definition is unknown. An unknown struct is an unresolved struct, but is also lacking any formal declaration.

```
extern struct a_struct;     // C Unresolved Struct Type
typedef struct b_struct T;  // C Unknown Struct
```

C Typedef Kinds

Use "c typedef" to match all C Typedef kinds.

Kind Name
C Private Member Typedef Type
C Protected Member Typedef Type
C Public Member Typedef Type
C Typedef Type
C Unknown Type
C Unknown Member Type
C Unresolved Private Member Typedef Type
C Unresolved Protected Member Typedef Type
C Unresolved Public Member Typedef Type
C Unresolved Typedef Type

A typedef is used to assign an alternate name to a data type.

```
typedef   int   COUNTER;    // C Typedef Type
```

A typedef may be a member of a class. In this case the typedef may be private, protected, or public.

```
class A {
  public:
    typedef int COUNTER; // C Public Member Typedef Type
};
```

An unresolved typedef is a typedef which is known to exist, but who's definition is unknown. This typically occurs when a header file is not part of the project. An unknown type is some type (typedef, class, etc.) whose definition and declaration is not found.

```
#include "my_type2.h"  // C Unresolved Header File;
mytype2 var;  // C Unresolved Typedef Type;
         // mytype2 is defined in unresolved header file

int func (mytype1 tvar) {   //mytype1 is C Unknown Type
  ...
}
```

C Union Kinds

Use "c union" to match all C Union kinds.

Kind Name
C Private Member Union Type
C Protected Member Union Type
C Public Member Union Type
C Union Type
C Unknown Union Type
C Unnamed Private Member Union Type
C Unnamed Protected Member Union Type
C Unnamed Public Member Union Type
C Unnamed Union Type
C Unresolved Private Member Union Type
C Unresolved Protected Member Union Type
C Unresolved Public Member Union Type
C Unresolved Union Type
C Unresolved Unnamed Union Type

A union allows storage of different types of data into the same storage area.

```
union anyNumber{           // C Union Type
   int i;
   float f;
};
```

A union may be a member of a class. In this case the union may be private, protected, or public.

```
class A {
  public:
    union anyNumber{     // C Public Member Union Type
     int i;
     float f;
    };
}
```

A union or member union may be unnamed.

```
union {                  // C Unnamed Union Type
   int i;
   float f;
} mydata;
```

An unresolved union is a union which is known to exist, but who's definition is unknown. This typically occurs when a header file is not part of the project. An unknown union is a union whose definition and declaration is not found.

```
#include "my_unions.h" // C Unresolved Header File;
union my_union data;  // C Unresolved Union Type
        // my_union is defined in unresolved header file
union unknown_union more_data;  //unknown_union is not
                  // defined and is C Unknown Type
```

C/C++ Reference Kinds

This section lists the general categories of C/C++ reference kinds (and their inverse relations).

The following sections provide details for related groups of C/C++ reference kinds. Each grouping includes both "forward" and "backward" references (call and callby, for example).

Each group of reference kinds is presented in a table format, showing the names for use in the Perl and C APIs and the entity performing the reference (scope) and the entity being referenced. The scope and entity noted are the entities returned by calls to *udbReferenceScope()* and *udbReferenceEntity()*, respectively.

C Base and Derive Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
C Private Base	D	B
C Private Derive	B	D
C Protected Base	D	B
C Protected Derive	B	D
C Public Base	D	B
C Public Derive	B	D
C Virtual Private Base	D	V
C Virtual Private Derive	V	D
C Virtual Protected Base	D	V
C Virtual Protected Derive	V	D
C Virtual Public Base	D	V
C Virtual Public Derive	V	D

Base and Derive reference kinds may be public, protected, or private. Only public code samples are shown here. The following code sample illustrates class D which is derived from base class B. The public declaration indicates that all public functionality of class B can be accessed from class D.

```
class D : public B{
...
};
```

A base class may also be virtual. Again, only public derivation is shown here. V is the virtual base class from which D is derived.

```
class  V {
  public:
    virtual void f();
    virtual void g();
};

class D : virtual public V{
  public:
    virtual void g();
};
```

C Call and Callby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
C Asm Call	func	func1
C Asm Callby	func1	func
C Call	func	func1
C Callby	func1	func
C Implicit Call	func	func1
C Implicit Callby	func1	func
C Inactive Call	func	func1
C Inactive Callby	func1	func
C Virtual Call	func	func1
C Virtual Callby	func1	func

Asm Call and Asm Callby indicates a reference in assembly code to a known C/C++ function. The function's name must already be declared at the point of the usage.

```
extern int func1();
int func() {
   _asm op1 _func1;
   _asm op1 _func2;
}
```

Call and Callby reference kinds indicate a reference to a known C/C++ function. The function’s name must already be declared at the point of the usage.

```
extern int func1(void);
int func() {
   func1();
   func2();
}
```

Inactive Call or Inactive Callby reference kinds indicate a reference to a known C/C++ function in an inactive region of code.

```
int func() {
#if 0
   func1();
#endif
}
```

C Declare and Declarein Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
C Declare	func	i
C Declarein	file.c	ext_func
C Declare Implicit	func	i
C Declarein Implicit	file.c	ext_func

Declare and Declarein reference kinds indicate the declaration of an entity.

```
extern int ext_func( char ) ;
int func (void) {
  int i;
  ...
}
```

C Define and Definein Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
C Define	ifile.c	func
	func	c
	func	i
C Definein	func	file.c
	c	func
	i	func

Define and Definein reference kinds indicate the definition of an entity.

```
int func (char c) {
  int i;
  ...
}
```

C End and Endby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
C End	func	func
C Endby	func	func

These reference kinds mark the end point of functions, classes, enums, and namespaces. Each entity references itself.

```
int func (char c) {
  int i;
}
```

C Exception Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
C Exception Allow	func	Type1
C Exception Allowby	Type1	func
C Exception Catch	-- for future use --	-- for future use --
C Exception Catchby	-- for future use --	-- for future use --
C Exception Throw	-- for future use --	-- for future use --
C Exception Throwby	-- for future use --	-- for future use --

The Exception Allow and Exception Allowby reference kinds are references in function declarations or definitions to the allowed exceptions the function may throw. For example:

```
void func() throw(int, Type1);
```

C Friend and Friendby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
C Friend	C	F
C Friendby	F	C

Friend and Friendby reference kinds indicate the granting of friendship to a class or member function.

A friend class:

```
class C {
  public:
  private:
     friend class F;
};
```

A friend function:

```
class F {
  public:
   void f();
};

class C {
```

```
  public:
  private:
    friend void F::f();
};
```

C Include and Includeby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
C Include	file.c	my_includes.h
C Includeby	my_includes.h	file.c
C Implicit Include	file.c	my_includes.h
C Implicit Includeby	my_includes.h	file.c

Include and Includeby references indicate a reference to an include file.

```
#include "my_includes.h"
```

C Modify and Modifyby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
C Deref Modify	func	i
C Deref Modifyby	i	func
C Modify	func	i
C Modifyby	i	func

A Modify or Modifyby indicates a reference where a variable is modified without an explicit assignment statement. The variable is both used and set at the same reference location.

```
int func(int i) {
  i++;             // modify
  ...}
```

Deref Modify and Deref Modifyby indicate a reference in which a variable is dereferenced. For example:

```
int *a,*b=0;
++*a;    // deref modify of a
```

C Overrides and Overriddenby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
C Overrides	B:func	A:func
C Overriddenby	A:func	B:func

These reference kids indicate when a method in one class overrides a virtual method in a base class. For example:

```
class A {
   public: virtual void func();
};
class B: public A {
   public: void func();
};
```

C Set and Setby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
C Set	func, func	i, j
C Setby	i, j	func, func
C Deref Set	func, func	i, j
C Deref Setby	i, j	func, func
C Set Init	func, func	i, j
C Setby Init	i, j	func, func
C Set Init Implicit	func, func	i, j
C Setby Init Implicit	i, j	func, func

A Set or Setby reference indicates any explicit assignment of a variable.

```
int func(int i) {
  int j;
  i = i + 1;       // i is both use and set
  j = i;           // j is set, i is use
}
```

Deref Set and Deref Setby indicate a reference in which a variable is dereferenced. For example:

```
int *a,*b=0;
*a = *b;  // Deref Set of a, Deref Use of b
```

C Typed and Typedby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
C Typed	b	BYTE
C Typedby	BYTE	b
C Typed Implicit	b	BYTE
C Typedby Implicit	BYTE	b

A Typed or Typedby reference indicates a reference to a known typedef variable.

```
typedef unsigned char   BYTE;
BYTE b;
```

C Use and Useby Kinds

Kind Name	Entity Performing Reference	Entity Being Referenced
C Asm Use	func	var1
C Asm Useby	var1	func
C Deref Use	b	a
C Deref Useby	a	b
C Inactive Use	func	var1
C Inactive Useby	var1	func
C Use	func	var1
C Useby	var1	func
C Use Macrodefine	containing file	func1
C Useby Macrodefine	func1	containing file
C Use Macroexpand	func	var1
C Useby Macroexpand	var1	func
C Use Ptr	main_func	funcCB
C Useby Ptr	funcCB	main_func
C Use Return	func	var1
C Useby Return	var1	func

An ordinary Use or Useby indicates a reference in an active region of (non-assembler) code to a known C/C++ variable. The variable's name must already be declared at the point of the usage.

```
extern int var1;
int func() {
   int local_var;
   local_var = var1;  // use of var1
}
```

Asm Use and Asm Useby indicates a reference in assembly code to a known C/C++ variable. The variable's name must already be declared at the point of the usage.

```
extern int var1;
int func() {
   _asm op1 _var1;   // var1 is known; use
   _asm op1 _var2;   // var2 is not known; not a use
}
```

Deref Use and Deref Useby indicate a reference in which a variable is dereferenced. For example:

```
int *a,*b=0;
*a = *b;  // Deref Set of a, Deref Use of b
```

Inactive Use and Inactive Useby indicates a reference in an inactive region of code to a known C/C++ variable. The variable's name must already be declared at the point of the usage.

```
extern int var1;
int func() {
   int local_var;
#if 0
   local_var = var1;  // inactive use of var1
#endif
}
```

Use Macrodefine and Useby Macrodefine indicate a reference to a known entity in a macro definition. For example:

```
int func1();
#define MACRO (func1())
```

A Use Ptr or Useby Ptr indicates a reference to a known function pointer.

```
extern int funcCB (int);
static void func2( int(*)() );

void func2(int(*)() cb) {
  (void)cb(1);
  return;
}
int main_func() {
   func2(funcCB);   // both Use Ptr and Useby Ptr
}
```

FORTRAN Entity Kinds

This section lists the general categories of FORTRAN entity kinds, the kind names for use with the Perl and C APIs. The following table lists kind names multiple times if they are part of multiple categories.

Category	Kind Name	See
Block	Fortran Block Data	page 7–63
	Fortran Block Variable	page 7–63
Common	Fortran Common	page 7–63
Datapool	Fortran Datapool	page 7–63
Dummy Argument	Fortran Dummy Argument	page 7–64
Entry	Fortran Entry	page 7–64
File	Fortran File	page 7–64
	Fortran Include File	page 7–64
	Fortran Unknown Include File	page 7–66
	Fortran Unresolved Include File	page 7–66
Function	Fortran Function	page 7–64
	Fortran Intrinsic Function	page 7–65
	Fortran Unresolved Function	page 7–66
Interface	Fortran Interface	page 7–64
Pointer Block	Fortran Pointer	page 7–65
Main	Fortran Main Program	page 7–65
Module	Fortran Module	page 7–65
	Fortran Unknown Module	page 7–66
Subroutine	Fortran Intrinsic Subroutine	page 7–65
	Fortran Subroutine	page 7–66
	Fortran Unresolved Subroutine	page 7–66
Type	Fortran Derived Type	page 7–63
Unresolved	Fortran Unresolved Function	page 7–66
	Fortran Unresolved Include File	page 7–66
	Fortran Unresolved Subroutine	page 7–66
Variable	Fortran Block Variable	page 7–63
	Fortran Variable	page 7–67
	Fortran Namelist Variable	page 7–65

Fortran Block Data

A block data program unit. For example, A and B in the following example.

```
BLOCK DATA BLKDATA
   ...
END BLOCK DATA
```

Fortran Block Variable

A variable that is part of a common block or datapool. For example, BLKDATA in the following example.

```
COMMON /CBLK/ A, B
```

Fortran Common Block

A common block. If no name is given for the common block, a common block entity will be created with the name "(Unnamed_Common)". For example, CBLK in the first example and (Unnamed_Common) in the second example.

```
COMMON /CBLK/ A, B
COMMON A, B
```

Fortran Datapool

A datapool entity. This is an extension supported by some FORTRAN versions. If no name is given with the datapool statement, a datapool entity will be created with the name "(Unnamed_Datapool)". For example, DPOOL in the first example and (Unnamed_Datapool) in the second example.

```
DATAPOOL /DPOOL/ A, B
DATAPOOL A, B
```

Fortran Derived Type

A derived type. For example, DTYPE in the following example.

```
TYPE DTYPE
   INTEGER COMP1
   INTEGER COMP2
END TYPE DTYPE
```

Fortran Dummy Argument

A function, subroutine, or entry dummy argument. For example, A and B in the following example.

```
SURBOUTINE SUB(A, B)
...
END SUBROUTINE
```

Fortran Entry

A subprogram entry. For example, ENT in the following example.

```
FUNCTION FUNC() RESULT (R)
   ...
   ENTRY ENT() RESULT (R)
   ...
END FUNCTION
```

Fortran File

A FORTRAN source file.

Fortran Function

A FORTRAN function. For example, FUNC in the following example.

```
FUNCTION FUNC() RESULT (R)
   ...
END FUNCTION
```

Fortran Include File

A FORTRAN file which is included by another file with an INCLUDE line. For example, inc.com in the following example.

```
INCLUDE "inc.com"
```

Fortran Interface

A generic interface block. For example, F in the following example.

```
INTERFACE F
   ...
END INTERFACE
```

Fortran Intrinsic Function or Subroutine

A FORTRAN intrinsic function or subroutine.

The list of intrinsic procedures used depends on the version of FORTRAN (77, 90, or 95) and may by configured by the user to include intrinsic procedures in their fortran implementation. The standard lists of intrinsic procedures can be found in sti/conf/understand/fortran.

Fortran Main Program

A FORTRAN main program. If the PROGRAM statement is not given, a main program entity will still be created. The name will be "(Unnamed_Main)". For example, MAIN in the following example.

```
PROGRAM MAIN
   ...
END
```

Also, (Unnamed_Main) in the following example.

```
...
END
```

Fortran Module

A FORTRAN module. For example, MOD in the following example.

```
MODULE MOD
...
END MODULE
```

Fortran NameList

A FORTRAN namelist. For example, LIST1 in the following example.

```
NAMELIST /LIST1/ A, B
```

Fortran Pointer

A FORTRAN pointer block. This is an extension to some versions of FORTRAN 77. It does not have the same syntax or meaning as the POINTER statement in FORTRAN 90. For example, PTR1 in the following example.

```
POINTER /PTR1/ A, B
```

Fortran Subroutine

A FORTRAN subroutine. For example, SUB in the following example.

```
SUBROUTINE SUB
...
END
```

Fortran Unknown Include File

An unknown (not found) include file. For example, inc.com in the following example where the file is not found.

```
INCLUDE "inc.com"
```

Fortran Unknown Module

A module that appears in a USE statement, but was not found in the project sources. For example, MOD in the following example.

```
USE MOD  ! where MOD is not found
```

Fortran Unresolved Function

A function that is called, but never declared in a FUNCTION statement in any of the project source files. For example, FUNC1 in the following example.

```
X = FUNC1()  ! FUNC1 is not declared in any project file
```

Fortran Unresolved Include File

An include file that does not appear in an open database.

Fortran Unresolved Subroutine

A subroutine that is called, but never declared in a SUBROUTINE statement in any of the project source files. For example, SUB1 in the following example.

```
CALL SUB1  ! SUB1 is not declared in any project file
```

Fortran Variable

A FORTRAN variable that is not part of a common block or datapool. If the variable is part of a common block or datapool, the kind will Fortran Block Variable instead. For example, X in the following example.

```
INTEGER X
```

FORTRAN Reference Kinds

This section lists the general categories of FORTRAN reference kinds (and inverse relations) the kind names for use with the Perl and C APIs.

Category	Kind Name	See
Call (Callby)	Fortran Call	page 7–69
	Fortran Call Ptr	page 7–70
	Fortran Callby	page 7–69
	Fortran Callby Ptr	page 7–70
Contain (Containin)	Fortran Contain	page 7–70
	Fortran Containin	page 7–70
Declare (Declarein)	Fortran Declare	page 7–70
	Fortran Declarein	page 7–70
Define (Definein)	Fortran Define	page 7–70
	Fortran Define Implicit	page 7–70
	Fortran Define Private	page 7–70
	Fortran Definein	page 7–70
	Fortran Define Implicitin	page 7–70
	Fortran Define Privatein	page 7–70
End (Endby)	Fortran End	page 7–73
	Fortran End Unnamed	page 7–73
	Fortran Endby	page 7–73
	Fortran Endby Unnamed	page 7–73
Equivalence (Equivalenceby)	Fortran Equivalence	page 7–73
	Fortran Equivalenceby	page 7–73
Include (Includeby)	Fortran Include	page 7–74
	Fortran Includeby	page 7–74
ModuleUse (Moduleuseby)	Fortran ModuleUse	page 7–74
	Fortran ModuleUseby	page 7–74
	Fortran ModuleUse Only	page 7–76
	Fortran ModuleUseby Only	page 7–76
Ref (Refby)	Fortran Ref	page 7–74

Category	Kind Name	See
	Fortran Refby	page 7–76
Set (Setby)	Fortran Set	page 7–75
	Fortran Setby	page 7–75
Typed (Typedby)	Fortran Typed	page 7–75
	Fortran Typedby	page 7–75
Use (Useby)	Fortran Use	page 7–75
	Fortran Useby	page 7–75
UseModuleEntity (UseModuleEntityby)	Fortran UseModuleEntity	page 7–76
	Fortran UseModuleEntityby	page 7–76
UseRenameEntity (UseRenameEntityby)	Fortran UseRenameEntity	page 7–76
	Fortran UseRenameEntityby	page 7–76

Fortran Call

Indicates an invocation of a subroutine or function.

```
SUBROUTINE A
   ...
END SUBROUTINE
SUBROUINE B
   CALL A
END SUBROUTINE
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Fortran Call	B	A
Fortran Callby	A	B

Fortran Call Ptr

Indicates the use of a subroutine or function as an actual argument.

```
SUBROUTINE A(ARG1)
   EXTERNAL ARG1
   INTEGER ARG1
   ...
END SUBROUTINE
FUNCTION B
   ...
END FUNCTION
SUBROUTINE C
   A(B)
END SUBROUTINE
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Fortran Call Ptr	C	B
Fortran Callby Ptr	B	C

Fortran Contain

Indicates a variable is part of a common block, datapool, namelist, or pointer block.

```
COMMON /C/ VAR1
DATAPOOL /D/ VAR2
NAMELIST /N/ VAR3
POINTER /P/ VAR4
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Fortran Contain	C, D, N, P	VAR1 to VAR4
Fortran Containin	VAR1 to VAR4	C, D, N, P

Fortran Declare and Define

Indicates an entity is defined or declared in a scope. The same reference kind is used for top level program units defined in files, nested program units contained in other program units, and also for objects/types defined in program units.

For subprograms, a “define” relation is generated for the actual subroutine or function statement. Other declarations (for example, function type declarations, external statements, and declarations in interface blocks), result in “declare” relations.

For objects, the first declaring statement for the object results in a “define” relation. Subsequent declaring statements result in “declare” relations.

For private entities in modules, the reference kinds Define Private and Definein Private are used in place of Define and Definein.

For implicit definitions, Define Implicit and Definein Implicit are used in place of Define and Definein.

Example 1: In the sample.f file:

```
SUBROUTINE A
...
CONTAINS
   SUBROUTINE B
      INTEGER X
      ...
   END SUBROUTINE B
END SUBROUTINE A
```

Example 2: In the sample.f file:

```
SUBROUTINE SUB1
   EXTERNAL FUNC
   ...
END SUBROUTINE
FUNCTION FUNC
   INTEGER FUNC
END FUNCTION
```

Example3:

```
SUBROUTINE SUB1
   INTEGER :: A(:)
   POINTER A
   ...
END SUBROUTINE
```

Example4:

```
MODULE M
   PRIVATE
   INTEGER A   ! private object
   ...
END MODULE
```

Example5:

```
SUBROUTINE SUB
   M = 1
   ...
END
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Fortran Declare	Ex. 2: SUB1 Ex. 3, line 3: SUB1	Ex. 2: FUNC Ex. 3, line 3: A
Fortran Declarein	Ex. 2: FUNC Ex. 3, line 3: A	Ex. 2: SUB1 Ex. 3, line 3: SUB1
Fortran Define	Ex. 1: sample.f Ex. 1: A Ex. 1: B Ex. 2: sample.f Ex. 3, line 2: SUB1	Ex. 1: A Ex. 1: B Ex. 1: X Ex. 2: FUNC Ex. 3, line 2: A
Fortran Definein	Ex. 1: A Ex. 1: B Ex. 1: X Ex. 2: FUNC Ex. 3, line 2: A	Ex. 1: sample.f Ex. 1: A Ex. 1: B Ex. 2: sample.f Ex. 3, line 2: SUB1
Fortran Define Private	Ex. 4: M	Ex. 4: A
Fortran Definein Private	Ex. 4: A	Ex. 4: M
Fortran Define Implicit	Ex. 5: SUB	Ex. 5: M
Fortran Definein Implicit	Ex. 5: M	Ex. 5: SUB

Fortran End

Indicates the end of a function, subroutine, module, block data program unit, main program, or interface block. If the name of the unit appears in the end statement, the reference kinds are End and Endby. If the name does not appear, the kinds are End Unnamed and End Unnamedby.

```
PROGRAM MAIN
   ...
END PROGRAM MAIN

SUBROUTINE SUB
   ...
END SUBROUTINE
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Fortran End	MAIN	MAIN
Fortran EndBy	MAIN	MAIN
Fortran End Unnamed	SUB	SUB
Fortran EndBy Unnamed	SUB	SUB

Fortran Equivalence

Indicates an object appeared in an equivalence statement. These references link all equivalenced objects.

```
EQUIVALENCE (A, B, C)
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Fortran Equivalence	A	B
	A	C
	B	C
Fortran Equivalenceby	B	A
	C	A
	C	B

Fortran Include

Indicates that a file is included by a program unit.

```
PROGRAM MAIN
   INCLUDE 'common.inc'
   ...
END PROGRAM
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Fortran Include	MAIN	common.inc
Fortran Includeby	common.inc	MAIN

Fortran ModuleUse

ModuleUse and ModuleUseby indicate that a program unit has a "USE" statement for a module, and that the use statement does not have an "ONLY" clause.

```
MODULE A
   ...
END MODULE A
MODULE B
   USE A
END MODULE B
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Fortran ModuleUse	B	A
Fortran ModuleUseby	A	B

Fortran Ref

Currently unused.

Fortran Set

Indicates a variable is set.

```
SUBROUTINE SUB
   INTEGER A
   INTEGER B
   ...
   A = B
END SUBROUTINE
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Fortran Set	SUB	A
Fortran Setby	A	SUB

Fortran Typed

Indicates than an object's type is a derived type.

```
TYPE SOME_TYPE
   ...
END TYPE

TYPE (SOME_TYPE) A
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Fortran Typed	A	SOME_TYPE
Fortran Typedby	SOME_TYPE	A

Fortran Use

Indicates a use (not a set) of a variable.

```
SUBROUTINE SUB
   INTEGER A
   INTEGER B
   ...
   A = B
END SUBROUTINE
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Fortran Use	SUB	B
Fortran Useby	B	SUB

Fortran UseModuleEntity and ModuleUse Only

UseModuleEntity and UseModuleEntityby indicate that an entity is referenced in an “ONLY” clause in a “USE” statement.

ModuleUse Only and ModuleUseby Only indicate that a program unit has a “USE” statement for a module, and that the use statement has an “ONLY” clause.

```
MODULE A
   INTEGER X
   INTEGER Y
END MODULE A
MODULE B
   USE A, ONLY X
END MODULE B
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Fortran UseModuleEntity	B	X
Fortran UseModuleEntityby	X	B
Fortran ModuleUse Only	B	A
Fortran ModuleUseby Only	A	B

Fortran UseRenameEntity

Indicates an entity is renamed in a use statement.

```
USE M, LOCAL_A => A
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Fortran UseRenameEntity	LOCAL_A	A
Fortran UseRenameEntityby	A	LOCAL_A

Java Entity Kinds

This section lists the general categories of Java entity kinds, the kind names for use with the Perl and C APIs.

Category	Kind Name	See
Class	Java Abstract Class Type Default Member	page 7–80
	Java Abstract Class Type Private Member	page 7–80
	Java Abstract Class Type Protected Member	page 7–80
	Java Abstract Class Type Public Member	page 7–80
	Java Class Type Anonymous Member	page 7–80
	Java Class Type Default Member	page 7–81
	Java Class Type Private Member	page 7–81
	Java Class Type Protected Member	page 7–81
	Java Class Type Public Member	page 7–81
	Java Final Class Type Default Member	page 7–81
	Java Final Class Type Private Member	page 7–81
	Java Final Class Type Protected Member	page 7–81
	Java Final Class Type Public Member	page 7–81
	Java Static Abstract Class Type Default Member	page 7–83
	Java Static Abstract Class Type Private Member	page 7–83
	Java Static Abstract Class Type Protected Member	page 7–83
	Java Static Abstract Class Type Public Member	page 7–83
	Java Static Class Type Default Member	page 7–83
	Java Static Class Type Private Member	page 7–83
	Java Static Class Type Protected Member	page 7–83
	Java Static Class Type Public Member	page 7–83
	Java Static Final Class Type Default Member	page 7–84
	Java Static Final Class Type Private Member	page 7–84
	Java Static Final Class Type Protected Member	page 7–84
	Java Static Final Class Type Public Member	page 7–84
File	Java File	page 7–81
	Java File Jar	page 7–81
Interface	Java Interface Type Default	page 7–82
	Java Interface Type Private	page 7–82

Category	Kind Name	See
	Java Interface Type Protected	page 7–82
	Java Interface Type Public	page 7–82
Method	Java Abstract Method Default Member	page 7–80
	Java Abstract Method Protected Member	page 7–80
	Java Abstract Method Public Member	page 7–80
	Java Method Constructor Member	page 7–82
	Java Final Method Default Member	page 7–81
	Java Final Method Private Member	page 7–81
	Java Final Method Protected Member	page 7–81
	Java Final Method Public Member	page 7–81
	Java Method Default Member	page 7–82
	Java Method Private Member	page 7–82
	Java Method Protected Member	page 7–82
	Java Method Public Member	page 7–82
	Java Static Final Method Default Member	page 7–84
	Java Static Final Method Private Member	page 7–84
	Java Static Final Method Protected Member	page 7–84
	Java Static Final Method Public Member	page 7–84
	Java Static Method Default Member	page 7–84
	Java Static Method Private Member	page 7–84
	Java Static Method Protected Member	page 7–84
	Java Static Method Public Member	page 7–84
	Java Static Method Public Main Member	page 7–85
Package	Java Package	page 7–83
	Java Package Unnamed	page 7–83
Parameters	Java Catch Parameter	page 7–80
	Java Parameter	page 7–83
Variable	Java Final Variable Local Member	page 7–82
	Java Final Variable Default Member	page 7–82
	Java Final Variable Private Member	page 7–82
	Java Final Variable Protected Member	page 7–82
	Java Final Variable Public Member	page 7–82
	Java Static Final Variable Default Member	page 7–84

Category	Kind Name	See
	Java Static Final Variable Private Member	page 7–84
	Java Static Final Variable Protected Member	page 7–84
	Java Static Final Variable Public Member	page 7–84
	Java Static Variable Default Member	page 7–85
	Java Static Variable Private Member	page 7–85
	Java Static Variable Protected Member	page 7–85
	Java Static Variable Public Member	page 7–85
	Java Variable Default Member	page 7–86
	Java Variable Local Member	page 7–86
	Java Variable Private Member	page 7–86
	Java Variable Protected Member	page 7–86
	Java Variable Public Member	page 7–86
Unknown	Java Unknown Class Type Member	page 7–85
	Java Unknown Method Member	page 7–85
	Java Unknown Package	page 7–85
	Java Unknown Variable Member	page 7–85
Unresolved	Java Unresolved Method	page 7–85
	Java Unresolved Package	page 7–86
	Java Unresolved Type	page 7–86
	Java Unresolved Variable	page 7–86
Unused	Java Unused	page 7–86

General Notes

"Default" is used in kind names for entities with default visibility.

"Public" is used in kind names for entities declared as having public visibility.

"Protected" is used in kind names for entities declared as having protected visibility.

"Private" is used in kind names for entities declared as having private visibility.

"Abstract" is used in kind names for classes and methods that are declared as abstract.

"Member" is used in kind names for entities that can be class members (for example classes and methods).

"Final" is used in kind names for entities declared as final (non-extendable).

"Static" is used in kind names for entities declared as static.

"Type" is used in kind names for classes and interfaces.

Java Abstract Class Type Member

An abstract class. Visibility may be Default, Public, Protected, or Private. For example, some_class in the following example has a kind of Java Abstract Class Type Private Member.

```
private abstract class some_class {
   ...
}
```

Java Abstract Method Member

An abstract method. Visibility may be Default, Public, Protected, or Private. For example, some_meth in the following example has a kind of Java Abstract Method Protected Member.

```
protected abstract int some_meth();
```

Java Catch Parameter

Parameter in a Java catch clause. For example, some_exception in the following example has a kind of Java Catch Parameter.

```
catch (some_exception e) {
   ...
}
```

Java Class Type Anonymous Member

An anonymous class. A unique name of the form (Anon-1), (Anon-2), etc. is given to each anonymous class. For example:

```
some_class some_meth() {
   return new some_class() { // creates anonymous class
      ...
   }
}
```

Java Class Type Member

A class. Visibility may be Default, Public, Protected, or Private. For example, some_class in the following example has a kind of Java Class Type Public Member.

```
public class some_class {
   ...
}
```

Java File

A source file.

Java File Jar

A source file from a jar file.

Java Final Class Type Member

A final (non-extendable) class. Visibility may be Default, Public, Protected, or Private. For example, some_class in the following example has a kind of Java Final Class Type Protected Member.

```
protected final class some_class {
   ...
}
```

Java Final Method Member

A final (non-extendable) method. Visibility may be Default, Public, Protected, or Private. For example, some_meth in the following example has a kind of Java Final Method Private Member.

```
private final int some_meth() {
   ...
}
```

Java Final Variable Member

A final class member variable (field). Visibility may be Default, Public, Protected, or Private. For example, some_constant in the following example has a kind of Java Final Variable Private Member.

```
class some_class {
   private final int some_constant = 5;
}
```

Java Final Variable Local

A final local variable. For example, some_constant in the following example has a kind of Java Final Variable Local.

```
int some_meth() {
   final int some_constant = 5;
}
```

Java Interface Type

An interface. Visibility may be Default, Public, Protected, or Private. For example, some_interface in the following example has a kind of Java Interface Type Protected.

```
protected interface some_interface {
   ...
}
```

Java Method Constructor Member

A constructor. For example, some_class in the following example has a kind of Java Constructor.

```
class some_class {
   some_class() {  // constructor
      ...
}
```

Java Method Member

A method. Visibility may be Default, Public, Protected, or Private. For example, some_meth in the following example has a kind of Java Method Private Member.

```
private int some_meth() {
   ...
}
```

Java Package

A Java package. For example, some_pack in the following example has a kind of Java Package.

```
package some_pack;
```

The default unnamed package has a kind of Java Package Unnamed. An unnamed package entity is created if any files in the project do not have a “package” statement. Any class that appears in a file without a “package” statement belongs to the unnamed package.

Java Parameter

A method parameter. For example, a_param in the following example has a kind of Java Parameter.

```
void some_meth(int a_param) {
    ...
}
```

Java Static Abstract Class Type Member

A static abstract class. Visibility may be Default, Public, Protected, or Private. For example, some_class in the following example has a kind of Java Static Abstract Class Type Private Member.

```
private static abstract class some_class {
    ...
}
```

Java Static Class Type Member

A static class. Visibility may be Default, Public, Protected, or Private. For example, some_class in the following example has a kind of Java Static Class Type Private Member.

```
private static class some_class {
    ...
}
```

Java Static Final Class Type Member

A static final class. Visibility may be Default, Public, Protected, or Private. For example, some_class in the following example has a kind of Java Static Final Class Type Private Member.

```
private static final class some_class {
   ...
}
```

Java Static Final Method Member

A static final (non-extendable) method. Visibility may be Default, Public, Protected, or Private. For example, some_meth in the following example has a kind of Java Static Final Method Private Member.

```
private static final int some_meth() {
   ...
}
```

Java Static Final Variable Member

A static final class member variable (field). Visibility may be Default, Public, Protected, or Private. For example, some_constant in the following example has a kind of Java Static Final Variable Private Member.

```
class some_class {
   private static final int some_constant = 5;
}
```

Java Static Method Member

A static method. Visibility may be Default, Public, Protected, or Private. For example, some_meth in the following example has a kind of Java Static Method Private Member.

```
private static int some_meth() {
   ...
}
```

Java Static Method Public Main Member

A Java "main" method. This is always public and static. For example, main in the following example has a kind of Java Static Method Public Main Member.

```
public static main(String[] args) {
   ...
}
```

Java Static Variable Member

A static class member variable (field). Visibility may be Default, Public, Protected, or Private. For example, some_constant in the following example has a kind of Java Static Variable Private Member.

```
class some_class {
   private static int some_constant = 5;
}
```

Java Unknown Class Type Member

An unknown entity which from context can be identified as a class.

Java Unknown Method Member

An unknown entity which from context can be identified as a method.

Java Unknown Package

An unknown entity which from context can be identified as a package.

Java Unknown Variable Member

An unknown entity which from context can be identified as a variable.

Java Unresolved Method

A method reference that was not linked up to a declaration in the resolve process. Should not occur.

Java Unresolved Package

A package reference that was not linked up to a declaration in the resolve process. Should not occur.

Java Unresolved Type

A type reference that was not linked up to a declaration in the resolve process. Should not occur.

Java Unresolved Variable

A variable reference that was not linked up to a declaration in the resolve process. Should not occur.

Java Unused

An entity from the standard src.jar or rt.jar that is not used by or required by any files in the project. To filter out these entities, use "~unused" in queries to select Java entities. For example, udbKindParse("java class ~unused") will select all classes that are part of the project or required by files in the project.

Java Variable Member

A class member variable (field). Visibility may be Default, Public, Protected, or Private. For example, some_var in the following example has a kind of Java Variable Private Member.

```
class some_class {
   private int some_var = 5;
}
```

Java Variable Local

A local variable. For example, some_var in the following example has a kind of Java Variable Local.

```
int some_meth() {
   int some_var = 5;
}
```

Java Reference Kinds

This section lists the general categories of Java reference kinds (and inverse relations) the kind names for use with the Perl and C APIs.

Category	Kind Name	See
Call (Callby)	Java Call	page 7–88
	Java Callby	page 7–88
	Java Call Nondynamic	page 7–89
	Java Callby Nondynamic	page 7–89
Cast (Castby)	Java Cast	page 7–89
	Java Castby	page 7–89
Contain (Containin)	Java Contain	page 7–90
	Java Containin	page 7–90
Couple (Coupleby)	Java Couple	page 7–90
	Java Coupleby	page 7–90
Create (Createby)	Java Create	page 7–90
	Java Createby	page 7–90
Define (Definein)	Java Define	page 7–91
	Java Definein	page 7–91
DotRef (DotRefby)	Java DotRef	page 7–91
	Java DotRefby	page 7–91
End (Endby)	Java End	page 7–91
	Java Endby	page 7–91
Extend (Extendby)	Java Extend Couple	page 7–92
	Java Extendby Couple	page 7–92
	Java Extend External Couple	page 7–92
	Java Extendby External Couple	page 7–92
	Java Extend Implicit Couple	page 7–92
	Java Extendby Implicit Couple	page 7–92
Implement (Implementby)	Java Implement Couple	page 7–93
	Java Implementby Couple	page 7–93
Import (Importby)	Java Import	page 7–93
	Java Importby	page 7–93

Category	Kind Name	See
	Java Import Demand	page 7–93
	Java Import Demandby	page 7–93
Modify (Modifyby)	Java Modify	page 7–93
	Java Modifyby	page 7–93
Override (Overrideby)	Java Override	page 7–94
	Java Overrideby	page 7–94
Set (Setby)	Java Set	page 7–94
	Java Setby	page 7–94
	Java Set Init	page 7–94
	Java Setby Init	page 7–94
Throw (Throwby)	Java Throw	page 7–95
	Java Throwby	page 7–95
Typed (Typedby)	Java Typed	page 7–95
	Java Typedby	page 7–95
Use (Useby)	Java Use	page 7–95
	Java Useby	page 7–95

Java Call

Indicates an invocation of a method.

```
class class1 {
   void meth1() {
      ...
   }
}
class class2 {
   class1 some_obj;
   void meth2() {
      some_obj.meth1();
   }
}
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java Call	meth2	meth1
Java Callby	meth1	meth2

Java Call Nondynamic

Indicates a non-dynamic invocation of a method.

```
class class1 {
   void meth1() {
      ...
   }
}
class class2 extends class1 {
   class1 some_obj;
   void meth1() {
      super.meth1();
   }
}
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java Call Nondynamic	class2.meth1	class1.meth1
Java Callby Nondynamic	class1.meth1	class2.meth1

Java Cast

Indicates a type is used to cast an instance to a different type.

```
class c1 {
   ...
}
class c2 extends c1 {
   ...
}
class c3 {
   c2 b = new c2();
   c1 a = (c1) b;
}
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java Cast	c3	c1
Java Castby	c1	c3

Java Contain

Indicates a class is in a package.

```
package some_pack;
class some_class {
   ...
}
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java Contain	some_pack	some_class
Java Containin	some_class	some_pack

Java Couple

Indicates a coupling link as counted in the OO coupling metrics. A link is created from a class to any external class (a class that is not in the extends/implements hierarchy) that is referenced.

```
public class c1 {
}
public class c2 {
   c1 obj;
}
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java Couple	c2	c1
Java Coupleby	c1	c2

Java Create

Indicates that an instance of a class is created (“new” operator) in a scope.

```
class c1 {
   ...
}
class c2 {
   c1 a = new c1();
}
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java Create	c2	c1
Java Createby	c1	c2

Java Define

Indicates that an entity is defined in a scope.

```
class some_class {
   int some_var = 5;
}
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java Define	some_class	some_var
Java Definein	some_var	some_class

Java DotRef

Indicates that an entity name was used to the left of a "." in a qualified name.

```
package some_pack;
class class1 {
   static int y;
}
class class2 {
   void some_meth() {
      some_pack.class1.y = 7;
   }
}
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java DotRef	some_meth	some_pack
Java DotRefby	some_pack	some_meth

Java End

Indicates the end of a class, interface, or method.

```
class some_class {
   ...
}
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java End	some_class	some_class
Java Endby	some_class	some_class

Java Extend Couple

Indicates one class or interface extends another. This extend relation is used when the extended class is in a file that is part of the project. If the extended class was found in a classpath .jar file, the relation is Java Extend Couple External. If the Indicates class implicitly extends the java.lang.Object class, the relation is Java Extend Couple Implicit.

Example 1:

```
class class1 {
   ...
}
class class2 extends class1 {
   ...
}
```

Example 2:

```
class some_class extends java.io.Writer {
   ...
}
```

Example 3:

```
class some_class {
   ...
}
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java Extend Couple	Ex 1: class2	Ex 1: class1
Java Extendby Couple	Ex 1: class1	Ex 1: class2
Java Extend Couple External	Ex 2: some_class	Ex 2: java.io.Writer
Java Extendby Couple External	Ex 2: java.io.Writer	Ex 2: some_class
Java Extend Couple Implicit	Ex 3: some_class	Ex 3: java.lang.Object
Java Extendby Couple Implicit	Ex 3: java.lang.Object	Ex 3: some_class

Java Implement Couple

Indicates a class implements an interface.

```
interface some_interface {
   ...
}
class some_class implements some_interface {
   ...
}
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java Implement Couple	some_class	some_interface
Java Implementby Coupleby	some_interface	some_class

Java Import

Java Import indicates a file imports an individual class. For example, the some_file.java file might contain:

```
import pack1.some_class;
```

Java Import Demand indicates a file has an on demand import statement for a package or class. For example, the some_file.java file might contain:

```
import pack1.*;
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java Import	some_file	pack1.some_class
Java Importby	pack1.some_class	some_file
Java Import Demand	some_file	pack1
Java Importby Demand	pack1	some_file

Java Modify

Indicates that a variable's value is modified or both read and set, as with the increment (++), decrement (--), and assignment/operator combinations (*=, /=, ...).

```
void some_meth() {
   int i = 5;
   i++;
}
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java Modify	some_meth	i
Java Modifyby	i	some_meth

Java Override

Indicates that a method overrides a method from a parent class.

```
class A {
   int some_meth() {
      ...
   }
}
class B extends A{
   int some_meth() {
      ...
   }
}
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java Override	B.some_meth	A.some_meth
Java Overrideby	A.some_meth	B.some_meth

Java Set

Java Set indicates that a variable is set by a separate statement.

```
void some_meth() {
   int i;
   i = 5;
}
```

Java Set Init indicates that a variable is initialized in its declaration.

```
void some_meth() {
   int i = 5;
}
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java Set	some_meth	i
Java Setby	i	some_meth
Java Set Init	some_meth	i
Java Setby Init	i	some_meth

Java Throw

Indicates that a method throws an exception.

```
void some_meth() throws java.io.IOException {
   ...
}
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java Throw	some_meth	java_io.IOException
Java Throwby	java_io.IOException	some_meth

Java Typed

Indicates the type of a variable or parameter.

```
class class1 {
}
class class2 {
   class1 some_obj;
}
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java Typed	some_obj	class1
Java Typedby	class1	some_obj

Java Use

Indicates that a variable is used or read.

```
class some_class {
   int some_var;
   void some_meth() {
      int local_var = some_var;   // read of some_var
   }
}
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Java Use	some_meth	some_var
Java Useby	some_var	some_meth

JOVIAL Entity Kinds

This section lists the general categories of JOVIAL entity kinds, the kind names for use with the Perl and C APIs.

In general, “External” in the kind name indicates that the entity that may be used outside the program unit in which it is declared. These are entities declared in COMPOOL modules, in COPY files, and entities declared as DEF.

“Local” in the kind name indicates that the entity may not be used outside the program unit in which it is declared.

The following table lists kind names multiple times if they are part of multiple categories.

Category	Kind Name	See
Block	Jovial External Type Block	page 7–103
	Jovial Local Type Block	page 7–103
	Jovial External Component Type Block	page 7–103
	Jovial Local Component Type Block	page 7–103
	Jovial External Component Variable Block	page 7–101
	Jovial External Variable Block	page 7–100
	Jovial Local Component Variable Block	page 7–101
	Jovial Local Variable Block	page 7–100
Compool	Jovial Compool File	page 7–99
	Jovial Compool Module	page 7–102
	Jovial Unresolved Compool	page 7–104
Constant	Jovial External Constant Component Variable Item	page 7–99
	Jovial External Constant Component Variable Table	page 7–100
	Jovial External Constant Variable Item	page 7–99
	Jovial External Constant Variable Table	page 7–99
	Jovial Local Constant Component Variable Item	page 7–99
	Jovial Local Constant Component Variable Table	page 7–100
	Jovial Local Constant Variable Item	page 7–99
	Jovial Local Constant Variable Table	page 7–99
File	Jovial Copy File	page 7–99
	Jovial File	page 7–101
	Jovial Unknown Copy File	page 7–104

Category	Kind Name	See
	Jovial Unresolved Copy File	page 7–104
Item	Jovial External Component Variable Item	page 7–101
	Jovial External Constant Component Variable Item	page 7–99
	Jovial External Constant Variable Item	page 7–99
	Jovial External Variable Item	page 7–100
	Jovial Local Component Variable Item	page 7–101
	Jovial Local Constant Component Variable Item	page 7–99
	Jovial Local Constant Variable Item	page 7–99
	Jovial Local Variable Item	page 7–100
Macro	Jovial Local Macro	page 7–102
	Jovial External Macro	page 7–102
	Jovial Unresolved Macro	page 7–104
Parameter	Jovial Input Parameter	page 7–102
	Jovial Output Parameter	page 7–102
Status	Jovial Statusname	page 7–102
Subroutine	Jovial Program Procedure Subroutine	page 7–102
	Jovial External Function Subroutine	page 7–102
	Jovial External Procedure Subroutine	page 7–102
	Jovial Local Function Subroutine	page 7–102
	Jovial Local Procedure Subroutine	page 7–102
	Jovial Unresolved Subroutine	page 7–104
Table	Jovial External Component Variable Table	page 7–101
	Jovial External Constant Component Variable Table	page 7–100
	Jovial External Constant Variable Table	page 7–99
	Jovial External Variable Table	page 7–100
	Jovial Local Component Variable Table	page 7–101
	Jovial Local Constant Component Variable Table	page 7–100
	Jovial Local Constant Variable Table	page 7–99
	Jovial Local Variable Table	page 7–100
Type	Jovial External Type Block	page 7–103
	Jovial External Type Item	page 7–103
	Jovial External Type Table	page 7–103
	Jovial Local Type Block	page 7–103

Category	Kind Name	See
	Jovial Local Type Item	page 7–103
	Jovial Local Type Table	page 7–103
	Jovial External Component Type Block	page 7–103
	Jovial External Component Type Item	page 7–103
	Jovial External Component Type Table	page 7–104
	Jovial Local Component Type Block	page 7–103
	Jovial Local Component Type Item	page 7–103
	Jovial Local Component Type Table	page 7–104
	Jovial Unresolved Type	page 7–105
Unknown	Jovial Unknown	page 7–104
	Jovial Unknown Copy File	page 7–104
Unresolved	Jovial Unresolved Compool	page 7–104
	Jovial Unresolved Copy File	page 7–104
	Jovial Unresolved Macro	page 7–104
	Jovial Unresolved Subroutine	page 7–104
	Jovial Unresolved Type	page 7–105
	Jovial Unresolved Variable	page 7–105
Variable	Jovial External Component Variable Block	page 7–101
	Jovial External Component Variable Item	page 7–101
	Jovial External Component Variable Table	page 7–101
	Jovial External Constant Variable Item	page 7–99
	Jovial External Constant Variable Table	page 7–99
	Jovial External Variable Block	page 7–100
	Jovial External Variable Item	page 7–100
	Jovial External Variable Table	page 7–100
	Jovial Local Component Variable Block	page 7–101
	Jovial Local Component Variable Item	page 7–101
	Jovial Local Component Variable Table	page 7–101
	Jovial Local Constant Variable Item	page 7–99
	Jovial Local Constant Variable Table	page 7–99
	Jovial Local Variable Block	page 7–100
	Jovial Local Variable Item	page 7–100

Category	Kind Name	See
	Jovial Local Variable Table	page 7–100
	Jovial Unresolved Variable	page 7–105

Jovial Copy File

A file processed with the !COPY directive. In the following example, somefile.cop is a Copy File entity.

```
!COPY 'somefile.cop';
```

Jovial CompoolFile

An entity named with the text that appears in a compool directive. This entity is created only to allow the right-click menu in the file editor to function properly. The referenced entity name must match the text in the file for the right click to work. In the following example, somefile.cmp is a CompoolFile entity.

```
!COMPOOL ('somefile.cmp');
```

Jovial Constant Variable Item

A constant item variable. May be local or external. For example:

```
CONSTANT ITEM VERSION U = 22;
```

Jovial Constant Variable Table

A constant table variable. May be local or external. For example:

```
CONSTANT TABLE T1(1 : 10);
   ITEM LEVEL U = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10;
```

Jovial Constant Component Variable Item

An item constant that is a component of a block. May be local or external. For example:

```
BLOCK SMBLOCK;
   BEGIN
   CONSTANT ITEM VERSION U = 22;
END
```

Jovial Constant Component Variable Table

A constant table that is a component of a block. May be local or external. For example:

```
BLOCK OTBLOCK;
   BEGIN
   CONSTANT TABLE THRESHOLDS (1 : 8);
      ITEM LEVEL U = 2, 12, 26, 45, 99, 200, 315, 500;
END
```

Jovial Variable Block

A block. May be local or external. For example:

```
BLOCK INVENTORY'GROUP;
   BEGIN
   ...
   END
```

Jovial Variable Item

An item. May be local or external. For example:

```
ITEM COUNT U 5;
```

Jovial Variable Table

A table. May be local or external. For example:

```
TABLE TBL1(1 : 3);
   BEGIN
   ...
   END;
```

Jovial Component Variable Block

A block that is a component of another block. May be local or external. For example:

```
BLOCK MAINGROUP;
   BEGIN
   ...
   BLOCK SUBGROUP;  -- A Component Variable BLOCK
      BEGIN
      ...
      END
   ...
   END
```

Jovial Component Variable Item

An item that is a component of a block or table. May be local or external. For example:

```
BLOCK MAINGROUP;
   BEGIN
      ITEM ITM1 U;  -- ITM1 is a Component Variable Item
      ...
   END
```

Jovial Component Variable Table

A table that is a component of a block. May be local or external. For example:

```
BLOCK MAINGROUP;
  BEGIN
    TABLE TBL1(10); -- TBL1 is Component Variable Table
    BEGIN
    ...
    END
  ...
END
```

Jovial File

A JOVIAL source file.

Jovial Macro

A JOVIAL define macro. May be local or external. For example:

```
DEFINE SIZE "100"
```

Jovial Compool Module

A compool. For example:

```
COMPOOOL cmpname;
```

Jovial Input Parameter / Jovial Output Parameter

An input or output parameter to a subroutine. For example:

```
// P1 is input parameter, P2 is output
PROC SOMEPROC (P1 : P2);
 ...
```

Jovial Program Procedure Subroutine

A JOVIAL program.

```
PROGRAM someproc;
   ...
```

Jovial Statusname

A value name for a status variable or type. For example:

```
TYPE COLOR STATUS(V(RED), V(YELLOW), V(GREEN));
// Red, Yellow, and Green are all JOVIAL statusnames.
```

Jovial Function Subroutine

A function (proc with a return value). May be local or external. For example:

```
PROC sum (p1, p2) U;
```

Jovial Procedure Subroutine

A procedure (proc with no return value). May be local or external. For example:

```
PROC sum (p1, p2 : p3;
```

Jovial Type Block

A block type. May be local or external. For example:

```
TYPE INVENTORY BLOCK
   BEGIN
   ...
   END
```

Jovial Type Item

An item type. May be local or external. For example:

```
TYPE MODIFIER F 20;
```

Jovial Type Table

A table type. May be local or external. For example:

```
TYPE PARRAY TABLE(10);
   ITEM POINT U;
```

Jovial Component Type Block

A block that is a component of a block type. May be local or external. For example:

```
TYPE B'TYPE BLOCK
  BEGIN
  ITEM ITEM1 U;
  BLOCK N'BLOCK; // N'BLOCK is 'component type block'
     BEGIN
     ITEM COMP1 U;
     END;
  END
```

Jovial Component Type Item

An item that is a component of a block or array type. May be local or external. For example:

```
 TYPE TAB'TYPE TABLE;
    BEGIN
    ITEM COMP2 U;  // COMP2 is a 'Component Type Item'
    END
```

Jovial Component Type Table

A table that is a component of a block type. May be local or external. For example:

```
TYPE B'TYPE BLOCK;
   BEGIN
   TABLE B'ARRAY; // B'ARRAY is a 'Component Type Table'
   ITEM COMP1 U;
   END
```

Jovial Unknown

An entity that is referenced but is not know from the source code in the project.

Jovial Unknown Copy File

A copy file that was not found in the project or in any of the specified copy directories.

Jovial Unresolved Compool

A compool reference that was not linked up to a declaration in the resolve process. Should not occur.

Jovial Unresolved Copy File

A copy file reference that was not linked up to a declaration in the resolve process. Should not occur.

Jovial Unresolved Macro

A macro reference that was not linked up to a declaration in the resolve process. Should not occur.

Jovial Unresolved Subroutine

A subroutine reference that was not linked up to a declaration in the resolve process. This will occur when there is a 'REF' for a subroutine but no 'DEF' is found. The following REF makes an “Unresolved Subroutine” if no DEF is found for SOMEPROC:

```
REF PROC SOMEPROC;
```

Jovial Unresolved Type

A type reference that was not linked up to a declaration in the resolve process. Should not occur.

Jovial Unresolved Variable

A variable reference that was not linked up to a declaration in the resolve process. This will occur when there is a 'REF' for a variable but no 'DEF' is found. The following REF makes an “Unresolved Variable” if no DEF is found for the item:

```
REF ITEM RATE U;
```

JOVIAL Reference Kinds

This section lists the general categories of JOVIAL reference kinds (and inverse relations) the kind names for use with the Perl and C APIs.

Category	Kind Name	See
Call	Jovial Call	page 7–107
	Jovial Callby	page 7–107
Compool	Jovial CompoolAccess	page 7–108
	Jovial CompoolAccessby	page 7–108
	Jovial CompoolAccess All	page 7–108
	Jovial CompoolAccessby All	page 7–108
	Jovial CompoolFileAccess	page 7–109
	Jovial CompoolFileAccessby	page 7–109
Copy	Jovial Copy	page 7–111
	Jovial Copyby	page 7–111
Declare	Jovial Declare	page 7–111
	Jovial Declarein	page 7–111
Define	Jovial Define	page 7–111
	Jovial Definein	page 7–111
Inline	Jovial Declare Inline	page 7–112
	Jovial Declarein Inline	page 7–112
Item	Jovial Item Access	page 7–109
	Jovial Item Accessby	page 7–109
	Jovial Item Access All	page 7–110
	Jovial Item Accessby All	page 7–110
	Jovial Item Access Implicit	page 7–110
	Jovial Item Accessby Implicit	page 7–110
Like	Jovial Like	page 7–112
	Jovial Likeby	page 7–112
Overlay	Jovial Overlay	page 7–113
	Jovial Overlayby	page 7–113
	Jovial Overlay Implicit	page 7–113
	Jovial Overlay Implicitby	page 7–113

Category	Kind Name	See
Ptr	Jovial Typed Ptr	page 7–113
	Jovial Typedby Ptr	page 7–113
Set	Jovial Set	page 7–114
	Jovial Setby	page 7–114
	Jovial Set Init	page 7–114
	Jovial Setby Init	page 7–114
Typed	Jovial Typed	page 7–114
	Jovial Typedby	page 7–114
Use	Jovial Use	page 7–115
	Jovial Useby	page 7–115
Value	Jovial Value	page 7–115
	Jovial Valueof	page 7–115

Jovial Call and Jovial Callby

Indicates an invocation of a subroutine. For example:

```
PROC P1;
   BEGIN
   ...
   END;

PROC P2;
   BEGIN
   P1;
   END;
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial Call	P2	P1
Jovial Callby	P1	P2

Jovial CompoolAccess/Accessby All

Indicates that a compool was named in a !COMPOOL directive using syntax that makes all declarations in the compool visible in the scope in which the !COMPOOL directive appears. For example:

```
START COMPOOL COMP1;
...
TERM

START COMPOOL COMP2;
!COMPOOL (COMP1);
...
TERM
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial CompoolAccess All	COMP2	COMP1
Jovial CompoolAccessby All	COMP1	COMP2

Jovial CompoolAccess and Jovial CompoolAccessby

Indicates that a compool was named in a !COMPOOL directive using syntax that makes only selected declarations in the compool visible in the scope in which the !COMPOOL directive appears. For example:

```
START COMPOOL COMP1;
   ITEM SOME'VAR U;
   ...
TERM
START COMPOOL COMP2;
!COMPOOL COMP1 SOME'VAR;
...
TERM
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial CompoolAccess	COMP2	COMP1
Jovial CompoolAccessby	COMP1	COMP2

Jovial CompoolFileAccess/Accessby

Created when !COMPOOL directive uses name of compool file instead of compool name. This links the context where the !COMPOOL occurs to a 'compoolfile' entity. (See also *Jovial CompoolFile* on page 7–99.) For example, file f3.cpl contains:

```
START COMPOOL BDATATYPES;
  ...
TERM
```

And, file f1.jov contains:

```
START
   !COMPOOL ('f3.cmp');
   ...
TERM
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial CompoolFileAccess	f1.jov	f3.cmp
Jovial CompoolFileAccessby	f3.cmp	f1.jov

Jovial ItemAccess and Jovial ItemAccessby

Indicates that an entity was named in a !COMPOOL directive as an entity to make visible from the specified compool. For example:

```
START COMPOOL COMP1;
  ITEM SOME'VAR U;
  ...
TERM
START COMPOOL COMP2;
  !COMPOOL COMP1 SOME'VAR;
  ...
TERM
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial ItemAccess	COMP2	SOME'VAR
Jovial ItemAccessby	SOME'VAR	COMP2

Jovial ItemAccess All and Jovial ItemAccessby All

Indicates that an entity was named in a !COMPOOL directive and the name was enclosed in parentheses. This syntax make the named entity and all its component entities available.

```
START COMPOOL COMP1;
   TABLE GRID(20,20);
   BEGIN
     ITEM XPART U;
     ITEM YPART U;
   END
TERM
START COMPOOL COMP2;
!COMPOOL COMP1 (GRID);
...
TERM
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial ItemAccess All	COMP2	GRID
Jovial ItemAccessby All	GRID	COMP2

Jovial ItemAccess/Accessby Implicit

Indicates the entity is a component of an entity referenced with the compool directive syntax that makes all components of a named entity available. When the syntax:

```
"!COMPOOL 'cmp_name' (obj_name)"
```

is used, all components of 'obj_name' are referenced with an 'ItemAccess Implicit' reference.

For example:

```
START COMPOOL COMP1;
   TABLE GRID(20,20);
   BEGIN
     ITEM XPART U;
     ITEM YPART U;
   END
TERM
START COMPOOL COMP2;
!COMPOOL COMP1 (GRID);
...
TERM
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial ItemAccess Implicit	COMP2, COMP2	XPART, YPART
Jovial ItemAccessby Implicit	XPART, YPART	COMP2, COMP2

Jovial Copy and Jovial Copyby

Indicates the file was named in a !copy directive. For example, suppose file test.jov contains:

```
!COPY 'cpfile.cop';
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial Copy	test.jov	cpfile.cop
Jovial Copyby	cpfile.cop	test.jov

Jovial Declare and Jovial Declarein

Used for a non-defining declaration of an entity. For example, a REF declaration of a variable or subroutine will generate a 'declare' relation. For example:

```
START COMPOOL COMP1;
   REF ITEM SOME'ITEM;
   ...
TERM
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial Declare	COMP1	SOME'ITEM
Jovial Declarein	SOME'ITEM	COMP1

Jovial Define and Jovial Definein

Used for the definition of an entity. For example, file file.jov contains:

```
START COMPOOL COMP2;
   DEF ITEM SOME'ITEM;
   ...
TERM
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial Define	file.jov, COMP2	COMP2, SOME'ITEM
Jovial Definein	COMP2 SOME'ITEM	file.jov COMP2

Jovial Declare Inline and Jovial Declarein Inline

Indicates the name appears in an INLINE declaration. For example:

```
 1  DEF PROC SOMEPROC;
 2     BEGIN
 3     ...
 4     INLINE TALLY;  // result in table below
 5     ...
 6     PROC TALLY;
 7 BEGIN
 8 ...
 9 END
10     END
11 TERM
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial Declare Inline	SOMEPROC	TALLY
Jovial Declarein Inline	TALLY	SOMEPROC

Jovial Like and Jovial Likeby

Indicates the name was used in a LIKE clause in a table type definition. For example:

```
TYPE ACTIVE'TABLE TABLE
   LIKE ID'TABLE;
   BEGIN
   ...
   END
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial Like	ACTIVE'TABLE	ID'TABLE
Jovial Likeby	ID'TABLE	ACTIVE'TABLE

Jovial Overlay and Jovial Overlayby

Indicates the name was used in an overlay statement. When several names are given in an overlay statement, an overlay relation is created between each pair. For example:

```
OVERLAY COUNT : TIME : RESULT;
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial Overlay	COUNT, COUNT, TIME	TIME, RESULT, RESULT
Jovial Overlayby	TIME, RESULT, RESULT	COUNT, COUNT, TIME

Jovial Overlay Implicit and Jovial Overlayby Implicit

Indicates that the entity is overlaid with another entity through use of a POS clause which positioned data at the same memory location as another variable. For example:

```
TABLE F'TYPE W 2;
    BEGIN
       ITEM FIXED1 A 0    POS (0,0);
       ITEM FIXED2 A 1    POS (0,1);
       ITEM FIXED3 A 0,31 POS (0,0);
     END
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial Overlay Implicit	FIXED1, FIXED2	FIXED3, FIXED3
Jovial Overlayby Implicit	FIXED3, FIXED3	FIXED1, FIXED2

Jovial Typed Ptr and Jovial Typedby Ptr

Indicates the type is used in the declaration of a pointer to the type. For example:

```
TYPE BLOCK'TYPE BLOCK;
   BEGIN
   ...
   END
TYPE BLOCK'PTR P BLOCK'TYPE;
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial Typed Ptr	BLOCK'PTR	BLOCK'TYPE
Jovial Typedby Ptr	BLOCK'TYPE	BLOCK'PTR

Jovial Set and Jovial Setby

Indicates the variable is set. For example:

```
DEF PROC SOMEPROC;
  BEGIN
     ITEM COUNT U;
     COUNT = 0;
  END;
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial Set	SOMEPROC	COUNT
Jovial Setby	COUNT	SOMEPROC

Jovial Set Init and Jovial Setby Init

Indicates the variable is initialized in its definition. For example:

```
DEF PROC SOMEPROC;
  BEGIN
     ITEM COUNT U = 0;
  END;
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial Set Init	SOMEPROC	COUNT
Jovial Setby Init	COUNT	SOMEPROC

Jovial Typed and Jovial Typedby

Indicates the entity was used as the type in a variable or function declaration. For example:

```
DEF PROC SOMEPROC;
   BEGIN
      TYPE COUNTER U 10;
      ITEM CT1 COUNTER;
   END;
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial Typed	CT1	COUNTER
Jovial Typedby	COUNTER	CT1

Jovial Use and Jovial Useby

Indicates the variable was read. For example:

```
DEF PROC SOMEPROC;
    BEGIN
       ITEM RADIUS U 10;
       ITEM DIAMETER U 10;
       ...
       DIAMETER = RADIUS * 2;
    END;
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial Use	SOMEPROC	RADIUS
Jovial Useby	RADIUS	SOMEPROC

Jovial Value and Jovial Valueof

Indicates an entity is a status constant for a status item or type. For example:

```
ITEM COLOR STATUS (V(RED), V(GREEN), V(YELLOW));
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Jovial Value	COLOR	V(RED)
Jovial Valueof	V(RED)	COLOR

Pascal Entity Kinds

This section lists the general categories of Pascal entity kinds, the kind names for use with the Perl and C APIs.

"Local" is part of the kind text for all entities that may not be used outside the program unit in which they are declared.

"Global" is part of the kind text for all entities that may be used outside the program unit in which they are declared. These are entities declared in the outermost declaration level of a compilation unit that creates an "environment", and all declarations with a [GLOBAL] or [EXTERNAL] attribute.

Category	Kind Name	See
CompUnit	Pascal CompUnit Module	page 7–118
	Pascal CompUnit Program	page 7–118
Constant	Pascal Constant Global	page 7–118
	Pascal Constant Local	page 7–118
Ennumerator	Pascal Enumerator Global	page 7–118
	Pascal Enumerator Local	page 7–118
Environment	Pascal Environment	page 7–119
Field	Pascal Field Discrim Global	page 7–119
	Pascal Field Discrim Local	page 7–119
	Pascal Field Global	page 7–119
	Pascal Field Local	page 7–119
File	Pascal File	page 7–119
	Pascal File Include	page 7–119
Parameter	Pascal Parameter Function Global	page 7–120
	Pascal Parameter Function Local	page 7–120
	Pascal Parameter Procedure Global	page 7–120
	Pascal Parameter Procedure Local	page 7–120
	Pascal Parameter Value Global	page 7–120
	Pascal Parameter Value Local	page 7–120
	Pascal Parameter Var Global	page 7–121
	Pascal Parameter Var Local	page 7–121
Predeclared	Pascal Predeclared Routine	page 7–121
	Pascal Predeclared Type	page 7–121

Category	Kind Name	See
	Pascal Predeclared Variable	page 7–121
Routine	Pascal Routine Function Global	page 7–120
	Pascal Routine Function Global Asynchronous	page 7–120
	Pascal Routine Function Local	page 7–120
	Pascal Routine Function Local Asynchronous	page 7–120
	Pascal Routine Procedure Global	page 7–121
	Pascal Routine Procedure Global Asynchronous	page 7–122
	Pascal Routine Procedure Local	page 7–121
	Pascal Routine Procedure Local Asynchronous	page 7–122
	Pascal Routine Procedure Local Initialize	page 7–122
Sql	Pascal Sql Alias	page 7–124
	Pascal Sql Column	page 7–124
	Pascal Sql Cursor	page 7–124
	Pascal Sql Group	page 7–125
	Pascal Sql Index	page 7–125
	Pascal Sql Procedure	page 7–125
	Pascal Sql Role	page 7–125
	Pascal Sql Rule	page 7–125
	Pascal Sql Statement	page 7–125
	Pascal Sql Table	page 7–126
	Pascal Sql Table GlobalTemp	page 7–126
	Pascal Sql Unresolved	page 7–126
	Pascal Sql Unresolved Table	page 7–126
	Pascal Sql User	page 7–126
Type	Pascal Type Global	page 7–122
	Pascal Type Local	page 7–122
	Pascal Type Unnamed Local	page 7–122
Unknown	Pascal Unknown Environment	page 7–123
	Pascal Unknown Function	page 7–123
	Pascal Unknown Type	page 7–123
	Pascal Unknown Variable	page 7–123
Unresolved	Pascal Unresolved Environment	page 7–123
	Pascal Unresolved Global Entity	page 7–123

Category	Kind Name	See
	Pascal Unresolved Global Function	page 7–123
	Pascal Unresolved Global Procedure	page 7–124
	Pascal Unresolved Global Variable	page 7–124
Variable	Pascal Variable Global	page 7–122
	Pascal Variable Local	page 7–122

Pascal Constant

A declared constant. May be local or global. For example:

```
MODULE SOME_MOD;
   CONST
      Year = 2004;
END;
```

Pascal CompUnit Module

A Pascal module compilation unit.

```
MODULE SOME_MOD;
...
END;
```

Pascal CompUnit Program

A Pascal program compilation unit. For example:

```
PROGRAM SOME_PROG;
...
END;
```

Pascal Enumerator

A value in an enumeration type. May be local or global. For example:

```
TYPE
   Day = (Mon, Tues, Wed, Thurs, Fri, Sat, Sun);
Mon, Tues Wed, Thurs, Fri, Sat and Sun are all
enumerators.
```

Pascal Environment

An environment, created with an [ENVIRONMENT ...] attribute. For example:

```
[ENVIRONMENT('GLB_ENV')] {creates environment GLB_ENV}
MODULE SOME_MOD
   ...
END.
```

Pascal Field

A component of a record. May be local or global. For example:

```
TYPE
   REC_TYPE = RECORD
      COMP1 : INTEGER;   { COMP1 and COMP2 are fields }
      COMP2 : REAL;
   END;
```

Pascal Field Discrim

A discriminant in a record variant or discriminant in an array schema. May be local or global. For example:

```
TYPE
   REC_TYPE(DISCRIM : INTEGER)  = RECORD
            { DISCRIM is a field discrim }
     ...
   END;
```

Pascal File

A Pascal source code file.

Pascal File Include

A Pascal file processed with a %include directive.

Pascal Routine Function

A Pascal function with no [ASYNCHRONOUS] attribute. May be local or global. For example:

```
Function SomeFunc(P1, P2 : INTEGER) : INTEGER;
   BEGIN
   ...
   END;
```

Pascal Routine Function Asynchronous

A Pascal function with an [ASYNCHRONOUS] attribute. May be local or global. For example:

```
[ASYNCHRONOUS] Function AsynchFunc(P1, P2 : INTEGER) :
INTEGER;
   BEGIN
   ...
   END;
```

Pascal Parameter Function

A function declared as a parameter to another routine. May be local or global. For example:

```
PROCEDURE SOMEPROC( FUNCTION P_Func( P1, P2 : REAL ) : REAL);
{ P_Func is a parameter function }
```

Pascal Parameter Procedure

A procedure declared as a parameter to another routine. May be local or global. For example:

```
PROCEDURE SOMEPROC( PROCEDURE P_Proc( P1, P2 : REAL ));
{ P_Proc is a parameter procedure }
```

Pascal Parameter Value

A parameter that is based by value to a routine (actual parameter can not be modified by the called routine). May be local or global. For example:

```
PROCEDURE SOMEPROC (P1 : REAL);
{ P1 is a value parameter }
```

Pascal Parameter Var

A parameter to a routine which is specified as being a var parameter (actual parameter can be modified by the called routine). May be local or global. For example:

```
PROCEDURE SOMEPROC (VAR P1 : REAL);
{ P1 is a var parameter }
```

Pascal Predeclared Routine

A routine that is predefined by the Pascal language. Database entities are created only for those predeclared routines that are actually referenced in the project code. For example:

```
GETTIMESTAMP( Cur_Time );
{ GETTIMESTAMP is a predeclared routine }
```

Pascal Predeclared Type

A type that is predefined by the Pascal language. Database entities are created only for those predeclared types that are actually referenced in the project code. For example:

```
VAR
    Cur_Time : TIMESTAMP;
    { TIMESTAMP is a predeclared type }
```

Pascal Predeclared Variable

An object that is predefined by the Pascal language. Database entities are created only for those predeclared objects that are actually referenced in the project code. For example:

```
TYPE
   POSINT = 1..MAXINT;
   { MAXINT is a predeclared variable }
```

Pascal Routine Procedure

A Pascal procedure with no [ASYNCHRONOUS] or [INITIALIZE] attributes. May be local or global. For example:

```
Procedure SomeProc(P1, P2 : INTEGER);
   BEGIN
   ...
   END;
```

Pascal Routine Procedure Asynchronous

A Pascal procedure with an [ASYNCHRONOUS] attribute. May be local or global. For example:

```
[ASYNCHRONOUS] Procedure AsynchProc(P1, P2 : INTEGER);
   BEGIN
   ...
   END;
```

Pascal Routine Procedure Local Initialize

A Pascal procedure with an [INITIALIZE] attribute. For example:

```
[INITIALIZE] Procedure InitProc(P1, P2 : INTEGER);
   BEGIN
   ...
   END;
```

Pascal Type Unnamed Local

An type entity created when a variable is declared as a record or enumeration variable by giving the record or enumeration description directly in the variable declaration. For example:

```
VAR
   MY_REC = RECORD
        COMP1 : INTEGER;
        COMP2 : REAL;
    END;
```

This creates an unnamed record type with components 'COMP1' and 'COMP2'. The unnamed record is linked to 'MY_REC' with the 'typed'/'typedby' relations.

Pascal Type

A declared type. May be local or global. For example:

```
TYPE
   Month = 1..12;
```

Pascal Variable

A declared variable. May be local or global. For example:

```
IntVar : INTEGER;
```

Pascal Unknown Environment

An environment referenced in an 'INHERIT' attributes that is not defined in the project source files. For example:

```
[INHERIT ('SOMEENV')] { where no SOMEENV is not found }
```

Pascal Unknown Variable

A referenced variable for which no definition is found.

Pascal Unknown Function

A referenced function for which no definition is found.

Pascal Unknown Type

A referenced type for which no definition is found.

Pascal Unresolved Environment

An environment reference that was not linked up to a declaration during the resolve process. Should not occur.

Pascal Unresolved Global Entity

An entity reference that was not linked up to a declaration during the resolve process. Should not occur.

Pascal Unresolved Global Function

A function reference that was not linked up to a definition during the resolve process. This will occur when there is an EXTERNAL attribute on a function declaration, but no definition of the function is found. For example:

```
[EXTERNAL] Function SomeFunc(p1 : Integer) : Integer;
EXTERNAL;
   { SomeFunc is unresolved if no definition is found }
```

Pascal Unresolved Global Procedure

An external procedure that was not linked up to a definition during the resolve process. This will occur when there is an EXTERNAL attribute on a procedure declaration, but no definition of the function is found. For example:

```
[EXTERNAL] Function SomeProc(p1 : Integer); EXTERNAL;
   { SomeProc is unresolved if no definition is found }
```

Pascal Unresolved Global Variable

A variable declared as EXTERNAL that was not linked up to a definition during the resolve process.

```
VAR
   SomeObj : [EXTERNAL] Integer;
   { SomeObj is unresolved if no definition is found }
```

Pascal Sql Alias

Table alias used in select statements. For example:

```
select e.name from employees e
{ e is an sql alias }
```

Pascal Sql Column

A column from an SQL table. A column entity will be created for referenced columns as well as for columns that are defined in table create/declare statements. For example:

```
create table employee
  (enumber smallint,
   address varchar(80));
 { enumber and address are columns }
exec sql select col1 into :x from table1...;
  { col1 column entity created }
```

Pascal Sql Cursor

An SQL cursor. For example:

```
EXEC SQL DECLARE A_CURSOR CURSOR FOR
   SELECT A.DATE FROM
      SOME_TBL A;
{ A_CURSOR is an sql cursor }
```

Pascal Sql Group

An SQL group. For example:

```
create group usr_group;
```

Pascal Sql Index

An SQL index. For example:

```
create index name_index on employee (last, first);
{ name_index is an sql index }
```

Pascal Sql Procedure

An SQL created database procedure. For example:

```
exec sql create
   procedure sql_proc as
   begin
   ...
   end;
```

Pascal Sql Role

An SQL role. For example:

```
create role some_role with nopassword;
```

Pascal Sql Rule

An SQL rule that executes an SQL procedure when a specified condition occurs. For example:

```
create rule some_rule after delete, insert,
   update of some_tble
   execute procedure sql_proc;
{ some_rule is an sql rule }
```

Pascal Sql Statement

An entity created for an SQL statement. An SQL statement entity is created for each unique statement kind occurring in the code. For example:

```
exec sql create table...;
exec sql select ...;
    { generates 'sql create' and 'sql select' entities }
```

Pascal Sql Table

An SQL table. For example:

```
exec sql create table employee...;
  { generates table employee }
```

Pascal Sql Table GlobalTemp

An SQL global temporary table. Created when session.tbl_name syntax is used as a parameter in an execute procedure statement. For example:

```
exec sql execute procedure some_proc(p1 = session.tmp_tbl);
    { tmp_tbl is a global temporary table }
```

Pascal Sql User

An SQL user. For example:

```
exec sql create user user1 with noprivileges;
```

Pascal Sql Unresolved

A entity reference that was not linked up to a declaration in the resolve process. Should not occur.

Pascal Sql Unresolved Table

An SQL table reference that was not linked up to a declaration in the resolve process. This will occur when a table is referenced in a select or other statement, and no create or declare for the table is found. For example:

```
exec sql select col1 into :x from table1...;
  { if table1 is never declared or created, it is an }
  { unresolved table }
```

Pascal Reference Kinds

This section lists the general categories of Pascal reference kinds (and inverse relations) the kind names for use with the Perl and C APIs. The following table lists kind names multiple times if they are part of multiple categories.

Category	Kind Name	See
Call	Pascal Call	page 7–128
	Pascal Callby	page 7–128
	Pascal Sql Call	page 7–133
	Pascal Sql Callby	page 7–133
	Pascal Sql Call Statement	page 7–134
	Pascal Sql Callby Statement	page 7–134
Contain	Pascal Contain	page 7–129
	Pascal Containin	page 7–129
Declare	Pascal Declare	page 7–129
	Pascal Declarein	page 7–129
Define	Pascal Define	page 7–130
	Pascal Definein	page 7–130
	Pascal Sql Define	page 7–134
	Pascal Sql Definein	page 7–134
End	Pascal End	page 7–130
	Pascal Endby	page 7–130
Inherit	Pascal Inherit	page 7–131
	Pascal Inheritby	page 7–131
	Pascal Inheritenv	page 7–131
	Pascal Inheritenvby	page 7–131
Hasenvironment	Pascal Hasenvironment	page 7–132
	Pascal Hasenvironmentby	page 7–132
Set	Pascal Set	page 7–132
	Pascal Setby	page 7–132
	Pascal Set Init	page 7–132
	Pascal Setby Init	page 7–132
	Pascal Sql Set	page 7–135

Category	Kind Name	See
	Pascal Sql Setby	page 7–135
Typed	Pascal Typed	page 7–133
	Pascal Typedby	page 7–133
	Pascal Sql Typed	page 7–135
	Pascal Sql Typedby	page 7–135
Use	Pascal Use	page 7–133
	Pascal Useby	page 7–133
	Pascal Sql Use	page 7–135
	Pascal Sql Useby	page 7–135
Sql	Pascal Sql Call	page 7–133
	Pascal Sql Callby	page 7–133
	Pascal Sql Call Statement	page 7–134
	Pascal Sql Callby Statement	page 7–134
	Pascal Sql Define	page 7–134
	Pascal Sql Definein	page 7–134
	Pascal Sql Set	page 7–135
	Pascal Sql Setby	page 7–135
	Pascal Sql Typed	page 7–135
	Pascal Sql Typedby	page 7–135
	Pascal Sql Use	page 7–135
	Pascal Sql Useby	page 7–135

Pascal Call and Pascal Callby

Indicates an invocation of a function or procedure. For example:

```
Module SomeMod;
   Procedure Proc1;
      Begin
      ...
      End;
   Procedure Proc2;
      Begin
 Proc1;
      End;
End.
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal Call	Proc2	Proc1
Pascal Callby	Proc1	Proc2

Pascal Contain and Pascal Containin

Links an environment entity to a compilation unit that the environment contains. For example:

```
[ENVIRONMENT ('SomeEnv')]
Module SomeMod;
...
END.
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal Contain	SomeEnv	SomeMod
Pascal Containin	SomeMod	SomeEnv

Pascal Declare and Pascal Declarein

Indicates that an entity is declared (as opposed to defined). For example, an [EXTERNAL] declaration of a variable or routine creates a "declare" relation. For example:

```
Module SomeMod;
   VAR
      SomeVar : [EXTERNAL] Integer;
   ...
End.
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal Declare	SomeMod	SomeVar
Pascal Declarein	SomeVar	SomeMod

Pascal Define and Pascal Definein

Indicates that an entity is defined. For example:

```
Module SomeMod;
   Procedure Proc1;
      Begin
      ...
      End;
End.
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal Define	SomeMod	Proc1
Pascal Definein	Proc1	SomeMod

Pascal End and Pascal Endby

Marks the end of a program, module, or routine. For example:

```
1   Module SomeMod;
2      Procedure Proc1;
3         Begin
4         ...
5         End;
6   End.
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal End	Proc1 (line 5) SomeMod (line 6)	Proc1 SomeMod
Pascal Endby	Proc1 (line 5) SomeMod (line 6)	Proc1 SomeMod

Pascal Inherit and Pascal Inheritby

Indicates that a module is inherited by another module or program. This is a direct link from one unit to another. The environment entity is linked to with an 'inheritenv' relation. For example:

```
[ENVIRONMENT ('env1')]
MODULE MOD1;
...
END.

[INHERIT ('env1')]
MODULE MOD2;
...
END.
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal Inherit	MOD2	MOD1
Pascal Inheritby	MOD1	MOD2

Pascal Inheritenv and Pascal Inheritenvby

Indicates that an environment is inherited by a module or program. For example:

```
[ENVIRONMENT ('env1')]
MODULE MOD1;
...
END.

[INHERIT ('env1')]
MODULE MOD2;
...
END.
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal Inheritenv	MOD2	env1
Pascal Inheritenvby	env1	MOD2

Pascal Hasenvironment/Hasenvironmentby

Links source file entity to the environment it creates. For example, suppose file some_file.pas contains:

```
[ENVIRONMENT ('env1')]
MODULE MOD1;
...
END.
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal Hasenvironment	some_file.pas	env1
Pascal Hasenvironmentby	env1	some_file.pas

Pascal Set and Pascal Setby

Indicates a variable is set. For example:

```
PROCECURE SOMEPROC;
   BEGIN
   ...
   SOME_VAR := 0;
   ...
   END;
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal Set	SOMEPROC	SOME_VAR
Pascal Setby	SOME_VAR	SOMEPROC

Pascal Set Init and Pascal Setby Init

Indicates a variable is initialized in its definition. For example:

```
PROCECURE SOMEPROC;
   VAR
     SOME_VAR : INTEGER := 0;
   BEGIN
     ...
   END;
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal Set Init	SOMEPROC	SOME_VAR
Pascal Setby Init	SOME_VAR	SOMEPROC

Pascal Typed and Pascal Typedby

Indicates that an entity was used as the type in a variable or function declaration. For example:

```
MODULE SOME_MOD;
   TYPE
     MONTH = 1..12;
   VAR
     SOME_VAR : MONTH;
   END.
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal Typed	SOME_VAR	MONTH
Pascal Typedby	MONTH	SOME_VAR

Pascal Use and Pascal Useby

Indicates a variable was read. For example:

```
PROCEDURE SOMEPROC;
   VAR
      RADIUS : INTEGER;
      DIAMETER : INTEGER;
   BEGIN
      ...
      DIAMETER := RADIUS * 2;
   END;
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal Use	SOMEPROC	RADIUS
Pascal Useby	RADIUS	SOMEPROC

Pascal Sql Call and Pascal Sql Callby

Indicates that an SQL procedure was called. For example:

```
PROCEDURE SOMEPROC;
BEGIN
   EXEC SQL EXECUTE PROCEDURE SQL_PROC;
END;
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal Sql Call	SOMEPROC	SQL_PROC
Pascal Sql Callby	SQL_PROC	SOMEPROC

Pascal Sql Call/Callby Statement

Indicates that an SQL statement was executed. Each exec SQL statement generates a Sql Call relation. For example:

```
PROCEDURE SOMEPROC;
BEGIN
   exec sql select name into :n_var
         from employee where enum = :num;
END;
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal Sql Call Statement	SOMEPROC	sql select
Pascal Sql Callby Statement	sql select	SOMEPROC

Pascal Sql Define and Pascal Sql Definein

Indicates an SQL entity is defined. Link goes from context to the defined entity. For example:

```
1  PROCEDURE SOMEPROC;
2  BEGIN
3     exec sql create table employee
4          (enumber smallint,
5           address varchar(80));
6  END;
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal Sql Define	SOMEPROC (line 3) employee (line 4) employee (line 5)	employee enumber address
Pascal Sql Definein	employee (line 3) enumber (line 4) address (line 5)	SOMEPROC employee employee

Pascal Sql Set and Pascal Sql Setby

Indicates an SQL entity was set. For example, this could be the set of a table appearing in an 'into' clause in an insert statement, or the set of a table and columns via a copy statement. For example:

```
1  PROCEDURE SOMEPROC;
2  BEGIN
3    exec sql insert into some_tbl (col1, col2)
4                  values (0, 1);
5  END;
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal Sql Set	SOMEPROC (line 3)	some_tbl
	SOMEPROC (line 4)	col1
	SOMEPROC (line 4)	col2
Pascal Sql Setby	some_tbl (line 3)	SOMEPROC
	col1 (line 4)	SOMEPROC
	col2 (line 4)	SOMEPROC

Pascal Sql Typed and Pascal Sql Typedby

Links a table alias used in a select statement to the table for which it is an alias. For example:

```
select e.name from employees e
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal Sql Typed	e	employees
Pascal Sql Typedby	employees	e

Pascal Sql Use and Pascal Sql Useby

A read or use of an SQL entity (the entity is not modified). For example:

```
1  PROCEDURE SOMEPROC;
2  BEGIN
3     exec sql select ename
4        into :pname
5        from employee
6        where idnum = :num;
7  END;
```

Reference Kind	Entity Performing Reference	Entity Being Referenced
Pascal Sql Use	SOMEPROC (line 3) SOMEPROC (line 5) SOMEPROC (line 6)	ename employee idnum
Pascal Sql Useby	ename (line 3) employee (line 5) idnum (line 6)	SOMEPROC SOMEPROC SOMEPROC

Index

Symbols

, (comma), in filter string, 7–4
~ (tilde), in filter string, 7–4

A

abort kinds, in Ada, 7–19
abstract kinds
  in C, 7–39
  in Java, 7–79, 7–80, 7–83
accessAttrTyped kinds, in Ada, 7–19
ActivePERL, using Perl API with, 2–3
Ada
  entity kinds, 7–6
    component kinds, 7–6
    constant kinds, 7–7
    derived types and subtypes, 7–6
    entry kinds, 7–8
    enumeration literal kinds, 7–8
    exception kinds, 7–8
    file kinds, 7–9
    function kinds, 7–9
    implicit kinds, 7–10
    "local" kinds, 7–6
    object kinds, 7–10
    package kinds, 7–11
    parameter kinds, 7–12
    procedure kinds, 7–12
    protected kinds, 7–13
    task kinds, 7–14
    type kinds, 7–15
    unknown kinds, 7–18
    unresolved kinds, 7–18
  reference kinds, 7–19
    abort and abortby kinds, 7–19
    accessAttrTyped and accessAttrTypedby kinds, 7–19
    association and associationby kinds, 7–20
    call and callby kinds, 7–20
    callParamFormal and callParamFormalfor kinds, 7–22
    child and parent kinds, 7–22
    declare and declarein kinds, 7–23
    derive and derivefrom kinds, 7–25
    dot and dotby kinds, 7–26
    elaborate body and elaborate bodyby kinds, 7–27
    end and endby kinds, 7–26
    handle and handleby kinds, 7–27
    instance and instanceof kinds, 7–28
    operation and operationfor kinds, 7–29
    override and overrideby kinds, 7–29
    raise and raiseby kinds, 7–30
    ref and refby kinds, 7–30
    rename and renameby kinds, 7–31
    root and rootin kinds, 7–32
    separatefrom and separate kinds, 7–32
    set and setby kinds, 7–33
    subtype and subtypefrom kinds, 7–33
    typed and typedby kinds, 7–34
    use and useby kinds, 7–35
    usepackage and usepackageby kinds, 7–36
    usetype and usetypeby kinds, 7–36
    with and withby kinds, 7–37
alias kinds, in Pascal, 7–124
"and" relationship in filters, 7–4
anonymous class kind, in Java, 7–80
APIs
  choosing between Perl and C/C++, 1–2
  see also C API; Perl API
application. *see* Understand application
association kinds, in Ada, 7–20

B

base kinds, in C, 7–52
block kinds
  in FORTRAN, 7–62, 7–63
  in JOVIAL, 7–96, 7–100, 7–101, 7–103
build number for C API, 5–26

C

C API, 4–2
  build number for, 5–26
  code examples for
    opening database, 4–7
    reporting entities, 6–4
    reporting files, 6–6
    reporting functions, 6–7
    reporting global objects, 6–9
    reporting metrics and functions, 4–7
    reporting references, 6–13
    reporting structs, 6–11
    returning entities, 6–2
  compiling and linking with, 4–6
  database access with, 4–3
  downloading and installing, 4–2
  features of, 1–2
  include file for, 4–3
  kinds, functions for, 7–3
  libraries for, 4–5, 4–6
  license for, 4–2, 4–3, 6–3
  memory allocation, 4–2
C language
  entity kinds, 7–39
    class kinds, 7–39
    enum kinds, 7–40
    enumerator kinds, 7–41
    file kinds, 7–42
    function kinds, 7–42
    macro kinds, 7–45
    namespace kinds, 7–46
    object kinds, 7–46
    parameter kinds, 7–47
    struct kinds, 7–48
    typedef kinds, 7–49
    union kinds, 7–50
  reference kinds, 7–52
    base and derive kinds, 7–52
    call and callby kinds, 7–53
    declare and declarein kinds, 7–54
    define and definein kinds, 7–55
    end and endby kinds, 7–55
    exception kinds, 7–56
    friend and friendby kinds, 7–56
    include and includeby kinds, 7–57
    modify and modifyby kinds, 7–57
    overrides and overriddenby kinds, 7–58
    set and setby kinds, 7–58
    typed and typedby kinds, 7–59
    use and useby kinds, 7–59
call kinds
  in Ada, 7–20
  in C, 7–53
  in FORTRAN, 7–69
  in Java, 7–88, 7–89
  in JOVIAL, 7–106, 7–107
  in Pascal, 7–128, 7–133, 7–134
call ptr kinds, in FORTRAN, 7–70
callParamFormal kinds, in Ada, 7–22
Cancel dialog, disabling, 3–15
cast kinds, in Java, 7–89
catch parameter kind, in Java, 7–80
checksum of text, 3–25
child kinds, in Ada, 7–22
class kinds
  in C, 7–39
  in Java, 7–77, 7–81, 7–85
classes for Perl API, list of, 2–4
  see also specific class names
code examples. *see* examples
column kinds, in Pascal, 7–124
comma (,), in filter string, 7–4
comments
  associated with entity, 2–8, 3–6, 5–5, 5–7
  association style of, 3–4
common block kind, in FORTRAN, 7–63
common kinds, in FORTRAN, 7–62
comparison operators, for entities, 3–6
compiling with C API, 4–6
component kinds
  in Ada, 7–6
  in JOVIAL, 7–99, 7–100, 7–101, 7–103, 7–104
compool kinds, in JOVIAL, 7–96, 7–99, 7–104, 7–106, 7–108, 7–109

compool module kinds, in JOVIAL, 7–102
compunit kinds, in Pascal, 7–118
conf/license subdirectory, 2–2, 4–3
constant kinds
  in Ada, 7–7
  in JOVIAL, 7–96, 7–99, 7–100
  in Pascal, 7–118
constructor kind, in Java, 7–82
contain kinds
  in FORTRAN, 7–70
  in Java, 7–90
  in Pascal, 7–129
copy file kinds, in JOVIAL, 7–99, 7–104
copy kinds, in JOVIAL, 7–106, 7–111
couple kinds, in Java, 7–90
create kinds, in Java, 7–90
cursor kinds, in Pascal, 7–124

D

databases
  accessing with C API, 4–3
  accessing with Perl API, 2–5
  closing, 3–4, 4–4, 5–8
  current, determining, 3–15
  filename for, 3–5, 5–10
  language of, 3–4, 5–9
  maximum number open, 2–6, 5–11
  opening, 2–5, 3–2, 4–4, 4–7, 5–11, 6–2
  Pascal SQL kinds, 7–124, 7–133
  read-only access to, 1–2, 2–2, 4–2
datapool kinds, in FORTRAN, 7–62, 7–63
$db->check_comment_association() method, 3–4
$db->close() method, 2–6, 3–4
$db->ents() method, 2–6, 3–4, 7–2
$db->language() method, 3–4
$db->last_id() method, 3–4
$db->lookup() method, 2–6, 3–5, 7–2
$db->lookup_uniquename() method, 3–5
$db->metric() method, 3–5
$db->metrics() method, 2–9, 3–5
$db->name() method, 3–5
declare kinds
  in Ada, 7–23
  in C, 7–54
  in FORTRAN, 7–70, 7–76
  in JOVIAL, 7–106, 7–111, 7–112
  in Pascal, 7–129
define kinds
  in C, 7–55
  in FORTRAN, 7–71
  in Java, 7–91
  in JOVIAL, 7–106, 7–111
  in Pascal, 7–130, 7–134
derive kinds
  in Ada, 7–6, 7–25
  in C, 7–52
  in FORTRAN, 7–63
dot kinds, in Ada, 7–26
dotRef kinds, in Java, 7–91
dummy argument kinds, in FORTRAN, 7–62, 7–64

E

elaborate body kinds, in Ada, 7–27
end kinds
  in Ada, 7–26
  in C, 7–55
  in FORTRAN, 7–73
  in Java, 7–91
  in Pascal, 7–130
$ent->comments() method, 2–8, 3–6, 7–3
$ent->draw() method, 3–7
$ent->ents() method, 3–9, 7–2, 7–3
$ent->filerefs() method, 3–9, 7–2, 7–3
$ent->ib() method, 2–10, 3–10
$ent->id() method, 3–11
$ent->kind() method, 2–7, 3–11, 7–2
$ent->kindname() method, 2–7, 3–11
$ent->language() method, 3–11
$ent->lexer() method, 2–10, 3–12
$ent->library() method, 3–12
$ent->longname() method, 2–7, 3–12
$ent->metric() method, 3–12

$ent->metrics() method, 2–9, 3–13
$ent->name() method, 2–7, 3–13
$ent->parameters() method, 3–13
$ent->ref() method, 3–14, 7–2, 7–3
$ent->refs() method, 2–7, 3–13, 7–2, 7–3
$ent->relname() method, 3–14
$ent->type() method, 2–7, 3–14
$ent->uniquename() method, 3–14
entities
  attributes of, 2–7
  comments for, 2–8, 3–4, 3–6, 5–5, 5–7
  comparisons between, 3–6
  at current position, 3–16
  filtering by library, 5–57
  finding in libraries, 5–55
  graphical views of, 2–10, 3–7, 3–25, 5–13, 5–14
  Info Browser information for, 2–10, 3–10
  kind of. *see* entity kinds
  language for, 3–11, 5–18
  last (maximum) entity id in database, 3–4
  lexemes for. *see* lexeme; lexer
  library for, 3–12, 5–19
  list of, creating by lookup, 5–73
  list of, filtering by kinds, 5–63
  list of, freeing, 5–64
  looking up by reference, 5–74
  looking up by unique name, 5–23, 5–75
  metrics for, determining if defined, 5–79
  metrics for, list of, 5–82, 5–83, 5–84
  metrics for, values of, 2–9, 5–88
  name of, 3–12, 3–13, 4–4, 5–20, 5–21, 5–22, 5–23, 6–4
  performing a reference, 3–24, 5–97
  references for, 2–7, 3–13, 3–14, 4–5, 5–24, 6–13
  references for, list of, 3–9, 5–69
  references for, name of, 3–14
  returning, 4–4, 6–2
  returning for reference, 5–93
  returning list of, 2–6, 3–4, 3–5, 5–62, 6–4
  returning list of entities referenced by, 3–9
  type of, 3–14, 5–25
  unique identifier for, 3–11, 5–16
  unique name of, 3–5, 3–14
entity filters, 7–2
entity kinds
  Ada, 7–6
    component kinds, 7–6
    constant kinds, 7–7
    derived types and subtypes, 7–6
    entry kinds, 7–8
    enumeration literal kinds, 7–8
    exception kinds, 7–8
    file kinds, 7–9
    function kinds, 7–9
    implicit kinds, 7–10
    "local" kinds, 7–6
    object kinds, 7–10
    package kinds, 7–11
    parameter kinds, 7–12
    procedure kinds, 7–12
    protected kinds, 7–13
    task kinds, 7–14
    type kinds, 7–15
    unknown kinds, 7–18
    unresolved kinds, 7–18
  C, 7–39
    class kinds, 7–39
    enum kinds, 7–40
    enumerator kinds, 7–41
    file kinds, 7–42
    function kinds, 7–42
    macro kinds, 7–45
    namespace kinds, 7–46
    object kinds, 7–46
    parameter kinds, 7–47
    struct kinds, 7–48
    typedef kinds, 7–49
    union kinds, 7–50
  file. *see* file kinds
  filtering reference list by, 5–71
  FORTRAN, 7–62
    block data kinds, 7–63
    block kinds, 7–62
    block variable kinds, 7–63

common block kinds, 7–63
common kinds, 7–62
datapool kinds, 7–62, 7–63
derived type kinds, 7–63
dummy argument kinds, 7–62, 7–64
entry kinds, 7–62, 7–64
file kinds, 7–62, 7–64
function kinds, 7–62, 7–64, 7–66
include file kinds, 7–64, 7–66
interface kinds, 7–62, 7–64
intrinsic kinds, 7–65
main kinds, 7–62
main program kinds, 7–65
module kinds, 7–62, 7–65, 7–66
namelist kinds, 7–65
pointer block kinds, 7–62
pointer kinds, 7–65
subroutine kinds, 7–62, 7–66
type kinds, 7–62
unknown kinds, 7–66
unresolved kinds, 7–62, 7–66
variable kinds, 7–62, 7–67
Java, 7–77
abstract class kinds, 7–80, 7–83
abstract method kinds, 7–80
anonymous class kinds, 7–80
catch parameter kinds, 7–80
class kinds, 7–77, 7–81, 7–85
constructor kinds, 7–82
file kinds, 7–77, 7–81
final class kinds, 7–81, 7–84
final method kinds, 7–81, 7–84
final variable kinds, 7–82, 7–84
interface kinds, 7–77, 7–82
jar file kinds, 7–81
local variable kinds, 7–86
main method kinds, 7–85
method kinds, 7–78, 7–82, 7–85
package kinds, 7–78, 7–83, 7–85, 7–86
parameter kinds, 7–78, 7–83
static class kinds, 7–83, 7–84
static method kinds, 7–84
static variable kinds, 7–84, 7–85
type kinds, 7–86
unknown kinds, 7–79
unresolved kinds, 7–79
unused kinds, 7–79, 7–86
variable kinds, 7–78, 7–85, 7–86
JOVIAL, 7–96
block kinds, 7–96, 7–100, 7–101, 7–103
component kinds, 7–99, 7–100, 7–101, 7–103, 7–104
compool file kinds, 7–99, 7–104
compool kinds, 7–96
compool module kinds, 7–102
constant kinds, 7–96, 7–99, 7–100
copy file kinds, 7–99, 7–104
"external" kinds, 7–96
file kinds, 7–96, 7–99, 7–101, 7–104
function kinds, 7–102
item kinds, 7–97, 7–99, 7–100, 7–101, 7–103
"local" kinds, 7–96
macro kinds, 7–97, 7–102, 7–104
parameter kinds, 7–97, 7–102
procedure kinds, 7–102
program kinds, 7–102
statusname kinds, 7–97, 7–102
subroutine kinds, 7–97, 7–102, 7–104
table kinds, 7–97, 7–99, 7–100, 7–101, 7–103, 7–104
type kinds, 7–97, 7–103, 7–104, 7–105
unknown kinds, 7–98, 7–104
unresolved kinds, 7–98, 7–104, 7–105
variable kinds, 7–98, 7–99, 7–100, 7–101, 7–105
language of, 5–30
list of, creating, 5–66
list of, freeing, 5–67
matching, 5–27
name of, 3–11
Pascal, 7–116
alias kinds, 7–124
column kinds, 7–124
compunit kinds, 7–118

constant kinds, 7–118
cursor kinds, 7–124
enumerator kinds, 7–118
environment kinds, 7–119, 7–123
field kinds, 7–119
file kinds, 7–119
function kinds, 7–120, 7–123
"global" kinds, 7–116
group kinds, 7–125
include kinds, 7–119
index kinds, 7–125
"local" kinds, 7–116
module kinds, 7–118
parameter kinds, 7–120, 7–121
predeclared kinds, 7–121
procedure kinds, 7–120, 7–121, 7–122, 7–124, 7–125
program kinds, 7–118
role kinds, 7–125
routine kinds, 7–120, 7–121, 7–122
rule kinds, 7–125
SQL kinds, 7–124, 7–125, 7–126
statement kinds, 7–125
table kinds, 7–126
type kinds, 7–121, 7–122, 7–123
unknown kinds, 7–123
unnamed kinds, 7–122
unresolved kinds, 7–123, 7–124, 7–126
user kinds, 7–126
value kinds, 7–120
variable kinds, 7–121, 7–122, 7–123, 7–124
returning for entity, 3–11, 5–17
returning list of, 3–18, 6–4
entry kinds
in Ada, 7–8
in FORTRAN, 7–62, 7–64
enum kinds, in C, 7–40
enumeration literal kinds, in Ada, 7–8
enumerator kinds
in C, 7–41
in Pascal, 7–118
environment kinds, in Pascal, 7–119, 7–123, 7–132
equivalence kinds, in FORTRAN, 7–73
examples
C API
opening database, 4–7
reporting entities, 6–4
reporting files, 6–6
reporting functions, 6–7
reporting global objects, 6–9
reporting metrics and functions, 4–7
reporting references, 6–13
reporting structs, 6–11
returning entities, 6–2
Perl scripts, 1–2, 2–5
exception kinds
in Ada, 7–8
in C, 7–56
extend kinds, in Java, 7–92
"external" kinds, in JOVIAL, 7–96

F

field kinds, in Pascal, 7–119
file
currently being edited
column of cursor position, 3–15
entity at cursor position, 3–16
line of cursor position, 3–16
name of, 3–16
selected text in, 3–17
word at cursor position, 3–17
for project, list of, 6–6
for reference, 3–24, 5–94
references in, list of, 5–70
file jar kind, in Java, 7–81
file kinds
determining if entity kind is, 5–28
in Ada, 7–9
in C, 7–42
in FORTRAN, 7–62, 7–64, 7–66
in Java, 7–77, 7–81
in JOVIAL, 7–96, 7–99, 7–101, 7–104, 7–109

in Pascal, 7–119
looking up entities of, 5–76
returning entities of, 5–65
filename for database, 3–5, 5–10
filters. *see* entity filters; kind name filters; reference filters
final class kind, in Java, 7–81, 7–84
final kinds, in Java, 7–80
final method kind, in Java, 7–81, 7–84
final variable kind, in Java, 7–82, 7–84
FORTRAN
entity kinds, 7–62
block data, 7–63
block kinds, 7–62
block variable, 7–63
common block, 7–63
common kinds, 7–62
datapool, 7–63
datapool kinds, 7–62
derived type, 7–63
dummy argument, 7–64
dummy argument kinds, 7–62
entry, 7–64
entry kinds, 7–62
file, 7–64
file kinds, 7–62
function, 7–64, 7–66
function kinds, 7–62
include file, 7–64, 7–66
interface, 7–64
interface kinds, 7–62
intrinsic, 7–65
main kinds, 7–62
main program, 7–65
module, 7–65, 7–66
module kinds, 7–62
namelist, 7–65
pointer, 7–65
pointer block kinds, 7–62
subroutine, 7–66
subroutine kinds, 7–62
type kinds, 7–62
unknown, 7–66
unresolved, 7–66
unresolved kinds, 7–62
variable, 7–67
variable kinds, 7–62
reference kinds, 7–68
call and callby kinds, 7–69
call ptr kinds, 7–70
contain and containin kinds, 7–70
declare and declarein kinds, 7–70
define and definein kinds, 7–71
end kinds, 7–73
equivalence kinds, 7–73
include kinds, 7–74
module use kinds, 7–74
set kinds, 7–75
typed kinds, 7–75
use kinds, 7–75
use module kinds, 7–76
use rename kinds, 7–76
free() function, not using, 4–2
friend kinds, in C, 7–56
function entity, return type of, 5–25
function kinds
in Ada, 7–9
in C, 7–42
in FORTRAN, 7–62, 7–64, 7–66
in JOVIAL, 7–102
in Pascal, 7–120, 7–123
functions in C API. *see specific function names*
functions, returning, 4–7, 6–7

G

"global" kinds, in Pascal, 7–116
global objects, reporting, 6–9
graphical views
list of, 5–14
of entity, creating, 2–10, 3–7, 3–25, 5–13
group kinds, in Pascal, 7–125
GUI Understand application. *see* Understand application

Gui::active() method, 3–15
Gui::column() method, 3–15
Gui::db() method, 3–15
Gui::disable_cancel() method, 3–15
Gui::entity() method, 3–16
Gui::filename() method, 3–16
Gui::line() method, 3–16
Gui::progress_bar() method, 3–16
Gui::selection() method, 3–17
Gui::word() method, 3–17
Gui::yield() method, 3–17

H

handle kinds, in Ada, 7–27
hash value as unique id for entities, 5–16

I

implement kinds, in Java, 7–93
implicit kinds, in Ada, 7–10
import kinds, in Java, 7–93
include file for C API, 4–3
include kinds
  in C, 7–57
  in FORTRAN, 7–64, 7–66, 7–74
  in Pascal, 7–119
index kinds, in Pascal, 7–125
Info Browser, entity information from, 2–10, 3–10
inherit kinds, in Pascal, 7–131
inline kinds, in JOVIAL, 7–106
instance kinds, in Ada, 7–28
interface kinds
  in FORTRAN, 7–62, 7–64
  in Java, 7–77, 7–82
intrinsic kind, in FORTRAN, 7–65
item kinds, in JOVIAL, 7–97, 7–99, 7–100, 7–101, 7–103, 7–106, 7–109, 7–110

J

jar file kind, in Java, 7–81
Java
entity kinds, 7–77
  abstract class, 7–80, 7–83
  abstract method, 7–80
  anonymous class, 7–80
  catch parameter, 7–80
  class, 7–81, 7–85
  class kinds, 7–77
  constructor, 7–82
  file, 7–77, 7–81
  final class, 7–81, 7–84
  final method, 7–81, 7–84
  final variable, 7–82, 7–84
  interface, 7–77, 7–82
  jar file, 7–81
  local variable, 7–86
  main method, 7–85
  method, 7–78, 7–82, 7–85
  package, 7–78, 7–83, 7–85, 7–86
  parameter, 7–78, 7–83
  static class, 7–83, 7–84
  static method, 7–84
  static variable, 7–84, 7–85
  type, 7–86
  unknown kinds, 7–79
  unresolved kinds, 7–79
  unused kinds, 7–79, 7–86
  variable, 7–78, 7–85, 7–86
reference kinds, 7–87
  call kinds, 7–88, 7–89
  cast kinds, 7–89
  contain kinds, 7–90
  couple kinds, 7–90
  create kinds, 7–90
  define kinds, 7–91
  dotref kinds, 7–91
  end kinds, 7–91
  extend kinds, 7–92
  implement kinds, 7–93
  import kinds, 7–93
  modify kinds, 7–93
  override kinds, 7–94
  set kinds, 7–94

throw kinds, 7–95
typed kinds, 7–95
use kinds, 7–95
JOVIAL
entity kinds, 7–96
block, 7–96, 7–100, 7–101, 7–103
component, 7–99, 7–100, 7–101, 7–103, 7–104
compool file, 7–99, 7–104
compool kinds, 7–96
compool module, 7–102
constant, 7–96, 7–99, 7–100
copy file, 7–99, 7–104
"external" kinds, 7–96
file, 7–96, 7–99, 7–101, 7–104
function, 7–102
item, 7–97, 7–99, 7–100, 7–101, 7–103
"local" kinds, 7–96
macro, 7–97, 7–102, 7–104
parameters, 7–97, 7–102
procedure, 7–102
program, 7–102
statusname, 7–97, 7–102
subroutine, 7–97, 7–102, 7–104
table, 7–97, 7–99, 7–100, 7–101, 7–103, 7–104
type, 7–97, 7–103, 7–104, 7–105
unknown, 7–98, 7–104
unresolved, 7–98, 7–104, 7–105
variable, 7–98, 7–99, 7–100, 7–101, 7–105
reference kinds, 7–106
call kinds, 7–106, 7–107
compool kinds, 7–106, 7–108, 7–109
copy kinds, 7–106, 7–111
declare kinds, 7–106, 7–111, 7–112
define kinds, 7–106, 7–111
file kinds, 7–109
inline and inlineby kinds, 7–106
item kinds, 7–106, 7–109, 7–110
like kinds, 7–106, 7–112
overlay kinds, 7–106, 7–113
ptr kinds, 7–107
set kinds, 7–107, 7–114
typed kinds, 7–107, 7–113, 7–114
use kinds, 7–107, 7–115
value kinds, 7–107, 7–115
jpg graphics files, 2–10, 3–7

K

kind name filters, 7–4
examples of, 7–5
Perl API methods for, 7–3
Kind::list_entity() method, 3–18, 7–3
Kind::list_reference() method, 3–18, 7–3
$kind->check() method, 3–18, 7–2
$kind->inv() method, 3–18, 7–2
$kind->longname() method, 7–2
$kind->name() method, 7–2
kinds
C API functions for, 7–3
comparing to string, 3–18
filtering, 5–63, 7–2
finding in kind list, 5–34
list of, building, 5–31
list of, copying, 5–32
list of, creating, 5–36
list of, freeing, 5–33
name of, 5–35, 5–37, 7–2
Perl API methods for, 7–2
returning entity metrics by, 5–83
returning for reference, 5–95
see also entity kinds; reference kinds

L

language
determining if metric defined for, 5–80
metrics for, 3–23
of database, 3–4, 5–9
of entity, 3–11, 5–18
of kind, 5–30
returning entity metrics by, 5–84
returning project metrics by, 5–85
lexeme, 3–20, 3–22
at specified position, 3–22
beginning column of, 3–20, 5–38

beginning line of, 3–20, 5–42
ending column of, 3–20, 5–39
ending line of, 3–20, 5–43
entity associated with, 3–20, 5–40
in inactive code, determining, 3–20, 5–41
reference associated with, 3–21, 5–46
returning all lexemes in lexer, 5–52
returning array of, 3–22
returning by position in lexer, 5–51
returning first, 5–50
returning first of lexer, 3–22
returning next, 3–21, 5–44
returning previous, 3–21, 5–45
text of, 3–21, 5–47
token kind of, 3–21, 5–48
$lexeme->column_begin() method, 3–20
$lexeme->column_end() method, 3–20
$lexeme->ent() method, 2–11, 3–20
$lexeme->inactive() method, 3–20
$lexeme->line_begin() method, 3–20
$lexeme->line_end() method, 3–20
$lexeme->next() method, 2–10, 3–21
$lexeme->previous() method, 3–21
$lexeme->ref() method, 3–21
$lexeme->text method, 2–11
$lexeme->text() method, 3–21
$lexeme->token() method, 2–11, 3–21
lexer, 3–22
creating for entity, 5–54
deleting and freeing, 5–49
first lexeme in, 5–50
lexeme at specified position of, 3–22
number of lines in, 3–22, 5–53
returning, 3–12
returning all lexemes in, 5–52
returning array of lexemes in, 3–22
returning first lexeme for, 3–22
returning lexeme by position in, 5–51
$lexer->ent() method, 3–22
$lexer->first() method, 2–10
$lexer->lexeme() method, 3–22
$lexer->lexemes() method, 2–10, 3–22
$lexer->lines() method, 3–22
libraries
comparing, 5–56
filtering entities with, 5–57
finding entity in, 5–55
for C API, 4–5, 4–6
for entity, 3–12, 5–19
list of, creating, 5–58
list of, freeing, 5–59
name of, 5–60
license, 2–2
location of, specifying, 2–2, 3–2, 4–3, 6–3
required for C API, 4–2
required to open database, 5–11
like kinds, in JOVIAL, 7–106, 7–112
linking with C API, 4–6
"local" kinds
in Ada, 7–6
in JOVIAL, 7–96
in Pascal, 7–116
local variable kind, in Java, 7–86

M

macro kinds
in C, 7–45
in JOVIAL, 7–97, 7–102, 7–104
main kinds
in FORTRAN, 7–62, 7–65
in Java, 7–85
memory allocation for C API, 4–2
method kinds, in Java, 7–78, 7–82, 7–85
methods in Perl API. *see specific method names*
Metric::description() method, 3–23
Metric::list() method, 3–23, 7–2
metrics
description for, 5–78
for entities
determining if defined, 5–79
list of, 3–12, 5–82, 5–83, 5–84
returning list of, 3–13
values of, 2–9, 5–88

for language
descriptions of, 3–23
determining if defined, 5–80
returning list of, 3–23
for project, 2–9, 3–5, 4–7, 5–85, 5–89
kind of, 5–81
name of, returning by UdbMetric literal, 5–87
UdbMetric literal for, 5–86
modify kinds
in C, 7–57
in Java, 7–93
module kinds
in FORTRAN, 7–62, 7–65, 7–66
in Pascal, 7–118

N

namelist kind, in FORTRAN, 7–65
namespace kinds, in C, 7–46
NULL library, 4–5

O

object kinds
in Ada, 7–10
in C, 7–46
operation kinds, in Ada, 7–29
"or" relationship in filters, 7–4
overlay kinds, in JOVIAL, 7–106, 7–113
override kinds
in Ada, 7–29
in C, 7–58
in Java, 7–94

P

package kinds
in Ada, 7–11
in Java, 7–78, 7–83, 7–85, 7–86
parameter kinds
in Ada, 7–12
in C, 7–47
in Java, 7–78, 7–83
in JOVIAL, 7–97, 7–102
in Pascal, 7–120, 7–121
parameter, type of, 5–25
parent kinds, in Ada, 7–22
Pascal
entity kinds, 7–116
alias, 7–124
column, 7–124
compunit, 7–118
constant, 7–118
cursor, 7–124
enumerator, 7–118
environment, 7–119, 7–123
field, 7–119
file, 7–119
function, 7–120, 7–123
"global" kinds, 7–116
group, 7–125
include, 7–119
index, 7–125
"local" kinds, 7–116
module, 7–118
parameter, 7–120, 7–121
predeclared, 7–121
procedure, 7–120, 7–121, 7–122, 7–124, 7–125
program, 7–118
role, 7–125
routine, 7–120, 7–121, 7–122
rule, 7–125
SQL, 7–124, 7–125, 7–126
statement, 7–125
table, 7–126
type, 7–121, 7–122, 7–123
unknown, 7–123
unnamed, 7–122
unresolved, 7–123, 7–124, 7–126
user, 7–126
value, 7–120
variable, 7–121, 7–122, 7–123, 7–124
reference kinds, 7–127
call kinds, 7–128, 7–133, 7–134

contain kinds, 7–129
declare kinds, 7–129
define kinds, 7–130, 7–134
end kinds, 7–130
environment kinds, 7–132
inherit kinds, 7–131
set kinds, 7–132, 7–135
SQL kinds, 7–133, 7–134, 7–135
typed kinds, 7–133, 7–135
use kinds, 7–133, 7–135
Perl API
advantages of, 1–2
classes provided with, 2–4
database access with, 2–5
example scripts for, 1–2, 2–5
features of, 1–2
kinds, methods for, 7–2
running scripts with uperl program, 2–3
Understand package for, 2–2
using with existing Perl installations, 2–3
version of Perl for, 2–2
png graphics files, 2–10, 3–7
pointer block kinds, in FORTRAN, 7–62
pointer kind, in FORTRAN, 7–65
predeclared kinds, in Pascal, 7–121
private kinds
in C, 7–40
in Java, 7–79
procedure kinds
in Ada, 7–12
in JOVIAL, 7–102
in Pascal, 7–120, 7–121, 7–122, 7–124, 7–125
program kinds
in JOVIAL, 7–102
in Pascal, 7–118
programming language. *see* language
progress bar, displaying, 3–16
project
files in, 6–6
metrics for, 2–9, 3–5, 4–7, 5–85, 5–89
protected kinds
in Ada, 7–13
in C, 7–40
in Java, 7–79
ptr kinds, in JOVIAL, 7–107
public kinds
in C, 7–40
in Java, 7–79

R

raise kinds, in Ada, 7–30
read-only access to database, 1–2, 2–2, 4–2
ref kinds, in Ada, 7–30
$ref->column() method, 3–23
$ref->ent() method, 3–23
$ref->file() method, 3–24
$ref->kind() method, 3–24
$ref->kindname() method, 3–24
$ref->line() method, 3–24
$ref->scope() method, 3–24
reference filters, 7–2
reference kinds
Ada, 7–19
abort and abortby kinds, 7–19
accessAttrTyped and accessAttrTypedby kinds, 7–19
association and associationby kinds, 7–20
call and callby kinds, 7–20
callParamFormal and callParamFormalfor kinds, 7–22
child and parent kinds, 7–22
declare and declarein kinds, 7–23
derive and derivefrom kinds, 7–25
dot and dotby kinds, 7–26
elaborate body and elaborate bodyby kinds, 7–27
end and endby kinds, 7–26
handle and handleby kinds, 7–27
instance and instanceof kinds, 7–28
operation and operationfor kinds, 7–29
override and overrideby kinds, 7–29
raise and raiseby kinds, 7–30
ref and refby kinds, 7–30

rename and renameby kinds, 7–31
root and rootin kinds, 7–32
separatefrom and separate kinds, 7–32
set and setby kinds, 7–33
subtype and subtypefrom kinds, 7–33
typed and typedby kinds, 7–34
use and useby kinds, 7–35
usepackage and usepackageby kinds, 7–36
usetype and usetypeby kinds, 7–36
with and withby kinds, 7–37
C
base and derive kinds, 7–52
call and callby kinds, 7–53
declare and declarein kinds, 7–54
define and definein kinds, 7–55
end and endby kinds, 7–55
exception kinds, 7–56
friend and friendby kinds, 7–56
include and includeby kinds, 7–57
modify and modifyby kinds, 7–57
overrides and overriddenby kinds, 7–58
set and setby kinds, 7–58
typed and typedby kinds, 7–59
use and useby kinds, 7–59
C language, 7–52
filtering reference list by, 5–71
FORTRAN, 7–68
call and callby kinds, 7–69
call ptr kinds, 7–70
contain and containin kinds, 7–70
declare and declarein kinds, 7–70
define and definein kinds, 7–71
end kinds, 7–73
equivalence kinds, 7–73
include kinds, 7–74
module use kinds, 7–74
set kinds, 7–75
typed kinds, 7–75
use kinds, 7–75
use rename kinds, 7–76
inverse of, 5–29, 7–2
Java, 7–87
call kinds, 7–88, 7–89
cast kinds, 7–89
contain kinds, 7–90
couple kinds, 7–90
create kinds, 7–90
define kinds, 7–91
dotref kinds, 7–91
end kinds, 7–91
extend kinds, 7–92
implement kinds, 7–93
import kinds, 7–93
modify kinds, 7–93
override kinds, 7–94
set kinds, 7–94
throw kinds, 7–95
typed kinds, 7–95
use kinds, 7–95
JOVIAL, 7–106
call kinds, 7–106, 7–107
compool kinds, 7–106, 7–108, 7–109
copy kinds, 7–106, 7–111
declare kinds, 7–106, 7–111, 7–112
define kinds, 7–106, 7–111
file kinds, 7–109
inline and inlineby kinds, 7–106
item kinds, 7–106, 7–109, 7–110
like kinds, 7–106, 7–112
overlay kinds, 7–106, 7–113
ptr kinds, 7–107
set kinds, 7–107, 7–114
typed kinds, 7–107, 7–113, 7–114
use kinds, 7–107, 7–115
value kinds, 7–107, 7–115
language of, 5–30
list of, creating, 5–68
list of, freeing, 5–67
logical inverse of, 3–18
looking up references by, 5–77
matching, 5–27
name of, 3–24
Pascal, 7–127

call kinds, 7–128, 7–133, 7–134
contain kinds, 7–129
declare kinds, 7–129
define kinds, 7–130, 7–134
end kinds, 7–130
environment kinds, 7–132
inherit kinds, 7–131
set kinds, 7–132, 7–135
SQL kinds, 7–133, 7–134, 7–135
typed kinds, 7–133, 7–135
use kinds, 7–133, 7–135
returning, 3–24, 5–95
returning list of, 3–18
references
column position of, 3–23, 5–90
copy of, creating, 5–91
copy of, freeing, 5–92
entity performing, 3–24, 5–97
entity referenced by, 3–23
file for, 3–24, 5–94
for entity, 2–7, 3–13, 3–14, 4–5, 5–24, 5–93, 6–13
for entity, list of, 3–9, 5–69
for entity, name of, 3–14
for lexeme, 3–21, 5–46
in file, list of, 5–70
kind of. *see* reference kinds
line of, 3–24, 5–96
list of, filtering by kind, 5–71
list of, freeing, 5–72
looking up by reference kinds, 5–77
looking up entity by, 5–74
rename kinds, in Ada, 7–31
role kinds, in Pascal, 7–125
root kinds, in Ada, 7–32
routine kinds, in Pascal, 7–120, 7–121, 7–122
rule kinds, in Pascal, 7–125

S

Scientific Toolworks sample scripts, 1–2
scripts
called from application, 3–15
Perl examples for, 1–2, 2–5
running with other Perl installations, 2–3
running with uperl program, 2–3
use command required for, 2–4, 2–5, 3–2
separate kinds, in Ada, 7–32
set kinds
in Ada, 7–33
in C, 7–58
in FORTRAN, 7–75
in Java, 7–94
in JOVIAL, 7–107, 7–114
in Pascal, 7–132, 7–135
SQL kinds
in Pascal, 7–124, 7–125, 7–126, 7–133, 7–134, 7–135
standard library, 4–5
statement kinds, in Pascal, 7–125
static class kinds, in Java, 7–83, 7–84
static kinds, in Java, 7–80
static method kind, in Java, 7–84
static variable kind, in Java, 7–84, 7–85
status kinds, in JOVIAL, 7–97
statusname kinds, in JOVIAL, 7–102
$STIHOME environment variable, 4–3
$STILICENSE environment variable, 2–2, 4–3
struct kinds, in C, 7–48
structs, returning, 6–11
subroutine kinds
in FORTRAN, 7–62, 7–66
in JOVIAL, 7–97, 7–102, 7–104
subtype kinds, in Ada, 7–33
subtypes, entity kinds in Ada for, 7–6

T

table kinds
in JOVIAL, 7–97, 7–99, 7–100, 7–101, 7–103, 7–104
in Pascal, 7–126
task kinds, in Ada, 7–14
text, checksum of, 3–25
throw kinds, in Java, 7–95

tilde (~), in filter string, 7–4
token kind for lexeme, 3–21, 5–48
type kinds
  in Ada, 7–15
  in FORTRAN, 7–62
  in Java, 7–86
  in JOVIAL, 7–97, 7–103, 7–104, 7–105
  in Pascal, 7–121, 7–122, 7–123
typed kinds
  in Ada, 7–34
  in C, 7–59
  in FORTRAN, 7–75
  in Java, 7–95
  in JOVIAL, 7–107, 7–113, 7–114
  in Pascal, 7–133, 7–135
typedef kinds, in C, 7–49

U

udb.h file, 4–3, 6–3
udb_api.a library, 4–6
udb_api.lib library, 4–6
udb_api.obj library, 4–6
udb_api.sl library, 4–6
udb_api.so library, 4–6
udbComment() function, 5–5
udbCommentRaw() function, 5–7
udbDbClose() function, 4–4, 5–8
  examples of, 4–8, 6–3
udbDbLanguage() function, 5–9
udbDbName() function, 5–10
udbDbOpen() function, 4–4, 5–11
  examples of, 4–7, 6–2
  failing if license unavailable, 4–4, 5–11
udbEntityDraw() function, 5–13
udbEntityId() function, 5–16
udbEntityKind() function, 5–17, 7–3
  examples of, 6–13
udbEntityLanguage() function, 5–18
udbEntityLibrary() function, 5–19
udbEntityNameLong() function, 4–4, 5–20
  examples of, 4–8, 6–6, 6–9, 6–11, 6–13
udbEntityNameShort() function, 4–4, 5–21
  examples of, 4–8, 6–4, 6–7, 6–9, 6–11, 6–13
udbEntityNameSimple() function, 5–22
udbEntityNameUnique() function, 5–23
udbEntityRefs() function, 5–24
udbEntityTypetext() function, 5–25
  examples of, 6–7, 6–11
udbInfoBuild() function, 5–26
udbIsKind() function, 5–27, 7–3
udbIsKindFile() function, 5–28
udbKindInverse() function, 5–29, 7–3
udbKindLanguage() function, 5–30
udbKindList() function, 5–31, 7–3
udbKindListCopy() function, 5–32
udbKindListFree() function, 5–33
udbKindLocate() function, 5–34, 7–3
udbKindLongname() function, 5–35
udbKindParse() function, 5–36, 7–3
  examples of, 4–8, 6–7, 6–9, 6–11, 6–13
udbKindShortname() function, 5–37
  examples of, 4–8, 6–4, 6–9, 6–13
udbLexemeColumnBegin() function, 5–38
udbLexemeColumnEnd() function, 5–39
udbLexemeEntity() function, 5–40
udbLexemeInactive() function, 5–41
udbLexemeLineBegin() function, 5–42
udbLexemeLineEnd() function, 5–43
udbLexemeNext() function, 5–44
udbLexemePrevious() function, 5–45
udbLexemeReference() function, 5–46
udbLexemeText() function, 5–47
udbLexemeToken() function, 5–48
udbLexerDelete() function, 5–49
udbLexerFirst() function, 5–50
udbLexerLexeme() function, 5–51
udbLexerLexemes() function, 5–52
udbLexerLines() function, 5–53
udbLexerNew() function, 5–54
udbLibraryCheckEntity() function, 5–55
udbLibraryCompare() function, 5–56
udbLibraryFilterEntity() function, 5–57
udbLibraryList() function, 5–58
udbLibraryListFree() function, 5–59

udbLibraryName() function, 5–60
udbListEntity() function, 4–4, 5–62
  examples of, 4–8, 6–4, 6–7, 6–9, 6–11
udbListEntityFilter() function, 4–4, 5–63, 7–3
  examples of, 4–8, 6–7, 6–9, 6–11
udbListEntityFree() function, 4–4, 5–64
  examples of, 4–8, 6–4, 6–6, 6–7, 6–9, 6–11
udbListFile() function, 5–65
  examples of, 6–6
udbListKindEntity() function, 5–66
udbListKindFree() function, 5–67
udbListKindReference() function, 5–68
udbListReference() function, 4–5, 5–69
  examples of, 4–8, 6–7, 6–9, 6–11, 6–13
udbListReferenceFile() function, 5–70
udbListReferenceFilter() function, 4–5, 5–71, 7–3
  examples of, 4–8, 6–7, 6–11, 6–13
udbListReferenceFree() function, 4–5, 5–72
  examples of, 6–7, 6–9, 6–11, 6–13
udbLookupEntity() function, 5–73
udbLookupEntityByReference() function, 5–74
udbLookupEntityByUniquename() function, 5–75
udbLookupFile() function, 5–76
udbLookupReferenceExists() function, 5–77
UdbMetric literal, 5–86, 5–87
udbMetricDescription() function, 5–78
udbMetricIsDefinedEntity() function, 5–79
udbMetricIsDefinedProject() function, 5–80
udbMetricKind() function, 5–81
udbMetricListEntity() function, 5–82
udbMetricListKind() function, 5–83
udbMetricListLanguage() function, 5–84
udbMetricListProject() function, 5–85
udbMetricLookup() function, 5–86
udbMetricName() function, 5–87
udbMetricValue() function, 5–88
udbMetricValueProject() function, 5–89
udbReferenceColumn() function, 5–90
udbReferenceCopy() function, 5–91
udbReferenceCopyFree() function, 5–92
udbReferenceEntity() function, 5–93
  examples of, 4–8, 6–7, 6–9, 6–11, 6–13
udbReferenceFile() function, 5–94
  examples of, 4–8, 6–9, 6–13
udbReferenceKind() function, 5–95, 7–3
  examples of, 4–8, 6–9, 6–13
udbReferenceLine() function, 5–96
  examples of, 4–8, 6–9, 6–13
udbReferenceScope() function, 5–97
udbSetLicense() function, 4–3, 6–3
Understand application
  Cancel dialog in, disabling, 3–15
  displaying progress bar in, 3–16
  scripts called from, determining, 3–15
  yield event in, causing, 3–17
Understand package, 2–2
  classes provided with, 2–4
  license for, 2–2
  use command required for, 3–2
  version of, 3–3
Understand::Db class
  $db->check_comment_association() method, 3–4
  $db->close() method, 3–4
  $db->ents() method, 2–6, 3–4, 7–2
  $db->language() method, 3–4
  $db->last_id() method, 3–4
  $db->lookup() method, 2–6, 3–5, 7–2
  $db->lookup_uniquename() method, 3–5
  $db->metric() method, 3–5
  $db->metrics() method, 2–9, 3–5
  $db->name() method, 3–5
  returning object of, 3–2, 3–4
Understand::Ent class
  $ent->comments() method, 2–8, 3–6, 7–3
  $ent->draw() method, 2–10, 3–7
  $ent->ents() method, 3–9, 7–2, 7–3
  $ent->filerefs() method, 3–9, 7–2, 7–3
  $ent->ib() method, 2–10, 3–10
  $ent->id() method, 3–11
  $ent->kind() method, 2–7, 3–11, 7–2

$ent->kindname() method, 2–7, 3–11
$ent->language() method, 3–11
$ent->lexer() method, 2–10, 3–12
$ent->library() method, 3–12
$ent->longname() method, 2–7, 3–12
$ent->metric() method, 3–12
$ent->metrics() method, 2–9, 3–13
$ent->name() method, 2–7, 3–13
$ent->parameters() method, 3–13
$ent->ref() method, 3–14, 7–2, 7–3
$ent->refs() method, 2–7, 3–13, 7–2, 7–3
$ent->relname() method, 3–14
$ent->type() method, 2–7, 3–14
$ent->uniquename() method, 3–14
returning objects of, 3–4, 3–6, 3–9, 3–23
Understand::Gui class, 2–11, 3–15
Gui::active() method, 3–15
Gui::column() method, 3–15
Gui::db() method, 3–15
Gui::disable_cancel() method, 3–15
Gui::entity() method, 3–16
Gui::filename() method, 3–16
Gui::line() method, 3–16
Gui::progress_bar() method, 3–16
Gui::selection() method, 3–17
Gui::word() method, 3–17
Gui::yield() method, 3–17
Understand::Kind class
Kind::list_entity() method, 3–18
Kind::list_reference() method, 3–18
$kind->check() method, 3–18, 7–2
$kind->inv() method, 3–18, 7–2
$kind->longname() method, 7–2
$kind->name() method, 7–2
returning objects of, 3–11, 3–18, 3–24, 7–2
Understand::Lexeme class, 2–10
$lexeme->column_begin() method, 3–20
$lexeme->column_end() method, 3–20
$lexeme->ent() method, 2–11, 3–20
$lexeme->inactive() method, 3–20
$lexeme->line_begin() method, 3–20
$lexeme->line_end() method, 3–20
$lexeme->next() method, 2–10, 3–21
$lexeme->previous() method, 3–21
$lexeme->ref() method, 3–21
$lexeme->text method, 2–11
$lexeme->text() method, 3–21
$lexeme->token() method, 2–11, 3–21
returning objects of, 3–20, 3–22
Understand::Lexer class, 2–10
$lexer->ent() method, 3–22
$lexer->first() method, 2–10
$lexer->lexeme() method, 3–22
$lexer->lexemes() method, 2–10, 3–22
$lexer->lines() method, 3–22
returning objects of, 3–12, 3–20, 3–22
Understand::license() method, 2–2, 3–2
Understand::Metric class
Metric::description() method, 3–23
Metric::list() method, 3–23
Understand::open() method, 2–5, 3–2
Understand::Ref class
$ref->column() method, 3–23
$ref->ent() method, 3–23
$ref->file() method, 3–24
$ref->kind() method, 3–24
$ref->kindname() method, 3–24
$ref->line() method, 3–24
$ref->scope() method, 3–24
returning objects of, 2–7, 3–9, 3–13, 3–14, 3–21, 3–23
Understand::Util class
Util::checksum() method, 3–25
Understand::version() method, 3–3
Understand::Visio class
Visio::draw() method, 3–25
Visio::quit() method, 3–25
union kinds, in C, 7–50
unique name
looking up entity by, 5–23, 5–75
unknown kinds
in Ada, 7–18
in FORTRAN, 7–66
in Java, 7–79, 7–85

in JOVIAL, 7–98, 7–104
in Pascal, 7–123
unnamed kinds, in Pascal, 7–122
unresolved kinds
in Ada, 7–18
in FORTRAN, 7–62, 7–66
in Java, 7–79, 7–85, 7–86
in JOVIAL, 7–98, 7–104, 7–105
in Pascal, 7–123, 7–124, 7–126
unused kinds, in Java, 7–79, 7–86
uperl program, 2–3
use command, 2–4, 2–5, 3–2
use kinds
in Ada, 7–35
in C, 7–59
in FORTRAN, 7–74, 7–75
in Java, 7–95
in JOVIAL, 7–107, 7–115
in Pascal, 7–133, 7–135
use module kinds, in FORTRAN, 7–76
use rename kinds, in FORTRAN, 7–76
usepackage kinds, in Ada, 7–36
user kinds, in Pascal, 7–126
usetype kinds, in Ada, 7–36
Util::checksum() method, 3–25

V

value kinds
in JOVIAL, 7–107, 7–115
in Pascal, 7–120
variable kind, in Java, 7–86
variable kinds
in FORTRAN, 7–62, 7–67
in Java, 7–78, 7–85, 7–86
in JOVIAL, 7–98, 7–99, 7–100, 7–101, 7–105
in Pascal, 7–121, 7–122, 7–123, 7–124
version requirements, 2–2, 3–3
views for entities. *see* graphical views
Visio
generating graphic for entity with, 3–7, 3–25
Perl API integration with, 1–2
quitting, 3–25
Visio::draw() method, 3–25
Visio::quit() method, 3–25
vsd file, generating, 3–25
vsd graphics files, 2–10

W

web sites
C API download, 4–2
Perl script examples, 1–2, 2–5
with kinds, in Ada, 7–37

Y

yield event, causing, 3–17