Inside the PGPsdk

Version 1.1

October 30, 1997

9-97. Printed in the United States of America.
PGPsdk, Version 1.1

Record the serial number from your ***License Agreement*** in the space provided below:

The software provided with this documentation is licensed to you for your individual use under the terms of the End User License Agreement and Limited Warranty provided with the software. The information in this document is subject to change without notice. Pretty Good Privacy, Inc. does not warrant that the information meets you requirements or that the information is free of errors. The information may include technical inaccuracies or typographical errors. Changes may be made to the information and incorporated in new editions of this document, if and when made available by Pretty Good Privacy, Inc.

Export of this software and documentation may be subject to compliance with the rules and regulations promulgated from time to time by the Bureau of Export Administration, United States Department of Commerce, which restrict the export and re-export of certain products and technical data.

Pretty Good Privacy, Inc. (650) 631-1747 voice
2121 South El Camino Real (650) 572-1932 fax
Suite 902
San Mateo, CA 94403

Document last revised: ***1997.12.02***

LIMITED WARRANTY

Limited Warranty. Pretty Good Privacy, Inc. (“PGP”) warrants that the Software Product will perform substantially in accordance with the accompanying written materials for a period of sixty (60) days from the date of original purchase. To the extent allowed by applicable law, implied warranties on the Software Product, if any, are limited to such sixty (60) day period. Some jurisdictions do not allow limitations on duration of an implied warranty, so the above limitation may not apply to you.

Customer Remedies. PGP's and its suppliers’ entire liability and your exclusive remedy shall be, at PGP's option, either (a) return of the purchase price paid for the license, if any or (b) repair or replacement of the Software Product that does not meet PGP's limited warranty and which is returned at your expense to PGP with a copy of your receipt. This limited warranty is void if failure of the Software Product has resulted from accident, abuse, or misapplication. Any repaired or replacement Software Product will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer. Outside the United States, neither these remedies nor any product support services offered by PGP are available without proof of purchase from an authorized international source and may not be available from PGP to the extent they subject to restrictions under U.S. export control laws and regulations.

NO OTHER WARRANTIES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, AND EXCEPT FOR THE LIMITED WARRANTIES SET FORTH HEREIN, THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" AND PGP AND ITS SUPPLIERS DISCLAIM ALL OTHER WARRANTIES AND CONDITIONS, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, CONFORMANCE WITH DESCRIPTION, TITLE AND NON-INFRINGEMENT OF THIRD PARTY RIGHTS, AND THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES. THIS LIMITED WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS. YOU MAY HAVE OTHERS, WHICH VARY FROM JURISDICTION TO JURISDICTION.

LIMITATION OF LIABILITY. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL PGP OR ITS SUPPLIERS BE LIABLE FOR ANY INDIRECT, INCIDENTAL, CONSEQUENTIAL, SPECIAL OR EXEMPLARY DAMAGES OR LOST PROFITS WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE PRODUCT OR THE FAILURE TO PROVIDE SUPPORT SERVICES, EVEN IF PGP HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY CASE, PGP'S CUMULATIVE AND ENTIRE LIABILITY TO YOU OR ANY OTHER PARTY FOR ANY LOSS OR DAMAGES RESULTING FROM ANY CLAIMS, DEMANDS OR ACTIONS ARISING OUT OF OR RELATING TO THIS AGREEMENT SHALL NOT EXCEED THE PURCHASE PRICE PAID FOR THIS LICENSE. BECAUSE SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY, THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU.

Table of Contents

Figures

Tables

Preface

Inside the PGPsdk is the reference manual for the PGP Software Development Kit (PGPsdk), Version 1.1. This initial release of the PGPsdk provides developers the functionality to readily add the PGP peer-reviewed cryptographic technology to their own applications. Because this is a reference manual, only a minimum of introductory or tutorial material is presented.

By using the PGPsdk as a part of your development effort, you can

- develop products that are as secure as *PGP for Business Security Version 5.5* (and optionally interoperating with it, where appropriate)
- easily develop, maintain, and use PGP cryptographic components in your application
- provide yourself and your customers with the confidence that comes from using the PGP trusted and peer-reviewed technology in your security protocols

The engineers at **Pretty Good Privacy, Inc.**, have used the identical PGPsdk supplied to external developers to produce *PGP for Business Security, Version 5.5*. Numerous excerpts from a sample application representing a greatly simplified version of *PGP for Business Security, Version 5.5* are included in this manual. In keeping with the PGP corporate policy of complete and open publication of source code for peer review, the final *PGP for Business Security, Version 5.5 Source Code* books (when available) will serve as the essential and definitive reference for developers using the PGPsdk for their own application development.

Audience

This book is written for experienced software engineers and application developers who need to incorporate the PGP cryptographic functionality in their application, or are developing a product that needs to communicate with other applications that create or understand PGP-encrypted or cryptographically signed data. Since the initial release of the PGPsdk supports a *C* language Application Programming Interface (API), you should have *C* language experience to use this product.

If you are not familiar with basic cryptographic concepts, PGP recommends that you read *Applied Cryptography, Second Edition*, by Bruce Schneier (John Wiley & Sons, Inc., 1996). This volume is arguably the best introduction and general reference to cryptography currently available to the public. For additional readings on **cryptography** and cryptographic theory, see the short list of recommended readings at the end of this chapter, or the more extensive list in Appendix C, “References and Recommended Reading.”

Manual Organization

Inside the PGPsdk presents the PGP cryptographic functionality in a manner that corresponds to the organization of the PGPsdk Software Library. Several overview chapters appear first, and detail the basic concepts, organization, and functional divisions of the PGPsdk.

Following the overview chapters are detailed reference chapters for each functional division of the PGPsdk, which contain detailed descriptions of the functions in each functional division. The reference chapters include

- an introductory overview of the functional division
- a list of the names of the associated *C* language header files
- Tables containing `#define` and enumerated type constants and their descriptions
- *C* language code fragments for any associated datatypes and structures
- a logical ordering of the events and/or functions within the functional division

Each event description includes

- an explanation of the event
- the data type and structures passed to/from the event
- the allowed `PGPO[ption]` values (if any)

Each function description includes

- the function's *C* language prototype
- argument descriptions
- an explanation of the function
- optional notable error codes
- optional notes, warnings, and tips on using the function
- optional sample code

The manual contains appendixes detailing

- error codes
- recommended readings in cryptography

The manual concludes with

- a glossary of cryptographic terms
- an index

Conventions Used in This Document

Typographic Conventions

C language code listings, reserved words, and names of data structures, fields, constants, arguments, and functions are shown in `Courier Font`.

Key terms or concepts appear in **boldface**, and are defined in the Glossary.

Notes, Warnings, and Tips Conventions

Notes may contain:

- non-essential but useful and/or interesting information
- information that is essential for understanding the material presented

Warnings contain information that is essential to understand. Failure to do so could result in crashes and/or loss of data.

Tips contain information specifically intended to aid the PGPsdk developer in using the function to the best advantage.

Development Environment and API Platform Support

The PGPsdk, Version 1.1 binaries and public header files are supported on three major platforms: Unix, 32-bit Windows, and Macintosh. While platforms and compilers other than those listed below may work with the PGPsdk (and some will be supported in future releases), the Version 1.1 release has only been verified as working with the following:

- Unix platform and compiler support includes Solaris for Sparc, Linux x86, OpenBSD x86, and NetBSD x86 environments, each using the GNU *C* compiler.
- 32-bit Windows platform and compiler support includes those 32-bit environments using the Microsoft Visual C++ 5.0 compiler

- MacOS platform and compiler support includes MacOS Version 7.6 environments using the MetroWerks CodeWarrior Version 12.

Related Documentation

PGP for Business Security, Version 5.5 Users Guide for Mac and Windows

PGP Security Officer's Guide, Version 5.5

Recommended Readings in Cryptography and Cryptographic Theory

Applied Cryptography, Second Edition, by Bruce Schneier, 1996, John Wiley & Sons, Inc.

Dr. Dobbs Essential Books on Cryptography and Security CD-ROM, (includes searchable version of Schneier, and several other useful books and papers), 1997, Dr. Dobbs CD-ROM Library.

PGP 5.0 Platform Independent Source Code - Five Volumes, Philip Zimmermann and Mark Weaver, eds., Warthman Associates, 1997.

PGP 5.0 Win95 Source Code - Three Volumes, Philip Zimmermann and Mark Weaver, eds., Warthman Associates, 1997.

PGP 5.0 Mac Source Code - Four Volumes, Philip Zimmermann and Mark Weaver, eds., Warthman Associates, 1997.

Chapter 1: Introduction to the PGP Software Development Kit

Overview of the PGPsdk

The PGPsdk consists of nine functional groups including, among others, key management functions, high- and low-level cryptographic functions, and pseudo-random number generation functions. Each group has a separately-compilable public header file that allows developers to include only the PGP cryptographic functionality that they want to impart to their applications. The more closely related header files are further grouped into six major functional areas. Each of these major functional areas is documented in a separate chapter in the Function Reference section of this document (Chapter 2 through Chapter 8).

Table 1-1: Organization of Public Header Files in This Document

Header File	Chapter
`pgpKeys.h`	Chapter 2: Key Management Functions
`pgpCBC.h`	Chapter 3: Ciphering and Authentication Functions
`pgpCFB.h`	
`pgpEncode.h`	
`pgpHash.h`	
`pgpPublicKey.h`	
`pgpSymmetricCipher.h`	
`pgpEncode.h`	Chapter 4: Option List Functions
`pgpRandomPool`	Chapter 5: Random Number Generation Functions
`pgpPubTypes.h`	Chapter 6: Utility Toolbox
`pgpSDKPrefs.h`	
`pgpUtilities.h`	
`pgpFeatures.h`	Chapter 7: Feature (Capability) Query Functions
`pgpKeyServer.h`	Chapter 8: Key Server Functions
`pgpKeyServerTypes.h`	
`pgpErrors.h`	Appendix A: PGPsdk Error Summary

Here are summaries of the six chapters in the function reference section of this book:

- Chapter 2: Function Reference - Key Management Functions. Key management functions allow applications to create, sign, add, remove, search for, and check the validity of keys on disk-based or in-memory key rings. Also found here are functions to check and set property values for keys, according to the PGP **Web of Trust** model. The key management function prototypes are listed in the public header file `pgpKeys.h`.
- Chapter 3: Function Reference - Ciphering and Authentication Functions. Algorithm-independent functions are provided for high-level cryptographic functions such as encrypting, decrypting, hashing, signing, and verifying messages. Not only are applications free of the details of the particular algorithms being used, but also new algorithms can be incorporated transparently as they become available. The high-level cryptographic function prototypes are listed in the public header file `pgpEncode.h`. The low-level cryptographic function prototypes are listed in the public header files `pgpCBC.h`, `pgpCFB.h`, `pgpHash.h`, and `pgpSymmetricCipher.h`, which appear as `#include` directives in `pgpEncode.h`.
- Chapter 4: Function Reference – Option List Functions. Option list functions provide a flexible and extensible mechanism for presenting arbitrary option specifications and data to functions accepting this mechanism. Option lists may be persistent or local to the function accepting them, and so support modular establishment and combining of option groups. Since the option list mechanism

was originally developed for encode/decode operations, the function prototypes are listed in the public header file `pgpEncode.h`.

- Chapter 5: Function Reference - Random Number Generation Functions. The cryptographic functions employed by the PGPsdk require random numbers to operate correctly. The PGPsdk provides pseudo-random number generation functions, along with functions to manage random numbers seeded from mouse movements, keystrokes, and other events. The random number generation function prototypes are listed in the public header file `pgpRandomPool.h`.
- Chapter 6: Function Reference - Utility Toolbox. Sections of the PGPsdk require miscellaneous utility functions such as context creation, memory management, file management, and date/time functions. These utility function prototypes are listed in the PGPsdk public header file `pgpUtilities.h`. Additionally, this Chapter 6ocuments a translation function that converts `PGPError` numeric codes to English language character strings. This utility function prototypes are listed in the PGPsdk public header file `pgpUtilities.h`.
- Chapter 7: Function Reference - Feature (Capability) Query Functions. The present state of U.S. export law and the continuously evolving set of cryptographic standards, algorithms, and formats, make the existence of multiple versions of the PGPsdk a very real possibility. For example, a version intended for export may support signing but not encryption. The PGPsdk includes functions that return version numbers and the availability of specific features (capabilities). These query function prototypes are listed in the public header file `pgpFeatures.h`.
- Chapter 8: Function Reference – Key Server Functions. The PGPsdk includes functions to facilitate communicating with both **HTTP** and **LDAP** key servers. These key server function prototypes are listed in the public header file `pgpKeyServer.h`.

PGPsdk Functionality

The PGP Software Development Kit (PGPsdk) allows software engineers and application developers to seamlessly incorporate the PGP cryptographic technology into such applications as e-mail package plug-ins, secure electronic interchange packages, and secure financial transaction packages. The PGP cryptographic technology consists of the following three basic cryptographic elements

- **key management**
- ciphering (**encryption**/**decryption**)
- **authentication** (signing and verifying)

Key management functions are used to

- create and/or add **keys**
- remove keys
- search for keys meeting certain ownership and/or property criteria
- check the validity of disk-based or in-memory key rings
- check and/or set key property values

Ciphering (encrypting/decrypting) functions are used to

- encrypt data or files
- decrypt data or files

Authentication (signing and verifying) functions are used to

- sign messages or data files
- verify the authenticity of messages or data files

Other functional areas include **pseudo-random** number generation, utility, and query functions that

- manage pseudo-random numbers seeded from mouse movements, keystrokes, and other events
- manage memory
- specify files
- effect date/time conversion (platform dependent)
- convert error codes to readable strings

- indicate the availability of specific features within the PGPsdk

The Application Programmer's Interface (**API**) to the PGPsdk consists of *C* language functions, and provides developers with a consistent interface and error handling protocols. These functions are organized into functional groups, and each group comprises a function reference chapter of this document (Chapter 2 through Chapter 8). Each of these chapters includes

- an overview of the functional group
- a logical ordering of the functions within the group (as applicable)
- the function group's associated header file(s)
- a full description of each individual function

The full description of each function includes

- a brief description of the function
- the function's *C* language prototype
- argument descriptions
- noteworthy error codes
- tips and notes on using the function
- sample code (as required)

To use the PGPsdk, simply incorporate calls to the PGPsdk functions into your *C* language application following the function prototypes listed in the public header files supplied as part of the PGPsdk and including the necessary header files, and then link with the supplied PGPsdk library binaries. Two versions of the PGPsdk library binaries are supplied: a debug version and a non-debug version. Both versions perform essentially the same error checking, and report the same error return codes. The debug version additionally asserts itself on error conditions, and reports the errors to the default output destination (platform dependent).

Library Binaries

The PGPsdk library binaries contain all of the functions described by the header file function prototypes, and link with your application. These libraries are distributed in both debug and non-debug versions, and have the following names on the following supported platforms:

• MacOS	`PGPsdkLib` `PGPsdkKeyServerLib`
• Win-32	`PGPsdkLib.dll` `PGPsdKS.dll`
• Unix	`libPGPsdk.a` `libPGPsdkKeyServer.a`

Note that the key server library is required only for those applications that implement direct communication with a key server (see Chapter 8).

Header File Interface

The PGPsdk header file interface consists of one major header file for each functional group. Generally, PGPsdk developers will need to `#include` only that header file to use the associated area of the PGPsdk. The major header files in the initial release of the PGPsdk include

- `pgpCBC.h`
- `pgpCFB.h`
- `pgpEncode.h`
- `pgpFeatures.h`
- `pgpHash.h`
- `pgpKeys.h`
- `pgpKeyServer.h`

- `pgpRandomPool.h`
- `pgpSDKPrefs.h`
- `pgpSymmetricCipher.h`
- `pgpUserInterface.h`
- `pgpUtilities.h`

These major header files may additionally `#include` the following header files that detail common data types, error codes, and platform specific data types, limits, macros, and constants.

- `pgpBase.h`
- `pgpConfig.h`
- `pgpErrors.h`
- `pgpKeyServerTypes.h`
- `pgpPubTypes.h`

Data Type, Constant, Macro, and Function Name Conventions

PGPsdk data types, macros, and functions have names beginning with `PGP`; PGPsdk constants have names beginning with `kPGP` (see Table 1-2).

Most PGPsdk data types are opaque, that is, they are references to the actual data. These data types have names of the form:

`PGP`***name***`Ref`
`PGPConst`***name***`Ref`

where ***name*** describes the data type. Because these data types are opaque, a reference to one is not necessarily a pointer in the *C* language sense, and so they should never be dereferenced.

Most of the PGPsdk opaque data types have special values to indicate that they are not referencing a valid instance. These values are useful for establishing initial or default values, and have names of the form:

`kInvalidPGP`***name***`Ref`

The PGPsdk supports byte array data through use of the *C* language types `char[]` and `void[]`, as well as their associated pointer types `char*` and `void*`. While these basic types may or may not have implementational differences, they do have important PGPsdk-specific semantic differences:

- `char[]` and `char*` always denote `NUL` terminated byte arrays, that is, *C* language strings
- `void[]` and `void*` always denote arbitrary byte arrays that may coincidentally be `NUL` termninated.

PGPsdk constants have names of the form:

`kPGP`***CategoryDescription***

for example, `kPGPKeyPropCanSign`. `kPGP` is the constant data type prefix, `KeyProp` indicates that the constant belongs to the category that refers to key properties, and `CanSign` implies a boolean indicating whether or not the associated key is allowed to sign other keys.

PGPsdk macros and functions have names of the form:

`PGP`***name***

which is a very general format. However, there are several categories of functions that have noteworthy naming conventions and implied semantics:

Data Reference Macros

Macros having names of the form:

`PGP`*`name`*`RefIsValid`

facilitate validation of opaque data types, and return a boolean value. Use of these macros is strongly encouraged, as they provide the PGPsdk developer with a guaranteed method for determining the validity of a data reference, while also maintaining its opacity.

PGPNew*Datatype* and PGPFree*Datatype*

`PGPNew`*`Datatype`* functions allocate a new, persistent instance of a PGPsdk opaque data types. The PGPsdk developer must eventually deallocate the instance with the corresponding "free" function. For example, `PGPNewContext` allocates a new `PGPContextRef`, and `PGPFreeContext` deallocates a `PGPContextRef`. Note that closely related PGPsdk opaque data types may share the same "free" function, for example, `PGPNewContextCustom` also uses `PGPFreeContext`.

PGPO*option*

`PGPO`*`option`* functions allocate `PGPOptionListRef` instances that are automatically deallocated once they are used in an option list management function (for example, `PGPBuildOptionList`), or as a sub-option (for example, `PGPOSignWithKey( …, PGPOPassphrase ( … ), … )`).

Other PGPsdk data types that have noteworthy implied semantics include:

PGPSize

`PGPSize` implies a length quantity, and further implies an in-memory context (similar to the *C* language type `size_t`). Values associated with `PGPSize` items are in terms of the platform's commonly used length quantity, which currently is always in terms of 8-bit bytes.

PGPFlags

`PGPFlags` items differs for other PGPsdk data types that assume enumerated values in that the associated values may be combined with boolean expressions, for example:

```
if  ( ( myFlags & ( kPGPKeyRingOpenFlags_Mutable | kPGPKeyRingOpenFlags_Create ) ) )
{
      /* features-are-available code */
}
```

PGPContext

The PGPsdk incorporates a global context /configuration mechanism for all PGPsdk functionality. The `PGPContext` data type replaces the many global variables used in previous PGP libraries, and thus provides a more robust and manageable application environment. Typically, an application will create a `PGPContext` at startup, use the context throughout its run, and finally free the context on exit.
The resultant `PGPContextRef` value is passed directly to most of the PGPsdk functions. However, some PGPsdk data types incorporate the `PGPContextRef` used to create them, and so the functions that accept these data types as arguments generally do not also require a `PGPContextRef` argument.

A `PGPContext` must *not* be freed until and unless all data items allocated using that context have already been freed. Failure to follow this protocol will not only result in memory leaks, but also precipitate application failures due to the associated context being invalid or incorrect.

A `PGPContext` is created and destroyed as follows:

```
PGPError              err;
PGPUInt32             clientAPIVersion = kPGPsdkVersion;
PGPContextRef         newContext;
PGPContextRef          newCustomContext;
PGPNewContextStruct    newCustomContextStruct;
struct pvtUserData
{
    PGPUInt32   pvtNum;
    char        pvtStr[ 256 ];
}   MyUserValue;

newCustomContextStruct.sizeOfStruct = sizeof( PGPNewContextStruct );
newCustomContextStruct.allocProc = MyAllocProc;
newCustomContextStruct.reallocProc  = MyReAllocProc;
newCustomContextStruct.deallocProc  = MyDeAllocProc;
newCustomContextStruct.allocUserValue = ( PGPUserValue )&MyUserValue;

err = PGPNewContext( clientAPIVersion,                    /* Create Default */
                     &newContext );
err = PGPNewContextCustom( clientAPIVersion,              /* Create Custom */
                           &newCustomContextStruct,
                           &newCustomContext );
err = PGPFreeContext( newContext );                       /* Free Either */
err = PGPFreeContext( newCustomContext );
```

Reference Counts

Most PGP opaque data types have an associated reference count of type …`RefCount`, which provides for simplified garbage collection. Upon creation of such a data type, its reference count is initialized to one. From that point, the PGPsdk automatically tracks the number of references to a particular resource, for example, a given key set may be referenced by any number of key lists and/or iterators. This not only results in a level of context independence, but also ensures that a resource's memory is released only when its last reference is deleted. The PGPsdk also provides functions to support manual adjustment of reference counts.

However, the automatic nature of the reference count management applies only to implied references. This means that the reference count of an underlying key set is automatically incremented whenever a key list is created from it, and is automatically decrememnted when that key list is freed. The PGPsdk developer is expected to adhere to the following basic rule:

> All PGP opaque data types explicitly created (`PGPNew`… functions), copied (`PGPCopy`… functions), or have had their reference count manually incremented must be be freed using the appropriate `PGPFree`… function.

Memory Management

Memory management within the PGPsdk is normally handled transparently by default functions analogous to the Standard *C* Library functions `malloc`, `dealloc`, and `realloc`. However, developers can override this behavior by specifying their own equivalent allocate, deallocate, and reallocate functions (see the `PGPNewContextStruct` data type that is used by the `PGPNewContextCustom` function).

Generally speaking, any PGPsdk function having a name of the form

`PGPNew`…*`datatype`*…

accepts a `PGPContext` reference, and allocates memory which the caller must explicitly deallocate with the corresponding PGPsdk function having a name of the form

PGPFree…*datatype*…

Error Codes

With several exceptions, PGPsdk functions return an error code (kPGPError_…) or void, and place any result values into output arguments. This convention allows for simple and consistent error checking. The PGPsdk provides the macros IsPGPError and IsntPGPError to test a function's return code, as shown in the following example:

```
if  ( IsPGPError( err = pgpOrderKeySet( src, kPGPUserIDOrdering, &keyList ) ) )
{
    /* error handling code */
}
else
{
 /* keyList holds the sorted key set */
}
```

Essentially all PGPsdk functions that return an error code can return one or more of the following:

- kPGPError_NoErr
- kPGPError_BadParams
- kPGPError_OutOfMemory

and these codes are rarely mentioned in the following function reference chapters. Of course, a function that has no parameters cannot return kPGPError_BadParams, nor can a function that does not allocate a new data item return kPGPError_OutOfMemory.

PGPsdk API Details and Data Structures - Key Management

Understanding how the PGPsdk key management functions perform their tasks requires understanding of several PGPsdk Version 1.1-specific concepts and data types. The following sections introduce the PGP key database, collections of keys from a key database, ordered lists of keys from a key database, iterating methods on a list of keys, and the construction of filters that in turn create collections and ordered lists of keys from a key database.

Key Database

The PGP key database represents one or more key files, and can be thought of as a backing store for a key ring. It can be composed of any number of files on disk, or it can be entirely memory based. While the PGPKeyDB is a very important data type to understand, it is currently never exported, nor is there currently a user-visible reference type.

Every key in the system belongs to exactly one key database. Whenever a key is modified, its corresponding key database is also modified. While equivalent keys may exist simultaneously in several key databases, each instance is a distinct key from the point of view of the PGPsdk key management functions - each instance has a unique pointer, and so modifications to one will not affect any of the others.

Collections of Keys in a Key Database

The PGPKeySet data type represents a subset (referred to as a *key set*) of exactly one key database, and may be thought of as a view onto that key database. The function PGPOpenDefaultKeyrings opens the caller's default key rings, which is conceptually a key database consisting of two files - the caller's **public key** and **private key** key ring files. The function then creates and returns a key set containing the full set of keys in that key database.

Any number of key sets may exist for a given key database (see the discussion on key filters in this chapter). For instance, one could create a key set that includes all keys, as well as a key set that includes only those keys signed by "Philip Zimmermann".

A key set is generally an "active" or a "live" view on a key database. To demonstrate what an active view is, consider a key set that is composed of all the keys that contain the name "Mark." Creating this key set with an active key filter and then adding a key containing name "Mark" to the associated key database results in that key being automatically and instantaneously added to the created key set, and vice versa.

Lists of Keys in a Key Database

Key sets have no ordering – they are merely collections of keys. The `PGPKeyList` data type facilitates operations on key sets by imposing an ordering that may be based on any sortable data item or sub-structure within a key, for example, name or key ID. The function `PGPOrderKeySet` accepts a key set and a sort order specification, and yields a key list. Like key sets, key lists are also "active", and are automatically updated whenever their associated key set is changed.

The `PGPKeyIter` data type implements iterating over a key list. Initially, it references the pseudo-element just before the first element in the key list, and then increments itself successively through each element of the key list. Most changes to a key list that occur while iterating are handled automatically. For example, inserting a new key causes the iteration to automatically "follow" the key it was working on. The `PGPKeyIter` data type also supports iteration over the sub-structures within the key, for example, iterating over the user ID structures of the key.

Key Filters

The PGPsdk allows the developer to construct very complicated key **filters** for operating on elements of the key database. These filters are built from **primitive key filters**, which in turn are created by the various `PGPNew…Filter` functions. These primitive key filters are generally of the form:

```
select all X that contain Y
```

A set of related functions allows negation, union, and intersection of primitive key filters, and so allows creation of key filters that implement arbitrary expressions such as

```
select all keys NOT containing "Phil" AND
    having keylengths longer than 1024 bits
```

Once the key filter is complete, the `PGPFilterKeySet` function applies the resultant key filter to a key set, yielding a new key set whose members satisfy the key filter criteria. Note that this resultant new key set may be empty.

Summary of the Opaque PGPsdk Data Types

Many of the data types described in this section are actually opaque to the PGPsdk user, and can only be passed as function arguments. Except for `PGPContextRef`, these data types exist as pairs, with one having the `const` qualifier. The following tables summarize these data types:

Table 1-2: Common Opaque Data Types

Data Type
`PGPContextRef`

PGPFileSpecRef
PGPOptionListRef
PGPConstFileSpecRef
PGPConstOptionListRef

Table 1-3: Key-related Opaque Data Types

Data Type
PGPFilterRef
PGPKeyRef
PGPKeydbRef
PGPKeyIterRef
PGPKeyListRef
PGPKeySetRef
PGPSigRef
PGPSubKeyRef
PGPUserIDRef
PGPConstFilterRef
PGPConstKeyRef
PGPConstKeydbRef
PGPConstKeyIterRef
PGPConstKeyListRef
PGPConstKeySetRef
PGPConstSigRef
PGPSubKeyRef
PGPConstUserIDRef

Table 1-4: Low-level Cipher-related Opaque Data Types

Data Type
PGPCBCContextRef
PGPCFBContextRef
PGPHashContextRef
PGPPrivateKeyContextRef
PGPPublicKeyContextRef
PGPSymmetricCipherContextRef
PGPConstCBCContextRef
PGPConstCFBContextRef
PGPConstHashContextRef
PGPConstPrivateKeyContextRef
PGPConstPublicKeyContextRef
PGPConstSymmetricCipherContextRef

PGPsdk API Details and Data Structures - Ciphering

Using the PGPsdk Ciphering API

The PGPsdk Ciphering API has two high-level entry points: `PGPEncode` and `PGPDecode`. `PGPEncode` provides for all encrypting and signing, while `PGPDecode` provides for all decrypting and signature verification. Each function accepts a `PGPContextRef`, and a variable number of options that control the behavior of the function. The similarity of their prototypes is illustrated by the following examples:

```
PGPError        PGPEncode( PGPContextRef pgpContext,
                           PGPOptionListRef firstOption,
                           ...,
                           PGPOLastOption( void ) );

PGPError          PGPDecode(  PGPContextRef pgpContext,
                           PGPOptionListRef firstOption,
                           ...,
                           PGPOLastOption( void ) );
```

A large number of options is available for both `PGPEncode` and `PGPDecode`, and each is defined as a function returning a `PGPOptionListRef`. Some options are suitable only for encoding operations, some options are suitable only for decoding operations, and some options are suitable for both operations. These options are described in Chapter 3, "Function Reference - Ciphering and Authentication Functions." A special argument provided by the `PGPOLastOption` function must appear as the last argument to indicate the end of the list.

Events and Callbacks

The `PGPOEventHandler` option allows the calling application to request callbacks when various events occur, and to define a function (**event handler**) that is the target of the callback. While an event handler is usually not needed for encryption operations, it is often needed for decryption operations.

An event handler serves two purposes – it provides notification to the calling application that an event has occurred, and provides a mechanism for the calling application to affect processing (in a limited, pre-defined manner). Notification includes a `PGPEvent` reference which, depending on the type of event, provides detailed information about the cause of the event. The calling application can then respond appropriately, which may or may not affect the course of further processing. For certain events, the calling application can modify the processing context by invoking `PGPAddJobOptions`.

PGPsdk API Details and Data Structures - Authentication

The PGPsdk performs authentication (signing and verification of messages) by using the supplied `PGPEncode` and `PGPDecode` functions. In the case of signing or verifying a message, the application invokes the appropriate `PGPO…` function(s) (for example, `PGPOSignWithKey` and `PGPODetachedSig`) to perform the needed authentication function. In the case of authentication, the message is first passed through a hash function before being signed by the sender's private key.

Hash Functions

The PGPsdk provides a number of **hash functions** (more commonly referred to as **hash algorithms**). Selection of a specific hash algorithm is sometimes implicit to the processing context; for example, **DSS** keys unequivocally use the **SHA-1** hash algorithm. For other processing contexts, the `PGPOHashAlgorithm` function can be used to "manually" configure the context; for example, the function can force the use of the SHA-1 hash function in an **RSA** signature.

PGPsdk Code Example

The sample code and other usage examples that appear in the Function Reference chapters of this book are for demonstration purposes only. The final *PGP for Business Security, Version 5.5 Source Code* books (when available) will serve as the essential and definitive reference for developers using the PGPsdk to develop their own secure applications.

Chapter 2: Function Reference - Key Management Functions

Introduction

The PGPsdk key management functions allow applications to create, sign, add, remove, search for, and check the validity of keys on disk-based or in-memory key rings. This Chapter 2lso documents functions that check and set property values for keys, as well as functions that import and export keys to files and buffers.

A PGP key is always a signing key, and for certain algorithims is also an encryption key. If a sub-key is present, then it is always considered to be an encryption key. Some algorithms, for example Diffie-Hellman, require sub-keys since the base key is always considered to be sign-only. Other algorithms, for example RSA, do not support sub-keys, and for these the base key is used for both signing and encrypting.

Diffie–Hellman keys may have associated **additional recipient request keys** (sometimes referred to by its marketing term, *additional decryption key*). When present, all messages encrypted to the base key are also automatically encrypted to each of the additional recipient request keys.

A key may have any number of associated sub-keys, additional recipient request (ARR) keys, and user IDs. A user ID in turn may have any number of associated signatures.

Figure 2-1. Diffie-Hellman Key Structure.

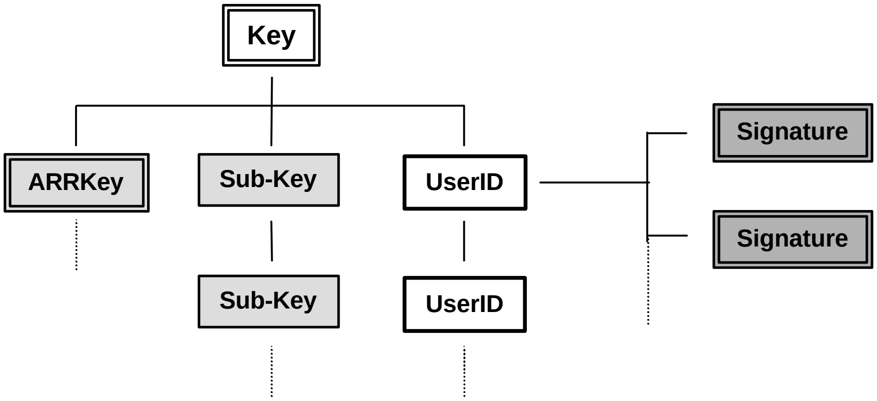
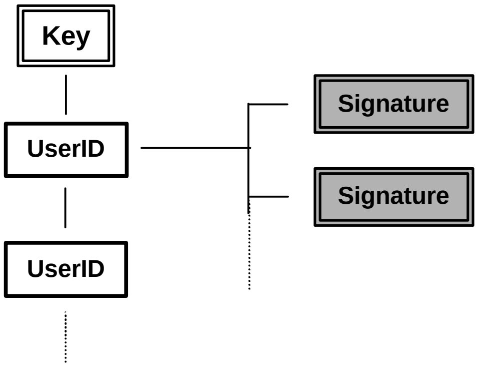


Figure 2-2. RSA Key Structure.



Header Files

`pgpKeys.h`

Constants and Data Structures

Table 2-1: Key Ring OPEN Flag Values.

Key Ring Open Flag Constants
`kPGPKeyRingOpenFlags_Create`
`kPGPKeyRingOpenFlags_Mutable`
`kPGPKeyRingOpenFlags_Private***`
`kPGPKeyRingOpenFlags_Reserved`
`kPGPKeyRingOpenFlags_Trusted***`

*** Applies to `PGPOpenKeyRing` only.

Table 2-2: Key Ordering Specification Values.

Key Ordering Constants
`kPGPAnyOrdering`
`kPGPCreationOrdering`
`kPGPEncryptKeySizeOrdering`
`kPGPExpirationOrdering`
`kPGPKeyIDOrdering`
`kPGPSigKeySizeOrdering`
`kPGPTrustOrdering`
`kPGPUserIDOrdering`
`kPGPValidityOrdering`
`kPGPReverseCreationOrdering`
`kPGPReverseEncryptKeySizeOrdering`
`kPGPReverseExpirationOrdering`
`kPGPReverseKeyIDOrdering`
`kPGPReverseSigKeySizeOrdering`
`kPGPReverseTrustOrdering`
`kPGPReverseUserIDOrdering`
`kPGPReverseValidityOrdering`

Table 2-3: Comparison Option Specification Values.***

Match Criterion Constants	
`kPGPMatchDefault`	same as `kPGPMatchEqual`
`kPGPMatchEqual`	searched value == supplied value
`kPGPMatchGreaterOrEqual`	searched value >= supplied value
`kPGPMatchLessOrEqual`	searched value <= supplied value
`kPGPMatchSubString`	searched value is contained in supplied value

*** Certain functions that accept a match criteria value support `kPGPMatchEqual` and/or `kPGPMatchSubString` only. This primarily affects functions which deal with text and/or arbitrary byte data.

Table 2-4: Key- and Sub-Key Related Property Specification Values.

Key and Sub-Key Property Constants
String Properties
kPGPKeyPropFingerprint
kPGPKeyPropPreferredAlgorithms
Numerical Properties
kPGPKeyPropAlgID
kPGPKeyPropBits
kPGPKeyPropTrust
kPGPKeyPropValidity
Time Properties
kPGPKeyPropCreation
kPGPKeyPropExpiration
Boolean Properties
kPGPKeyPropCanEncrypt
kPGPKeyPropCanSign
kPGPKeyPropHasUnverifiedRevocation
kPGPKeyPropIsAxiomatic
kPGPKeyPropIsDisabled
kPGPKeyPropIsExpired
kPGPKeyPropIsNotCorrupt
kPGPKeyPropIsRevoked
kPGPKeyPropIsSecret
kPGPKeyPropNeedsPassphrase

Table 2-5: User ID-Related Property Specification Values.

User ID Property Constants
String properties
kPGPUserIDPropName
Numeric properties
kPGPUserIDPropConfidence
kPGPUserIDPropValidity
Time properties
Boolean properties

Table 2-6: Signature-Related Property Specification Values.

Signature Property Constants
String properties
Numeric properties
kPGPSigPropAlgID
kPGPSigPropKeyID
kPGPSigPropTrustValue
Time properties
kPGPSigPropCreation
kPGPSigPropExpiration
Boolean properties
kPGPSigPropHasUnverifiedRevocation
kPGPSigPropIsExportable
kPGPSigPropIsMySig***
kPGPSigPropIsNotCorrupt

kPGPSigPropIsRevoked
kPGPSigPropIsTried
kPGPSigPropIsVerified

*** kPGPPropSigIsMySig is a convenience property for determining whether the certification was made by one of the caller's own private keys. This will yield TRUE only if the signing key is in the same base key set as the certification. If the signing key is suspected to be in a different base key set, then use the following code:

```
PGPGetSigCertifierKey( certset, signerset, &key );
PGPGetKeyBoolean( key, kPGPKeyPropIsSecret, &secret );
if  ( secret )
{
  /* signing key is one of the caller’s private keys */
}
else
{
  /* signing key is not one of the caller’s private keys */
}
```

Table 2-7: Key ID String Type Specification Values.

Key ID String Type Constants
kPGPKeyIDString_Abbreviated
kPGPKeyIDString_Full

Table 2-8a: Valid PGPOptionListRef Options for Key Generation Functions.

Option Function	PGPGetKeyEntropyNeeded	PGPGenerateKey	PGPGenerateSubKey
PGPOAdditionalRecipientRequestKeySet		●	
PGPOAllocatedOutputBuffer			
PGPOAppendOutput			
PGPOArmorOutput			
PGPOAskUserForEntropy			
PGPOCipherAlgorithm			
PGPOClearSign			
PGPOCommentString			
PGPOCompression			
PGPOConventionalEncrypt			
PGPODataIsASCII			
PGPODetachedSig			
PGPODiscardOutput			
PGPOEncryptToKey			
PGPOEncryptToKeySet			
PGPOEncryptToUserID			
PGPOEventHandler		●	●
PGPOExpiration		●	●
PGPOExportable			
PGPOExportPrivateKeys			
PGPOFailBelowValidity			
PGPOForYourEyesOnly			
PGPOHashAlgorithm			
PGPOImportKeysTo			
PGPOInputBuffer			
PGPOInputFile			
PGPOInputFileFSSpec			
PGPOKeyGenFast	●	●	●
PGPOKeyGenMasterKey			●
PGPOKeyGenName		●	
PGPOKeyGenParams	●	●	●
PGPOKeySetRef		●	
PGPOLastOption	●	●	●
PGPOLocalEncoding			
PGPONullOption	●	●	●

PGPOOmitMIMEVersion			
PGPOOutputBuffer			
PGPOOutputFile			
PGPOOutputFileFSSpec			
PGPOOutputLineEndType			
PGPOPassphrase		●	●
PGPOPassphraseBuffer		●	●
PGPOPassThroughIfUnrecognized			
PGPOPGPMIMEEncoding			
PGPOPreferredAlgorithms		●	
PGPORawPGPInput			
PGPOSendEventIfKeyFound			
PGPOSendNullEvents			
PGPOSignWithKey			
PGPOSigRegularExpression			
PGPOSigTrust			
PGPOVersionString			
PGPOWarnBelowValidity			

Table 2-8b: Valid `PGPOptionListRef` Options for Key Generation (User ID) Functions.

Option Function	PGPAddUserID	PGPSignUserID
PGPOAdditionalRecipientRequestKeySet		
PGPOAllocatedOutputBuffer		
PGPOAppendOutput		
PGPOArmorOutput		
PGPOAskUserForEntropy		
PGPOCipherAlgorithm		
PGPOClearSign		
PGPOCommentString		
PGPOCompression		
PGPOConventionalEncrypt		
PGPODataIsASCII		
PGPODetachedSig		
PGPODiscardOutput		
PGPOEncryptToKey		
PGPOEncryptToKeySet		
PGPOEncryptToUserID		
PGPOEventHandler		
PGPOExpiration		●
PGPOExportable		●
PGPOExportPrivateKeys		
PGPOFailBelowValidity		
PGPOForYourEyesOnly		
PGPOHashAlgorithm		
PGPOImportKeysTo		
PGPOInputBuffer		
PGPOInputFile		
PGPOInputFileFSSpec		
PGPOKeyGenFast		
PGPOKeyGenMasterKey		
PGPOKeyGenName		
PGPOKeyGenParams		
PGPOKeySetRef		
PGPOLastOption	●	●
PGPOLocalEncoding		
PGPONullOption	●	●
PGPOOmitMIMEVersion		
PGPOOutputBuffer		
PGPOOutputFile		
PGPOOutputFileFSSpec		
PGPOOutputLineEndType		
PGPOPassphrase	●	●
PGPOPassphraseBuffer	●	●
PGPOPassThroughIfUnrecognized		
PGPOPGPMIMEEncoding		

PGPOPreferredAlgorithms		
PGPORawPGPInput		
PGPOSendEventIfKeyFound		
PGPOSendNullEvents		
PGPOSignWithKey		
PGPOSigRegularExpression		●
PGPOSigTrust		●
PGPOVersionString		
PGPOWarnBelowValidity		

Table 2-8c: Valid `PGPOptionListRef` Options for Key Revocation Functions.

Option Function	PGPRevokeKey	PGPRevokeSubKey	PGPRevokeSig
PGPOAdditionalRecipientRequestKeySet			
PGPOAllocatedOutputBuffer			
PGPOAppendOutput			
PGPOArmorOutput			
PGPOAskUserForEntropy			
PGPOCipherAlgorithm			
PGPOClearSign			
PGPOCommentString			
PGPOCompression			
PGPOConventionalEncrypt			
PGPODataIsASCII			
PGPODetachedSig			
PGPODiscardOutput			
PGPOEncryptToKey			
PGPOEncryptToKeySet			
PGPOEncryptToUserID			
PGPOEventHandler			
PGPOExpiration			
PGPOExportable			
PGPOExportPrivateKeys			
PGPOFailBelowValidity			
PGPOForYourEyesOnly			
PGPOHashAlgorithm			
PGPOImportKeysTo			
PGPOInputBuffer			
PGPOInputFile			
PGPOInputFileFSSpec			
PGPOKeyGenFast			
PGPOKeyGenMasterKey			
PGPOKeyGenName			
PGPOKeyGenParams			
PGPOKeySetRef			
PGPOLastOption	●	●	●
PGPOLocalEncoding			
PGPONullOption	●	●	●
PGPOOmitMIMEVersion			
PGPOOutputBuffer			
PGPOOutputFile			
PGPOOutputFileFSSpec			
PGPOOutputLineEndType			
PGPOPassphrase	●	●	●
PGPOPassphraseBuffer	●	●	●
PGPOPassThroughIfUnrecognized			
PGPOPGPMIMEEncoding			
PGPOPreferredAlgorithms			
PGPORawPGPInput			
PGPOSendEventIfKeyFound			
PGPOSendNullEvents			
PGPOSignWithKey			
PGPOSigRegularExpression			
PGPOSigTrust			
PGPOVersionString			
PGPOWarnBelowValidity			

Events and Callbacks

A number of the key management functions allow the calling application to request callbacks to track the progress of the operation. Those functions that permit inclusion of a `PGPOEventHandler` option generally execute so quickly that an event handler is of limited benefit unless the key set involved is very large. Those functions that include an explicit event handler argument generally require a perceptible amount of execution time, regardless of the size of the key set.

An event handler serves two purposes – it provides notification to the calling application that an event has occurred, and provides a mechanism for the calling application to affect processing (in a pre-defined manner). Notification includes a pointer to a `PGPEvent` data type that, depending on the type of event, provides detailed information about the cause of the event. The calling application can then respond appropriately, which may or may not intervene and affect the course of further processing. If the calling application wishes to intervene, then it can abort the job by returning an error code (a value other than `kPGPError_NoErr`). Additionally, depending on the type of event, it can modify the processing context by invoking `PGPAddJobOptions`.

All event handlers are declared as

```
PGPError myEvents( PGPContextRef pgpContext,
                   PGPEvent *event,
                   PGPUserValue userValue );
```

The `pgpContext` argument is the reference to the context of the function posting the event. The `event` argument references a `PGPEvent` data type as follows:

```
struct PGPEvent_
{
    PGPVersion             version;
    struct PGPEvent_       *nextEvent;
    PGPJobRef              job;
    PGPEventType           type;
    PGPEventData           data;
};
typedef struct PGPEvent_ PGPEvent;
```

The `version` and `nextEvent` members are currently reserved for internal use. The `job` member is not applicable to key management functions. The `type` member identifies the event being posted, and recognizes `kPGPEvent_`… values (see Table 3-1). The `data` member is a `union` of the event-specific data structures, which are described with their corresponding event.

None of the key management functions currently support modification of the processing context by invoking `PGPAddJobOptions`.

Figure 2-3: (Sub-)Key Generation Event Sequence.

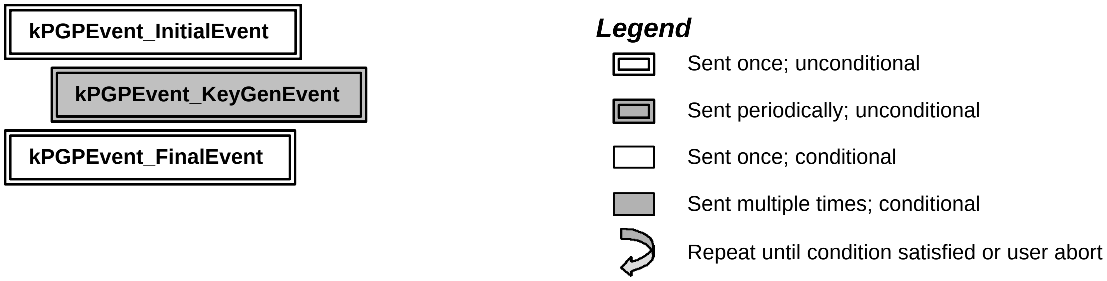
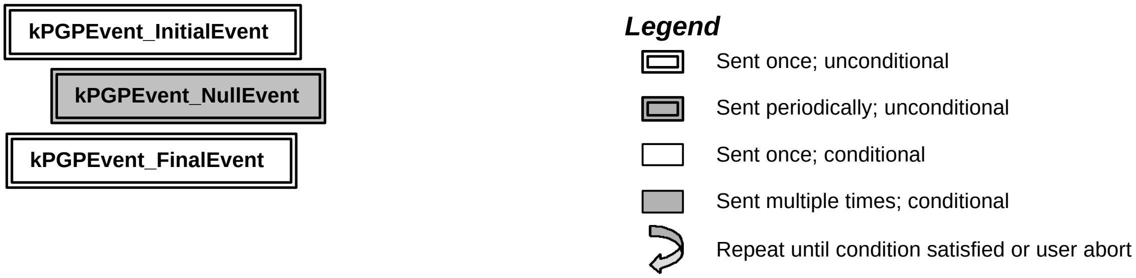


Figure 2-4: Key Set Operation Event Sequence.



Key Management Events

kPGPEvent_InitialEvent

Sent before all other events. Implies entry to the function.

Data

None

Options

None

kPGPEvent_NullEvent

Sent during the course of key set import/export processing if explicitly requested with `PGPOSendNullEvents` (see `PGPExportKeySet` and `PGPImportKeySet`). Automatically sent during signature checking (see `PGPCheckKeyRingSigs`).

The event data allows the PGPsdk developer to determine the sending function's progress by way of its completion percentage. The event data members should be treated as relative, unscaled quantities – they are not necessarily byte quantities or number-of-keys values. In all cases, the completion percentage is calculated as follows:

```
double      completionPercent;

if  ( event->type = kPGPEvent_NullEvent )
{
     if  ( event->nullData.bytesTotal != 0 )
     {
       completionPercent = ( 100 * event->nullData.bytesWritten ) /
                             event->nullData.bytesTotal;
     }
     else
     {
       completionPercent = 100;
     }
}
```

Data

```
typedef struct PGPEventNullData_
```

```
{
    PGPFileOffset     bytesWritten;
    PGPFileOffset     bytesTotal;
} PGPEventNullData;
```

Options

None

kPGPEvent_KeyGenEvent

Automatically sent during the course of key and sub-key generation (see `PGPGenerateKey` and `PGPGenerateSubKey`).

The event data allows the PGPsdk developer to determine the progress of the key generation process. If the event handler returns an error, then the key generation process aborts.

The `state` value indicates the *approximate* state of the key generation process, and assumes the *character* values that were used by previous text-versions of PGP:

- **.** selected value failed pseudo-primality test
- **/** all selected values failed pseudo-primality test; re-initializing the prime number generation environment
- **-** selected value passed pseudo-primality test; further processing required
- **+** selected value passed pseudo-primality test; further processing required
- ***** selected value passed pseudoprimality test; processing for this phase is near completion.
- ***space*** completion of this phase of (sub-)key generation. The actual number of phases varies from key to key, and has no fixed value or range

Data

```
typedef struct PGPEventKeyGenData_
{
    PGPUInt32         state;
} PGPEventKeyGenData;
```

Options

None

kPGPEvent_FinalEvent

Sent after all other events. Implies return from the function.

Data

None

Options

None

Key Set Functions

PGPOpenDefaultKeyRings

```
PGPError         PGPOpenDefaultKeyRings(
                      PGPContextRef       pgpContext,
                      PGPKeyRingOpenFlags
                                          openFlags,
                      PGPKeySetRef        *keySet );
```

Arguments

`pgpContext`	the target context
`openFlags`	the open options, which recognizes `kPGPKeyRingOpenFlags_`... values (see Table 2-1)
`keySet`	the receiving field for the new key set

Description

Creates a key set that contains all of keys in the default public key and private key key rings. Any trust information associated with the public key ring is included.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant key set with `PGPFreeKeySet`.

This is a convenience function that is the equivalent of

```
if  (IsPGPError( err = PGPsdkPrefGetFileSpec ( pgpContext,
                              kPGPsdkPref_PublicKeyring,
                              &pubKeyFileSpec ) ) )
{
      return( err );
}
if  (IsPGPError( PGPsdkPrefGetFileSpec ( pgpContext,
                              kPGPsdkPref_PrivateKeyring,
                              &privKeyFileSpec ) ) )
{
      return( err );
}

err = PGPOpenKeyRingPair( pgpContext,
                           openFlags,
                           pubFileSpec,
                           secFileSpec,
                           &keySet );
return( err );
```

Any `kPGPKeyRingOpenFlags_Private` or `kPGPKeyRingOpenFlags_Trusted` open options specification are ignored (debug versions, however, will assert; see `PGPGetFeatureFlags and kPGPFeatureMask_IsDebugBuild`).

PGPOpenKeyRingPair

```
PGPError        PGPOpenKeyRingPair(
                     PGPContextRef      pgpContext,
                     PGPKeyRingOpenFlags
                                        openFlags,
                     PGPFileSpec        pubFileSpec,
                     PGPFileSpec        secFileSpec,
                     PGPKeySetRef       *keySet );
```

Arguments

`pgpContext`	the target context
`openFlags`	the open option flags value
`pubFileSpec`	the target public key ring file
`secFileSpec`	the target private key ring file
`keySet`	the receiving field for the new key set

Description

Creates a key set that contains all of the keys in the specified public and private key ring files. Any trust information associated with the public key ring are included.

Notes, Warnings, and Tips

For most applications, `PGPOpenDefaultKeyRings` provides all the functionality required.

The caller is responsible for deallocating the resultant key set with `PGPFreeKeySet`.

Any `kPGPKeyRingOpenFlags_Private` or `kPGPKeyRingOpenFlags_Trusted` open options specification are ignored (debug versions, however, will assert; see `PGPGetFeatureFlags and kPGPFeatureMask_IsDebugBuild`).

PGPOpenKeyRing

```
PGPError        PGPOpenKeyRing(
                     PGPContextRef      pgpContext,
                     PGPKeyRingOpenFlags
                                        openFlags,
                     PGPFileSpec        fileSpec,
                     PGPKeySetRef       *keySet );
```

Arguments

`pgpContext`	the target context
`openFlags`	the open option flags value
`fileSpec`	the target key ring file
`keySet`	the receiving field for the new key set

Description

Creates a key set that contains all of the keys in the specified key ring file. The open options are interpreted as follows:

- kPGPKeyRingOpenFlags_Create – TRUE if the specified key ring file should be created if it doesn't already exist.
- kPGPKeyRingOpenFlags_Mutable – TRUE if the resultant key set should be made modifiable; FALSE if the resultant key set should be made read-only.
- kPGPKeyRingOpenFlags_Trusted – TRUE if any associated trust information should be included.
- kPGPKeyRingOpenFlags_Private – TRUE if the specified key ring file should be considered private; FALSE if the specified key ring file should be considered public.

Errors

kPGPError_FileNotFound
kPGPError_FileLocked

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant key set with PGPFreeKeySet.

kPGPError_FileNotFound will never be returned if the open options include kPGPKeyRingOpenFlags_Create.

PGPNewKeySet

```
PGPError          PGPNewKeySet(
                       PGPContextRef       pgpContext,
                       PGPKeySetRef        *keySet );
```

Arguments

pgpContext	the target context
keySet	the receiving field for the new key set

Description

Creates a new memory-based *key database*, as well as an empty key set on that key database.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant key set with PGPFreeKeySet.

The current implementation treats the resultant key set as an indirect parameter that references a key database, rather than as an explicit destination.

The indirect nature of this interface is likely to change in a future version, and will almost certainly involve changes to this function's parameterization.

PGPNewEmptyKeySet

```
PGPError          PGPNewEmptyKeySet(
                       PGPKeySetRef        baseKeySet,
                       PGPKeySetRef        *newKeySet );
```

Arguments

`baseKeySet`	the source key set
`newKeySet`	the receiving field for the new key set

Description

Creates a new, empty key set on the *key database associated with* the specified source key set.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant key set with `PGPFreeKeySet`.

The current implementation treats the supplied key set as an indirect parameter that references a key database, rather than as an explicit source.

The indirect nature of this interface is likely to change in a future version, and will almost certainly involve changes to this function's parameterization.

PGPNewSingletonKeySet

```
PGPError          PGPNewSingletonKeySet(
                       PGPKeyRef           key,
                       PGPKeySetRef        *keySet );
```

Arguments

`key`	the seed key
`keySet`	the receiving field for the new key set

Description

Creates a key set that is *not associated with any key database*, and that contains only the specified seed key.

Notes, Warnings, and Tips

This function does *not* create a new key database; the resultant key set contains only the one key.

The caller is responsible for deallocating the resultant key set with `PGPFreeKeySet`.

PGPUnionKeySets

```
PGPError          PGPUnionKeySets(
                       PGPKeySetRef        firstKeySet,
                       PGPKeySetRef        secondKeySet,
                       PGPKeySetRef        *resultKeySet );
```

Arguments

`firstKeySet`	the first source key set
`secondKeySet`	the second source key set
`resultKeySet`	the receiving field for the new key set

Description

Creates a new key set that is the union of the two source key sets

Notes, Warnings, and Tips

The two source key sets must be in the same key database.

The caller is responsible for deallocating the resultant key set with `PGPFreeKeySet`.

PGPFreeKeySet

```
PGPError         PGPFreeKeySet(
                      PGPKeySetRef       keySet );
```

Arguments

`keySet`	the target key set

Description

Decrements the reference count for the specified key set, and frees the key set if the reference count reaches zero.

PGPReloadKeyRings

```
PGPError         PGPReloadKeyRings(
                      PGPKeySetRef       keySet );
```

Arguments

`keySet`	the target key set

Description

Forcibly re-establishes the *key database associated with* the specified key set from the key database source files.

Notes, Warnings, and Tips

The current implementation treats the target key set as an indirect parameter that references a key database, rather than as an explicit destination.

The indirect nature of this interface is likely to change in a future version, and will almost certainly involve changes to this function’s parameterization.

PGPImportKeySet

```
PGPError         PGPImportKeySet(
                      PGPContextRef      pgpContext,
                      PGPKeySetRef       keySet,
                      PGPOptionListRef    firstOption,
                      ...,
                      PGPOLastOption( void ) );
```

Arguments

```
pgpContext              the target context
keySet                  the target key set
firstOption             the initial option list instance
...                     subsequent option list instances
PGPOLastOption( void )
                        must always appear as the final argument to terminate the argument list
```

Description

Imports the specified keys from the input source specified in the options list, and merges them with the specified key set. By including an option that specifies sending null events, the PGPsdk developer can provide for tracking the progress of the function (see PGPOSendNullEvents).

Notes, Warnings, and Tips

One of PGPOInputBuffer, PGPOInputFile, and PGPOInputFileFSSpec is required to specify the key source file.

PGPExportKeySet

```
PGPError         PGPExportKeySet(
                        PGPKeySetRef        keySet,
                        PGPOptionListRef    firstOption,
                        ...,
                        PGPOLastOption( void ) );
```

Arguments

```
keySet                  the target key set
firstOption             the initial option list instance
...                     subsequent option list instances
PGPOLastOption( void )
                        must always appear as the final argument to terminate the argument list
```

Description

Exports the specified keys in the specified key set to the output destination specified in the options list. By including an option that specifies sending null events, the PGPsdk developer can provide for tracking the progress of the function (see PGPOSendNullEvents).

Notes, Warnings, and Tips

One of PGPOAllocatedOutputBuffer, PGPOOutputBuffer, PGPOOutputFile, and PGPOOutputFileFSSpec is required to specify an output destination for functions that accept this option.

Exporting a key set and then importing it back does not necessarily result in a key set identical to that which was initially exported. For example, if a key was signed as being non-exportable, then its signature data will be lost (see PGPOExportable).

PGPCountKeys

```
PGPError         PGPCountKeys(
                      PGPKeySetRef       keySet,
                      PGPUInt32          *numKeys );
```

Arguments

`keySet`	the target key set
`numKeys`	the receiving field for the key count

Description

Retrieves the number of keys in the specified key set.

PGPKeySetIsMember

```
PGPBoolean  PGPKeySetIsMember(
                      PGPKeyRef          key,
                      PGPKeySetRef       keySet );
```

Arguments

`key`	the target key
`keySet`	the target key set

Description

Returns `TRUE` if the specified key is in the specified key set.

PGPKeySetIsMutable

```
PGPBoolean       PGPKeySetIsMutable(
                      PGPKeySetRef       keySet );
```

Arguments

`keySet`	the target key set

Description

Returns `TRUE` if the specified key set can be modified, that is if keys and their components (subkeys, signatures, and user IDs) can be added to the key set, deleted from the key set, and have their properties changed in the key set.

PGPAddKeys

```
PGPError         PGPAddKeys(
                      PGPKeySetRef       keysToAdd,
                      PGPKeySetRef       keySet );
```

Arguments

`keysToAdd`	the source key set, which contains the keys to be added
`keySet`	the target (“to be augmented”) key set

Description

Copies all of the keys in the specified source key set to the *key database associated with* the specified destination (“to be augmented”) key set.

Notes, Warnings, and Tips

The current implementation treats the destination key set as an indirect parameter that references a key database, rather than as an explicit destination. Because of key filtering and the “live” nature of its resultant view-style key sets, the keys added by this function may appear in any key set based upon that key database, and further may or may not appear in the specified destination key set, depending upon its key filtering criteria.

The indirect nature of this interface is likely to change in a future version, and will almost certainly involve changes to this function’s parameterization.

PGPRemoveKeys

```
PGPError        PGPRemoveKeys(
                     PGPKeySetRef        keysToRemove,
                     PGPKeySetRef        keySet );
```

Arguments

`keysToremove`	the source key set, which contains the keys to be removed
`keySet`	the target (“to be pruned”) key set

Description

Removes each of the keys in the specified source key set from the *key database associated with* the specified destination (“to be pruned”) key set.

Notes, Warnings, and Tips

The current implementation treats the destination key set as an indirect parameter that references a key database, rather than as an explicit destination. Because of key filtering and the “live” nature of its resultant view-style key sets, the keys removed by this function may disappear from any key set based upon that key database, and further may or may not disappear from the specified destination key set, depending upon its key filtering criteria.

The indirect nature of this interface is likely to change in a future version, and will almost certainly involve changes to this function’s parameterization.

PGPCheckKeyRingSigs

```
PGPError        PGPCheckKeyRingSigs(
                     PGPKeySetRef        keysToCheck,
                     PGPKeySetRef        keysSigning,
                     PGPBoolean          checkAll,
                     PGPEventHandlerProcPtr
                                         eventHandler,
                                PGPUserValue       eventHandlerArg );
```

Arguments

`keysToCheck`	the target key set
`keysSigning`	the look-up key set that contains the signing keys
`checkAll`	`TRUE` to check all signatures; `FALSE` to check only those marked as being unchecked
`eventHandler`	event handler or `(PGPEventHandlerProcPtr)NULL` to ignore any and all events
`eventHandlerArg`	user-defined data, to be passed to the event handler (meaningful only in conjunction with `eventHandler`)

Description

Checks all signatures (or only those marked unchecked) of each key in the *key database associated with* the target key set. Each signature is assumed to exist in the *key database associated with* the look-up key set, which is typically all of the client's default keys.

Events of type `kPGPEvent_NullEvent` are sent during the course of processing, and the PGPsdk developer can choose to handle them with the optional event handler.

Notes, Warnings, and Tips

This is a resource-intensive function, whose execution time can be quite lengthy.

The PGPsdk developer can choose to point the optional event handler to a function that implements a progress bar display, or anything else that the PGPsdk developer desires. `userValue` is passed to the event handler function, and has meaning only in conjunction with the event handler function (see the description for `kPGPEvent_NullEvent`).

Specify `eventHandlerArg` as `( PGPUserData )0` to indicate a dummy argument.

The current implementation treats the target and look-up key sets as indirect parameters that reference key databases, rather than as explicit destinations and sources. Because of key filtering and the "live" nature of its resultant view-style key sets, the keys modified as a result any action by the optional event handler may be reflected in any key set based upon that key database, and further may or may not be reflected in the specified destination key set, depending upon its key filtering criteria.

The indirect nature of this interface is likely to change in a future version, and will almost certainly involve changes to this function's parameterization.

PGPPropagateTrust

```
PGPError         PGPPropagateTrust(
                      PGPKeySetRef       keySet );
```

Arguments

`keySet`	tthe target key set

Description

Propagates the **trust** information across the *key database associated with* the specified key set.

Notes, Warnings, and Tips

The current implementation treats the destination key set as an indirect parameter that references a key database, rather than as an explicit destination. Because of key filtering and the “live” nature of its resultant view-style key sets, the trust values propagated by this function may be reflected in any key set based upon that key database, and further may or may not be reflected in the specified destination key set, depending upon its key filtering criteria.

The indirect nature of this interface is likely to change in a future version, and will almost certainly involve changes to this function’s parameterization.

PGPKeySetNeedsCommit

```
PGPBoolean  PGPKeySetNeedsCommit(
                     PGPKeySetRef       keySet );
```

Arguments

`keySet`	the target key set

Description

Returns `TRUE` if there any changes pending for the *key database associated wit*h the target key set.

Notes, Warnings, and Tips

The current implementation treats the target key set as an indirect parameter that references a key database, rather than as an explicit destination.

The indirect nature of this interface is likely to change in a future version, and will almost certainly involve changes to this function’s parameterization.

PGPCommitKeyRingChanges

```
PGPError         PGPCommitKeyRingChanges(
                     PGPKeySetRef       keySet );
```

Arguments

`keySet`	the target key set

Description

Checks any signatures that are marked as unchecked, and re-propagates their trust model information and other attributes. It then writes any changes pending in the *key database associated with* the target key set to the disk file(s) upon which the key database is based.

Notes, Warnings, and Tips

Changes are only written to disk if and when the PGPsdk client calls this function.

The current implementation treats the target key set as an indirect parameter that references a key database, rather than as an explicit destination. Because of key filtering and the “live” nature of its resultant view-style key sets, any keys modified by this function may be reflected in any key set based upon that key database, and further may or may not be reflected in the specified destination key set, depending upon its key filtering criteria.

The indirect nature of this interface is likely to change in a future version, and will almost certainly involve changes to this function's parameterization.

PGPRevertKeyRingChanges

```
PGPError         PGPRevertKeyRingChanges(
                      PGPKeySetRef        keySet );
```

Arguments

`keySet`	the target key set

Description

Undoes all changes made to the *key database associated with* the specified key set since it was last opened, or since it was last the target of a call to `PGPRevertKeyRingChanges`.

Notes, Warnings, and Tips

The current implementation treats the target key set as an indirect parameter that references a key database, rather than as an explicit destination.

The indirect nature of this interface is likely to change in a future version, and will almost certainly involve changes to this function's parameterization.

PGPOrderKeySet

```
PGPError         PGPOrderKeySet(
                      PGPKeySetRef        keySet,
                      PGPKeyOrdering      order,
                      PGPKeyListRef       *keyList );
```

Arguments

`keySet`	the target key set
`order`	the ordering criteria, which recognizes `kPGP…Ordering` values (see Table 2-2)
`keyList`	the receiving field for the resultant ordered key list

Description

Creates a key list from the target key set with the specified ordering, suitable for iteration (see this chapter's section on key iterator functions)

Notes, Warnings, and Tips

The PGPsdk supports only single-level ordering. For example, this function does not support creation of a key list ordered by expiration date within encryption key size.

The caller is responsible for deallocating the resultant key set with `PGPFreeKeyList`.

PGPFreeKeyList

```
PGPError         PGPFreeKeyList(
                      PGPKeyListRef       keySet );
```

Arguments

`keySet`	the target key list

Description

Decrements the reference count for the specified key list, and frees the key list if the reference count reaches zero.

KeyFilter Functions

PGPNewKeyCreationTimeFilter

```
PGPError        PGPNewKeyCreationTimeFilter(
                     PGPContextRef      pgpContext,
                     PGPTime            creationTime,
                     PGPMatchCriterion  match,
                     PGPFilterRef       *outFilter );
```

Arguments

`pgpContext`	the target context
`creationTime`	the desired creation time value
`match`	the match criterion, which recognizes `kPGPMatch`… values (see Table 2-3)
`outFilter`	the receiving field for the resultant key filter

Description

Creates a key filter that will select those keys whose creation time meets the match criterion with respect to the specified creation time.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant key filter with `PGPFreeFilter`.

PGPNewKeyDisabledFilter

```
PGPError        PGPNewKeyDisabledFilter(
                     PGPContextRef      pgpContext,
                     PGPBoolean         disabled,
                     PGPFilterRef       *outFilter );
```

Arguments

`pgpContext`	the target context
`disabled`	`TRUE` to match `disabled` keys; `FALSE` to match enabled keys
`outFilter`	the receiving field for the resultant key filter

Description

Creates a key filter that will select for all disabled keys or for all enabled keys, depending on the value of the `disabled` argument.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant key filter with `PGPFreeFilter`.

PGPNewKeyEncryptAlgorithmFilter

```
PGPError         PGPNewKeyEncryptAlgorithmFilter(
                      PGPContextRef       pgpContext,
                      PGPPublicKeyAlgorithm
                                          encryptAlgorithm,
                      PGPFilterRef        *outFilter );
```

Arguments

`pgpContext`	the target context
`encryptAlgorithm`	the desired public key encryption algorithm, which recognizes `kPGPPublicKeyAlgorithm_`… values (see Table 3-5)
`outFilter`	the receiving field for the resultant key filter

Description

Creates a key filter that will select those keys that use the specified public key algorithm.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant key filter with `PGPFreeFilter`.

It may be useful to first determine if the desired public key encryption algorithm is available (see `PGPGetIndexdPublicKeyAlgorithmInfo`).

PGPNewKeyEncryptKeySizeFilter

```
PGPError         PGPNewKeyEncryptKeySizeFilter(
                      PGPContextRef       pgpContext,
                      PGPUInt32           keySize,
                      PGPMatchCriterion   match,
                      PGPFilterRef        *outFilter );
```

Arguments

`pgpContext`	the target context
`keySize`	the desired number of bits in the encryption key
`match`	the match criterion, which recognizes `kPGPMatch`… values (see Table 2-3)
`outFilter`	the receiving field for the resultant key filter

Description

Creates a key filter that will select those keys whose encryption key size (in bits) meets the match criterion with respect to the specified encryption key size.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant key filter with `PGPFreeFilter`.

PGPNewKeyExpirationTimeFilter

```
PGPError         PGPNewKeyExpirationTimeFilter(
                      PGPContextRef       pgpContext,
                      PGPTime             expirationTime,
                      PGPMatchCriterion   match,
                      PGPFilterRef        *outFilter );
```

Arguments

`pgpContext`	the target context
`expirationTime`	the desired expiration time value
`match`	the match criterion, which recognizes `kPGPMatch`… values (see Table 2-3)
`outFilter`	the receiving field for the resultant key filter

Description

Creates a key filter that will select those keys whose expiration time meets the match criterion with respect to the specified expiration time.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant key filter with `PGPFreeFilter`.

PGPNewKeyFingerPrintFilter

```
PGPError         PGPNewKeyFingerPrintFilter(
                      PGPContextRef       pgpContext,
                      void const          *fingerPrint,
                      PGPSize             fingerPrintLength,
                      PGPFilterRef        *outFilter );
```

Arguments

`pgpContext`	the target context
`fingerPrint`	the desired key **fingerprint**
`fingerPrintLength`	the size of the desired fingerprint
`outFilter`	the receiving field for the resultant key filter

Description

Creates a key filter that will select for those keys having the specified fingerprint.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant key filter with `PGPFreeFilter`.

PGPNewKeyRevokedFilter

```
PGPError         PGPNewKeyRevokedFilter(
                    PGPContextRef      pgpContext,
                    PGPBoolean         revoked,
                    PGPFilterRef       *outFilter );
```

Arguments

`pgpContext`	the target context
`revoked`	`TRUE` to match **revoked** keys; `FALSE` to match non-revoked keys
`outFilter`	the receiving field for the resultant key filter

Description

Creates a key filter that will select for all revoked keys or for all non-revoked keys, depending on the value of the `revoked` argument.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant key filter with `PGPFreeFilter`.

PGPNewSubKeyIDFilter

```
PGPError         PGPNewSubKeyIDFilter(
                      PGPContextRef      pgpContext,
                      PGPConstKeyIDRef   subKeyID,
                      PGPFilterRef       *outFilter );
```

Arguments

`pgpContext`	the target context
`subKeyID`	the desired sub-key ID
`outFilter`	the receiving field for the resultant key filter

Description

Creates a key filter that will select for the specified sub-key ID.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant key filter with `PGPFreeFilter`.

PGPNewUserIDEmailFilter

```
PGPError         PGPNewUserIDEmailFilter(
                      PGPContextRef      pgpContext,
                      char const         *emailString,
                      PGPMatchCriterion  match,
                      PGPFilterRef       *outFilter );
```

Arguments

```
pgpContext         the target context
emailString        the desired user e-mail address
match              the match criterion, which recognizes kPGPMatch… values (see Table 2-
                   3)
outFilter          the receiving field for the resultant key filter
```

Description

Creates a key filter that will select for keys whose user ID information contains the specified e-mail address.

Notes, Warnings, and Tips

The `emailString` argument length must not exceed `kPGPMaxUserIDLength`.

The caller is responsible for deallocating the resultant key filter with `PGPFreeFilter`.

PGPNewUserIDNameFilter

```
PGPError          PGPNewUserIDNameFilter(
                       PGPContextRef      pgpContext,
                       char const         *nameString,
                       PGPMatchCriterion  match,
                       PGPFilterRef       *outFilter );
```

Arguments

```
pgpContext         the target context
nameString         the desired user name
match              the match criterion, which recognizes kPGPMatch… values (see Table 2-
                   3)
outFilter          the receiving field for the resultant key filter
```

Description

Creates a key filter that will select for keys whose user ID information contains the specified user name.

Notes, Warnings, and Tips

Currently, the function effects the comparison as a sub-string match, and assumes a match criteria value of `kPGPMatchSubString`.

The `nameString` argument length must not exceed `kPGPMaxUserIDLength`.

The caller is responsible for deallocating the resultant key filter with `PGPFreeFilter`.

PGPNewUserIDStringFilter

```
PGPError          PGPNewUserIDStringFilter(
                       PGPContextRef      pgpContext,
                       char const         *userIDString,
                       PGPMatchCriterion  match,
                              PGPFilterRef       *outFilter );
```

Arguments

`pgpContext`	the target context
`userIDString`	the desired user ID
`match`	the match criterion, which recognizes `kPGPMatch`… values (see Table 2-3)
`outFilter`	the receiving field for the resultant key filter

Description

Creates a key filter that will select for keys whose user ID information matches the specified data string.

Notes, Warnings, and Tips

Currently, the function effects the comparison on the entire string, and assumes a match criteria value of `kPGPMatchEqual`.

The `userIDString` argument length must not exceed `kPGPMaxUserIDLength`.

The caller is responsible for deallocating the resultant key filter with `PGPFreeFilter`.

PGPNewKeySigAlgorithmFilter

```
PGPError         PGPNewKeySigAlgorithmFilter(
                      PGPContextRef      pgpContext,
                      PGPPublicKeyAlgorithm
                                         sigAlgorithm,
                      PGPFilterRef       *outFilter );
```

Arguments

`pgpContext`	the target context
`sigAlgorithm`	the desired signature algorithm
`outFilter`	the receiving field for the resultant key filter

Description

Creates a key filter that will select those keys using the specified signature algorithm.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant key filter with `PGPFreeFilter`.

PGPNewKeySigKeySizeFilter

```
PGPError    PGPNewKeySigKeySizeFilter(
                      PGPContextRef      pgpContext,
                      PGPUInt32          keySize,
                      PGPMatchCriterion  match,
                      PGPFilterRef       *outFilter );
```

Arguments

`pgpContext`	the target context
`keySize`	the desired number of bits in the signature key
`match`	the match criterion, which recognizes `kPGPMatch`… values (see Table 2-3)
`outFilter`	the receiving field for the resultant key filter

Description

Creates a key filter that will select those keys whose signature key size (in bits) meets the match criterion with respect to the specified signature key size.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant key filter with `PGPFreeFilter`.

PGPNewSigKeyIDFilter

```
PGPError         PGPNewSigKeyIDFilter(
                      PGPContextRef      pgpContext,
                      PGPKeyID const     *keyID,
                      PGPFilterRef       *outFilter );
```

Arguments

`pgpContext`	the target context
`keyID`	the desired signature key ID
`outFilter`	the receiving field for the resultant key filter

Description

Creates a key filter that will select those keys that were signed by the key having the specified key ID.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant key filter with `PGPFreeFilter`.

PGPNewKeyIDFilter

```
PGPError         PGPNewKeyIDFilter(
                      PGPContextRef      pgpContext,
                      PGPConstKeyIDRef   keyID,
                      PGPFilterRef       *outFilter );
```

Arguments

`pgpContext`	the target context
`keyID`	the desired key ID
`outFilter`	the receiving field for the resultant key filter

Description

Creates a key filter that will select for the specified key ID.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant key filter with `PGPFreeFilter`.

PGPNegateFilter

```
PGPError         PGPNegateFilter(
                      PGPFilterRef       filter,
                      PGPFilterRef       *outFilter );
```

Arguments

`filter`	the source key filter
`outFilter`	the receiving field for the resultant key filter

Description

Creates a new key filter that is the will select those keys that the input key filter will not.

Notes, Warnings, and Tips

This function does *not* use copy semantics - the input key filter is freed, even if the function returns an error.

The caller is responsible for deallocating the resultant key filter with `PGPFreeFilter`.

PGPIntersectFilters

```
PGPError         PGPIntersectFilters(
                      PGPFilterRef       filter1,
                      PGPFilterRef       filter2,
                      PGPFilterRef       *outFilter );
```

Arguments

`filter1`	the first source key filter
`filter2`	the second source key filter
`outFilter`	the receiving field for the resultant key filter

Description

Creates a new key filter that is the logical intersection of the two input key filters. For example, for the resultant key filter to select an item, that item would have to be selectable by both of the input key filters.

Errors

`kPGPError_InconsistentFilterClasses`

Notes, Warnings, and Tips

This function does *not* use copy semantics. The input key filters are freed, even if the function returns an error.

The caller is responsible for deallocating the resultant key filter with `PGPFreeFilter`.

PGPUnionFilters

```
PGPError          PGPUnionFilters(
                       PGPFilterRef        filter1,
                       PGPFilterRef        filter2,
                       PGPFilterRef        *outFilter );
```

Arguments

`filter1`	first input key filter
`filter2`	second input key filter
`outFilter`	the receiving field for the resultant key filter

Description

Creates a key filter that is the logical union of the two input key filters. For example, for the resultant key filter to select an item, that item would have to be selectable by either of the input key filters.

Errors

```
kPGPError_InconsistentFilterClasses
```

Notes, Warnings, and Tips

This function does *not* use copy semantics. The input key filters are freed, even if the function returns an error.

PGPFreeFilter

```
PGPError     PGPFreeFilter(
                       PGPFilterRef        filter );
```

Arguments

`filter`	the target key filter

Description

Decrements the reference count for the specified key filter, and frees the key filter if the reference count reaches zero.

PGPFilterKeySet

```
PGPError          PGPFilterKeySet(
                       PGPKeySetRef        origSet,
                       PGPFilterRef        filter,
                       PGPKeySetRef        *resultSet );
```

Arguments

`origSet`	the source key set
`filter`	the target key filter
`resultSet`	the receiving field for resultant key set

Description

Applies the specified key filter to the specified key set. This yields a resultant key set that contains all of the keys from the source key set that meet the key filter criteria.

Notes, Warnings, and Tips

The resultant key set may be empty.

PGPLDAPQueryFromFilter

```
PGPError         PGPLDAPQueryFromFilter(
                      PGPFilterRef       filter,
                      char               **queryOut );
```

Arguments

`filter`	the target key filter
`queryOut`	the receiving field for a pointer to the resultant LDAP key server format query string

Description

Converts the key filter criteria to an LDAP key server format query string, which can then be passed to the key server for processing.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant query string with `PGPFreeData`.

PGPHKSQueryFromFilter

```
PGPError         PGPHKSQueryFromFilter(
                      PGPFilterRef       filter,
                      char               **queryOut );
```

Arguments

`filter`	the target key filter
`queryOut`	the receiving field for a pointer to the resultant HTTP key server format query string

Description

Converts the key filter criteria to an HTTP key server format query string, which can then be passed to the key server for processing.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant query string with `PGPFreeData`.

A significant number of key filter options are not supported by HTTP key servers:

Key Iteration Functions

The PGPsdk includes support for passing through the keys in a key list or the sub-parts of an individual key. Note that whenever these functions return `kPGPError_EndOfIteration`, the caller should treat the iterator's value as being undefined.

PGPNewKeyIter

```
PGPError         PGPNewKeyIter(
                      PGPKeyListRef       keySet,
                      PGPKeyIterRef       *keyIter );
```

Arguments

`keySet`	the list of keys on which to iterate
`keyIter`	the receiving field for the iterator

Description

Creates an iterator on a list of keys.

Notes, Warnings, and Tips

A key list may have any number of iterators associated with it.

The caller is responsible for freeing the iterator with `PGPFreeKeyIter`.

PGPCopyKeyIter

```
PGPError         PGPCopyKeyIter(
                      PGPKeyIterRef       iterOrig,
                      PGPKeyIterRef       *iterCopy );
```

Arguments

`iterOrig`	the source iterator
`iterCopy`	the receiving field for the copy of the iterator

Description

Creates an exact copy of the source iterator, including its current index.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant iterator copy with `PGPFreeKeyIter`.

PGPFreeKeyIter

```
PGPError         PGPFreeKeyIter(
                      PGPKeyIterRef       iter );
```

Arguments

`iter`	the target iterator

Description

Decrements the reference count for the specified iterator, and frees the iterator if the reference count reaches zero.

PGPKeyIterIndex

```
PGPUInt32   PGPKeyIterIndex(
                      PGPKeyIterRef      iter );
```

Arguments

`iter`	the target iterator

Description

Returns the current index value of the specified iterator.

Notes, Warnings, and Tips

The caller should not infer anything based upon the returned index value.

PGPKeyIterKey

```
PGPError         PGPKeyIterKey(
                      PGPKeyIterRef      iter,
                      PGPKeyRef          *key );
```

Arguments

`iter`	the target iterator
`key`	the receiving field for the resultant key

Description

Yields the key associated with the current index value of the specified iterator.

Errors

`kPGPError_EndOfIteration`

Notes, Warnings, and Tips

`kPGPError_EndOfIteration` is only returned if the key has been deleted.

PGPKeyIterSubKey

```
PGPError         PGPKeyIterSubKey(
                      PGPKeyIterRef      iter,
                      PGPSubKeyRef       *subKey );
```

Arguments

`iter`	the target iterator
`subKey`	the receiving field for the resultant `sub-key`

Description

Yields the sub-key associated with the current index value of the specified iterator.

Errors

`kPGPError_EndOfIteration`

Notes, Warnings, and Tips

`kPGPError_EndOfIteration` is only returned if the sub-key has been deleted.

PGPKeyIterUserID

```
PGPError         PGPKeyIterUserID(
                      PGPKeyIterRef      iter,
                      PGPUserIDRef       *userID );
```

Arguments

`iter`	the target iterator
`userID`	the receiving field for the resultant user ID

Description

Yields the user ID associated with the current index value of the specified iterator.

Errors

`kPGPError_EndOfIteration`

Notes, Warnings, and Tips

`kPGPError_EndOfIteration` is only returned if the user ID has been deleted.

PGPKeyIterSig

```
PGPError         PGPKeyIterSig(
                      PGPKeyIterRef      iter,
                      PGPSigRef          *sig );
```

Arguments

`iter`	the target iterator
`sig`	the receiving field for the resultant signature

Description

Yields the signature associated with the current index value of the specified iterator.

Errors

`kPGPError_EndOfIteration`

Notes, Warnings, and Tips

`kPGPError_EndOfIteration` is only returned if the signature has been deleted.

PGPKeyIterMove

```
PGPError          PGPKeyIterMove(
                       PGPKeyIterRef       iter,
                       PGPInt32            relOffset,
                       PGPKeyRef           *key );
```

Arguments

`iter`	the target iterator
`relOffset`	the relative offset from the current position
`key`	the receiving field for the resultant key

Description

Moves the specified iterator by the specified relative offset, and yields the resultant key. Negative offsets move the iterator towards the beginning of the list; positive offsets move the iterator towards the end of the list.

Errors

`kPGPError_EndOfIteration`

Notes, Warnings, and Tips

If `kPGPError_EndOfIteration` is returned, then `key` will be set to `( PGPKeyRef * )NULL`.

If `kPGPError_EndOfIteration` is returned, then the resultant key may have been deleted.

PGPKeyIterSeek

```
PGPInt32    PGPKeyIterSeek(
                       PGPKeyIterRef       iter,
                       PGPKeyRef           key );
```

Arguments

`iter`	the target iterator
`key`	key to match

Description

Scans the key set associated with the iterator, and returns the index (zero-based) of the first key that matches the specified search-for key.

Notes, Warnings, and Tips

If the specified search-for key is not found, then the iterator is forcibly reset to point to the first key in the list. This should only happen if the search-for key was removed.

PGPKeyIterNext

```
PGPError          PGPKeyIterNext(
                       PGPKeyIterRef       iter,
                       PGPKeyRef           *key );
```

Arguments

`iter`	the target iterator
`key`	the receiving field for the resultant key

Description

Moves the specified iterator forward by one, and yields the resultant key.

Errors

`kPGPError_EndOfIteration`

Notes, Warnings, and Tips

This function is the equivalent of

```
      PGPKeyIterMove( iter, 1, &key );
```

If `kPGPError_EndOfIteration` is returned, then `key` will be set to `( PGPKeyRef * )NULL`.

If `kPGPError_EndOfIteration` is returned, then the resultant key may have been deleted.

PGPKeyIterNextSubKey

```
PGPError          PGPKeyIterNextSubKey(
                       PGPKeyIterRef       iter,
                       PGPSubKeyRef        *subKey );
```

Arguments

`iter`	the target iterator
`subKey`	the receiving field for the resultant sub-key

Description

Moves the specified iterator forward by one, and yields the resultant sub-key associated with the current key.

Errors

`kPGPError_EndOfIteration`

Notes, Warnings, and Tips

If `kPGPError_EndOfIteration` is returned, then `subKey` will be set to `( PGPSubKeyRef * )NULL`.

If `kPGPError_EndOfIteration` is returned, then the resultant sub-key may have been removed.

PGPKeyIterNextUserID

```
PGPError          PGPKeyIterNextUserID(
                       PGPKeyIterRef      iter,
                       PGPUserIDRef       *userID );
```

Arguments

`iter`	the target iterator
`userID`	the receiving field for the resultant `userID`

Description

Moves the specified iterator forward by one, and yields the resultant user ID associated with the current key.

Errors

`kPGPError_BadParams`
The current key has no associated user ID, or the associated user ID has been removed.

`kPGPError_EndOfIteration`

Notes, Warnings, and Tips

If `kPGPError_EndOfIteration` is returned, then `userID` will be set to `( PGPUserIDRef * )NULL`.

PGPKeyIterNextUIDSig

```
PGPError          PGPKeyIterNextUIDSig(
                       PGPKeyIterRef      iter,
                       PGPSigRef          *sig );
```

Arguments

`iter`	the target iterator
`sig`	the receiving field for the resultant signature

Description

Moves the specified iterator forward by one, and yields the resultant signature associated with the current user ID of the current key.

Errors

`kPGPError_BadParams`
The current key has no associated user ID, or the associated user ID has been removed.

`kPGPError_EndOfIteration`

Notes, Warnings, and Tips

If `kPGPError_EndOfIteration` is returned, then `sig` will be set to `( PGPSigRef * )NULL`.

PGPKeyIterPrev

```
PGPError         PGPKeyIterPrev(
                      PGPKeyIterRef       iter,
                      PGPKeyRef           *key );
```

Arguments

`iter`	the target iterator
`key`	the receiving field for the resultant key

Description

Moves the specified iterator backward by one, and yields the resultant key.

Errors

`kPGPError_EndOfIteration`

Notes, Warnings, and Tips

This function is the equivalent of

```
PGPKeyIterMove( iter, -1, &key );
```

If `kPGPError_EndOfIteration` is returned, then `key` will be set to `( PGPKeyRef * )NULL`.

If `kPGPError_EndOfIteration` is returned, then the resultant key may have been deleted.

PGPKeyIterPrevSubKey

```
PGPError         PGPKeyIterPrevSubKey(
                      PGPKeyIterRef       iter,
                      PGPKeyRef           *key );
```

Arguments

`iter`	the target iterator
`key`	the receiving field for the resultant sub-key

Description

Moves the specified iterator backward by one, and yields the resultant sub-key associated with the current key.

Errors

`kPGPError_EndOfIteration`

Notes, Warnings, and Tips

If `kPGPError_EndOfIteration` is returned, then the resultant sub-key may have been removed.

PGPKeyIterPrevUserID

```
PGPError          PGPKeyIterPrevUserID(
                       PGPKeyIterRef      iter,
                       PGPUserIDRef       *userID );
```

Arguments

`iter`	the target iterator
`userID`	the receiving field for the resultant user ID

Description

Moves the specified iterator forward by one, and yields the resultant user ID associated with the current key.

Errors

`kPGPError_BadParams`

The current key has no associated user ID, or the associated user ID has been removed.

`kPGPError_EndOfIteration`

Notes, Warnings, and Tips

If `kPGPError_EndOfIteration` is returned, then `userID` will be set to `( PGPUserIDRef * )NULL`.

PGPKeyIterPrevUIDSig

```
PGPError          PGPKeyIterPrevUIDSig(
                       PGPKeyIterRef      iter,
                       PGPSigRef          *sig );
```

Arguments

`iter`	the target iterator
`sig`	the receiving field for the resultant signature

Description

Moves the specified iterator backward by one, and yields the resultant signature associated with the current user ID of the current key.

Errors

`kPGPError_BadParams`

The current key has no associated user ID, or the associated user ID has been removed.

`kPGPError_EndOfIteration`

Notes, Warnings, and Tips

If `kPGPError_EndOfIteration` is returned, then `sig` will be set to `( PGPSigRef * )NULL`.

PGPKeyIterRewind

```
PGPError         PGPKeyIterRewind(
                      PGPKeyIterRef      iter );
```

Arguments

`iter`	the target iterator

Description

Resets the iterator such that a subsequent `PGPKeyIterNext` will yield the first key in the associated key list.

PGPKeyIterRewindSubKey

```
PGPError         PGPKeyIterRewindSubKey(
                      PGPKeyIterRef      iter );
```

Arguments

`iter`	the target iterator

Description

Resets the iterator such that a subsequent `PGPKeyIterNextSubKey` will yield first sub-key associated with the current key.

PGPKeyIterRewindUserID

```
PGPError         PGPKeyIterRewindUserID(
                      PGPKeyIterRef      iter );
```

Arguments

`iter`	the target iterator

Description

Resets the iterator such that a subsequent `PGPKeyIterNextUserID` will yield the first user ID associated with the key.

PGPKeyIterRewindUIDSig

```
PGPError         PGPKeyIterRewindUIDSig(
                      PGPKeyIterRef      iter );
```

Arguments

`iter`	the target iterator

Description

Resets the iterator such that a subsequent `PGPKeyIterNextUIDSig` will yield the first signature associated with the current user ID of the current key.

Key Reference Count Functions

The PGPsdk automatically tracks the number of data items pointing to a particular resource. For example, a given key set may be referenced by any number of key lists and/or key iterators. This not only results in a level of context independence, but also ensures that a resource's memory is released only when its last reference is deleted. The PGPsdk also provides functions to support manual adjustment of a data item's reference count.

PGPIncKeySetRefCount

```
PGPError          PGPIncKeySetRefCount(
                       PGPKeySetRef        keySet );
```

Arguments

`keySet`	the target key set

Description

Increments the reference count of the specified key set. This provides a mechanism for manually incrementing the reference count should it be necessary.

PGPIncFilterRefCount

```
PGPError          PGPIncFilterRefCount(
                       PGPFilterRef        filter );
```

Arguments

`filter`	the target key filter

Description

Increments the reference count of the specified key filter. This provides a mechanism for manually incrementing the reference count should it be necessary.

PGPIncKeyListRefCount

```
PGPError          PGPIncKeyListRefCount(
                       PGPKeyListRef       keySet );
```

Arguments

`keySet`	the target key list

Description

Increments the reference count of the specified key list. This provides a mechanism for manually incrementing the reference count should it be necessary.

Key Manipulation Functions

PGPGenerateKey

```
PGPError        PGPGenerateKey(
                     PGPContextRef       pgpContext,
                     PGPKeyRef           *key,
                     PGPOptionListRef    firstOption,
                     ...,
                     PGPOLastOption( void ) );
```

Arguments

`pgpContext`	the target context
`key`	the receiving field for the generated key
`firstOption`	the initial option list instance
`...`	subsequent option list instances
`PGPOLastOption( void )`	must always appear as the final argument to terminate the argument list

Description

Generates a new key according to the specified options.

Errors

`kPGPError_OutOfEntropy`

Notes, Warnings, and Tips

Sufficient **entropy** must be available for this function to succeed.

The current implementation treats any destination key set specified with `PGPOKeySetRef` as an indirect parameter that references a key database, rather than as an explicit destination. Because of key filtering and the “live” nature of its resultant view-style key sets, the key generated by this function may appear in any key set based upon that key database, and further may or may not appear in the specified destination key set, depending upon its key filtering criteria.

The indirect nature of this interface is likely to change in a future version, and will almost certainly involve changes to this function’s parameterization.

PGPChangePassphrase

```
PGPError        PGPChangePassphrase(
                     PGPKeyRef           key,
                     char const          *oldphrase,
                     char const          *newphrase );
```

Arguments

`key`	the target key
`oldphrase`	the current **passphrase**
`newphrase`	the new passphrase

Description

Changes the passphrase for the specified key.

Notes, Warnings, and Tips

If sub-key(s) exist, then their passphrases should first be changed via `PGPChangeSubKeyPassphrase`.

PGPEnableKey

```
PGPError         PGPEnableKey(
                      PGPKeyRef           key );
```

Arguments

`key`	the target key

Description

Marks a key as enabled for encryption and signing.

PGPDisableKey

```
PGPError         PGPDisableKey(
                      PGPKeyRef           key );
```

Arguments

`key`	the target key

Description

Marks a key as disabled for encryption and signing.

Notes, Warnings, and Tips

The target key is still enabled for decryption and verifying.

PGPRevokeKey

```
PGPError         PGPRevokeKey(
                      PGPKeyRef           key,
                      PGPOptionListRef    firstOption,
                      ...,
                      PGPOLastOption( void ) );
```

Arguments

`key`	the key to be revoked
`firstOption`	the initial option list instance
`...`	subsequent option list instances
`PGPOLastOption( void )`	must always appear as the final argument to terminate the argument list

Description

Revokes the specified key.

Notes, Warnings, and Tips

In order to successfully revoke a key, its passphrase must be known. This implies that the function must minimally be passed a `PGPOPassphrase` or `PGPOPassphraseBuffer` option list instance.

Sample Code

```
err = PGPRevokeKey( pgpContext,
                    key,
                    PGPOPassphrase( pgpContext,
                                   “Please don’t hardcode passphrases – EVER!” ),
                     PGPOLastOption( void ) );
```

PGPSetKeyAxiomatic

```
PGPError        PGPSetKeyAxiomatic(
                     PGPKeyRef           key,
                     PGPBoolean          checkPassphrase,
                     char const          *passphrase );
```

Arguments

`key`	the target key
`checkPassphrase`	`TRUE` if a passphrase is included and should be checked as being valid for the target key
`passphrase`	the assumed passphrase for the target key or `NULL`

Description

Forces the specified key to be axiomatically trusted. If `checkPassphrase` is `TRUE`, then passphrase must be both non-`NULL` and valid for the specified key. See `PGPUnSetKeyAxiomatic`.

Errors

```
kPGPError_BadPassphrase
kPGPError_KeyExpired
kPGPError_KeyRevoked
kPGPError_ItemIsReadOnly
```

Notes, Warnings, and Tips

The specifed key must be enabled and mutable.

Unless the key has just been created, a passphrase should be required to set such an unconditional trust level.

Sample Code

```
err = PGPSetKeyAxiomatic( key1,
                          FALSE,
                          NULL );

err = PGPSetKeyAxiomatic( key2,
                          TRUE,
                          “Please don’t hardcode passphrases – EVER!” );
```

PGPUnsetKeyAxiomatic

```
PGPError        PGPUnsetKeyAxiomatic(
                     PGPKeyRef          key );
```

Arguments

`key`	the target key

Description

Removes the axiomatic trust from the specified key (see `PGPSetKeyAxiomatic`).

Errors

```
kPGPError_KeyExpired
kPGPError_KeyRevoked
kPGPError_ItemIsReadOnly
```

Notes, Warnings, and Tips

The specifed key must be enabled and mutable.

PGPSetKeyTrust

```
PGPError        PGPSetKeyTrust(
                     PGPKeyRef          key,
                     PGPUInt32          trust );
```

Arguments

`key`	the target key
`trust`	the desired trust level, which recognizes `kPGPKeyTrust_`… values (see Table 3-7a)

Description

Set the trust level of the specified key to that specified.

Errors

```
kPGPError_KeyExpired
kPGPError_KeyRevoked
kPGPError_ItemIsReadOnly
```

Notes, Warnings, and Tips

The specifed key must be enabled and mutable.

`kPGPKeyTrust_Undefined` and `kPGPKeyTrust_Ultimate` may not be used as `trust` argument values.

PGPCompareKeys

```
PGPInt32    PGPCompareKeys(
                      PGPKeyRef          key1,
                      PGPKeyRef          key2,
                      PGPKeyOrdering     order );
```

Arguments

`key1`	the first target key
`key2`	the second target key
`order`	the ordering to be applied to the target keys, which recognizes `kPGP…Ordering` values (see Table 2-2)

Description

Compares the specified keys according to the specified ordering, and returns `–1`, `0`, or `1` depending on whether or not `key1` is less than, equal to, or greater than `key2`.

Notes, Warnings, and Tips

If the keys compare as equal with respect to the specified ordering, then the result reflects a comparison of their associated key IDs.

If both keys are found to be invalid, then the function returns `0` (zero).

PGPGenerateSubKey

```
PGPError         PGPGenerateSubKey(
                      PGPContextRef      pgpContext,
                      PGPSubKeyRef       *subkey,
                      PGPOptionListRef   firstOption,
                      ...,
                      PGPOLastOption( void ) );
```

Arguments

`pgpContext`	the target context
`subkey`	the receiving field for the generated sub-key
`firstOption`	the initial option list instance
`...`	subsequent option list instances
`PGPOLastOption( void )`	must always appear as the final argument to terminate the argument list

Description

Generates a new sub-key according to the specified options.

Errors

`kPGPError_OutOfEntropy`

Notes, Warnings, and Tips

Enough entropy must be available for this function to succeed.

The current implementation treats any destination key set specified with `PGPOKeySetRef` as an indirect parameter that references a key database, rather than as an explicit destination. Because of key filtering and the “live” nature of its resultant view-style key sets, the sub-key generated by this function may be reflected in any key set based upon that key database, and further may or may not be reflected in the specified destination key set, depending upon its key filtering criteria.

The indirect nature of this interface is likely to change in a future version, and will almost certainly involve changes to this function’s parameterization.

PGPRemoveSubKey

```
PGPError         PGPRemoveSubKey(
                      PGPSubKeyRef       subkey );
```

Arguments

`subkey`	the target sub-key

Description

Removes the specified sub-key from its associated key.

PGPChangeSubKeyPassphrase

```
PGPError         PGPChangeSubKeyPassphrase(
                      PGPSubKeyRef       subkey,
                      char const         *oldphrase,
                      char const         *newphrase );
```

Arguments

`subkey`	the target sub-key
`oldphrase`	the current passphrase
`newphrase`	the new passphrase

Description

Changes the passphrase for the specified sub-key.

PGPRevokeSubKey

```
PGPError         PGPRevokeSubKey(
                      PGPSubKeyRef       subkey,
                      PGPOptionListRef   firstOption,
                      ...,
                      PGPOLastOption( void ) );
```

Arguments

`subkey`	the target sub-key
`firstOption`	the initial option list instance
`...`	subsequent option list instances
`PGPOLastOption( void )`	must always appear as the final argument to terminate the argument list

Description

Revokes the specified sub-key.

PGPAddUserID

```
PGPError          PGPAddUserID(
                       PGPKeyRef           key,
                       char const          *name,
                       PGPOptionListRef    firstOption,
                       ...,
                       PGPOLastOption( void ) );
```

Arguments

`key`	the key to add the user ID to
`name`	a character string (the user ID)
`firstOption`	the initial option list instance
`...`	subsequent option list instances
`PGPOLastOption( void )`	must always appear as the final argument to terminate the argument list

Description

Creates an additional user ID for the specified key, and sets the user ID information to that specified.

Notes, Warnings, and Tips

The `name` argument length must not exceed `kPGPMaxUserIDLength`.

Keys may have multiple user IDs. The user ID added by this function will be put on the “bottom” of the list of user ID’s for this key.

PGPRemoveUserID

```
PGPError          PGPRemoveUserID(
                       PGPUserIDRef        userID );
```

Arguments

`userID`	the target user ID

Description

Removes the specified user ID from its associated key.

PGPSetPrimaryUserID

```
PGPError          PGPSetPrimaryUserID(
                       PGPUserIDRef       userID );
```

Arguments

`userID`	the target user ID

Description

Makes the specified user ID the primary user ID for its associated key.

Errors

```
kPGPError_KeyExpired
kPGPError_KeyRevoked
kPGPError_ItemIsReadOnly
```

Notes, Warnings, and Tips

The specifed key must be enabled and mutable.

PGPCompareUserIDStrings

```
PGPInt32          PGPCompareUserIDStrings(
                       char const         *userIDString1,
                       char const         *userIDString2 );
```

Arguments

`userIDString1`	the first target user ID string
`userIDString2`	the second target user ID string

Description

Compares the specified user ID strings, and returns `–1`, `0`, or `1` depending on whether or not `userIDString2` is less than, equal to, or greater than `userIDString1`.

Notes, Warnings, and Tips

The `userIDString`*n* arguments length must not exceed `kPGPMaxUserIDLength`.

If the user ID strings compare as equal, then the result reflects a comparison of the associated key IDs.

If either `userIDString1` or `userIDString2` is `NULL`, then the function returns `0` (zero).

PGPSignUserID

```
PGPError          PGPSignUserID(
                       PGPUserIDRef       userID,
                       PGPKeyRef          signingKey,
                       PGPOptionListRef   firstOption,
                       ...,
                        PGPOLastOption( void ) );
```

Arguments

`userID`	the target user ID
`signingKey`	the desired signing key
`firstOption`	the initial option list instance
`...`	subsequent option list instances
`PGPOLastOption( void )`	must always appear as the final argument to terminate the argument list

Description

Signs the key associated with the specified user ID with the specified signing key. Available options include:

- `PGPOPassphrase` – specifies the passphrase required to unlock the target key
- `PGPOPassphraseBuffer` – specifies the passphrase required to unlock the target key
- `PGPOExpiration` – specifies the expiration date of the signature
- `PGPOExportable` – specifies whether or not the key component may be exported
- `PGPOSigTrust` – specifies the trust level of the signature, which recognizes `kPGPNameTrust_`… values (see Table 3-7b)
- `PGPOSigRegularExpression`

Errors

```
kPGPError_KeyExpired
kPGPError_KeyRevoked
kPGPError_ItemIsReadOnly
```

Notes, Warnings, and Tips

The specifed key must be enabled and mutable.

Only one of `PGPOPassphrase` and `PGPOPassphraseBuffer` is allowed.

Sample Code

```
err = PGPSignUserID( userID,
                     signingKey,
                     PGPOptionPassPhraseBuffer( pgpContext,
                                                &ppBuf,
                                                ppBufCount );
                     PGPOExpiration( pgpContext,
                                     ( 180 * ( 24 * 60 * 60 ) ) ),
                     PGPOLastOption( void ) );
```

PGPRemoveSig

```
PGPError          PGPRemoveSig(
                        PGPSigRef          sig );
```

Arguments

`sig`	the signature to be removed

Description

Removes the specified signature from its associated user ID of the associated key.

PGPRevokeSig

```
PGPError        PGPRevokeSig(
                     PGPSigRef          sig,
                     PGPKeySetRef       keySet,
                     PGPOptionListRef   firstOption,
                     ...,
                     PGPOLastOption( void ) );
```

Arguments

`sig`	the target signature
`keySet`	the target key set
`firstOption`	the initial option list instance
`...`	subsequent option list instances
`PGPOLastOption( void )`	must always appear as the final argument to terminate the argument list

Description

Revokes the specified signature from all keys in the *key database associated with* the specified target key set.

Notes, Warnings, and Tips

The current implementation treats the destination key set as an indirect parameter that references a key database, rather than as an explicit destination. Because of key filtering and the “live” nature of its resultant view-style key sets, the signature revoked by this function may be reflected in any key set based upon that key database, and further may or may not be reflected in the specified destination key set, depending upon its key filtering criteria.

The indirect nature of this interface is likely to change in a future version, and will almost certainly involve changes to this function’s parameterization.

PGPCountAdditionalRecipientRequests

```
PGPError        PGPCountAdditionalRecipientRequests(
                     PGPKeyRef          baseKey,
                     PGPUInt32          *numARRKeys );
```

Arguments

`baseKey`	the target key
`numARRKeys`	the receiving field for the resultant count

Description

Provides the number of **additional recipient request keys** that are available for the specified base key.

Notes, Warnings, and Tips

Use this count as the upper limit when indexing through the available additional recipient keys (see the sample code for `PGPGetIndexedAdditionalRecipientRequest`).

PGPGetIndexedAdditionalRecipientRequest

```
PGPError        PGPGetIndexedAdditionalRecipientRequest(
                     PGPKeyRef           baseKey,
                     PGPKeySetRef        arrKeySet,
                     PGPUInt32           index,
                     PGPKeyRef           *arrKey,
                     PGPByte             *arrKeyClass );
```

Arguments

`baseKey`	the target key
`arrKeySet`	the look-up key set
`index`	the index (zero-based) of the desired additional recipient request key
`arrKey`	the receiving field for the *n*th additional recipient request key
`arrKeyClass`	the receiving field for the class of the additional recipient request key

Description

Provides a means of indexing through the available additional recipient request keys and retrieving each key and its class. All available additional recipient request keys are presumed to reside in the *key database associated with* the look-up key set.

Errors

`kPGPError_ItemNotFound`
`kPGPError_OutOfRings`

Notes, Warnings, and Tips

The current implementation treats the look-up key set as an indirect parameter that references a key database, rather than as an explicit destination.

The indirect nature of this interface is likely to change in a future version, and will almost certainly involve changes to this function's parameterization.

Sample Code

```
PGPUInt32       index;
PGPUInt32       numARRKeys;
PGPError        err;
PGPKeySetRef    arrKeySet;
```

```
    PGPKeyRef       arrKey;
    PGPByte         arrClass;

    if  ( IsPGPError( err = PGPCountAdditionalRecipientRequests( baseKey
                                                            &numARRKeys ) ) )
    {
        return( err );
    }

    for ( index = 0,; index < numARRKeys; index++ )
    {
        if  ( IsPGPError( err = PGPGetIndexedAdditionalRecipientRequest( baseKey,
                                                                  arrKeySet,
                                                                  index,
                                                                  &arrKey
                                                                  &arrKeyClass ) ) )
        {
              return( err );
        }
        /*
        ** Process the ARRKeys
        */
    }

    if  ( index >= numARRKeys )
    {
        /*
        ** Being here means that there were no ARRKeys
        */
    }

    return( kPGPError_NoErr );
```

PGPGetKeyEntropyNeeded

```
PGPUInt32   PGPGetKeyEntropyNeeded(
                   PGPContextRef       pgpContext,
                   PGPOptionListRef    firstOption,
                   ...,
                   PGPOLastOption( void ) );
```

Arguments

`pgpContext`	the target context
`firstOption`	the initial option list instance
`...`	subsequent option list instances
`PGPOLastOption( void )`	must always appear as the final argument to terminate the argument list

Description

Returns the amount of entropy needed to generate a (sub-)key according to the specified key generation parameters (see `PGPOKeyGenParams`).

PGPGetSigCertifierKey

```
PGPError         PGPGetSigCertifierKey(
                      PGPSigRef           sig,
                      PGPKeySetRef        allKeys,
                      PGPKeyRef           *sigKey );
```

Arguments

`sig`	the target signature
`allKeys`	the target key set
`sigKey`	the receiving field for the key that signed the target signature

Description

Searches the specified key set for the key that signed the specified signature.

Errors

`kPGPError_ItemNotFound`

PGPPassphraseIsValid

```
PGPBoolean       PGPPassphraseIsValid(
                      PGPKeySetRef        key,
                      const char          *passphrase );
```

Arguments

`key`	the target key
`passphrase`	the assumed associated passphrase

Description

Returns TRUE if the specified passphrase is valid for the specified key.

Get Property Functions

PGPGetHashAlgUsed

```
PGPError         PGPGetHashAlgUsed(
                      PGPKeyRef           key,
                      PGPHashAlgorithm    *hashAlg );
```

Arguments

`key`	the target key
`hashAlg`	the receiving field for the hash algorithm value, which recognizes `kPGPHashAlgorithm_`… values (see Table 3-3)

Description

Obtains the hash algorithm associated with the target key.

PGPGetKeyBoolean

```
PGPError         PGPGetKeyBoolean(
                      PGPKeyRef           key,
                      PGPKeyPropName      propName,
                      PGPBoolean          *propData );
```

Arguments

`key`	the target key
`propName`	the name of the target property, which recognizes `kPGPKeyProp…` values (see Table 2-4)
`propData`	the receiving field for the target property value

Description

Retrieves the value of the specified boolean property of the specified key.

Notes, Warnings, and Tips

If RSA encryption is not available (for example, a version of the PGPsdk that supports only **Diffie-Hellman** encryption), then `propData` will be `FALSE` for both `kPGPKeyPropCanSign` and `kPGPKeyPropCanEncrypt`.

Sample Code

```
PGPBoolean      keyIsSecret;

err = PGPGetKeyBoolean( key,
                        kPGPKeyPropIsSecret,
                        &keyIsSecret );
if ( ( err == kPGPError_NoErr ) && ( keyIsSecret ) )
{
     /*
     ** Process secret key
     */
}
```

PGPGetKeyNumber

```
PGPError         PGPGetKeyNumber(
                      PGPKeyRef           key,
                      PGPKeyPropName      propName,
                      PGPUInt32           *propData );
```

Arguments

`key`	the target key
`propName`	the name of the desired property, which recognizes `kPGPKeyProp…` values (see Table 2-4)
`propData`	the receiving field for the desired property value

Description

Retrieves the value of the specified numeric property of the specified key.

PGPGetKeyPropertyBuffer

```
PGPError         PGPGetKeyPropertyBuffer(
                      PGPKeyRef           key,
                      PGPKeyPropName      propName,
                      PGPSize             availLength,
                      void                *propData,
                      PGPSize             *usedLength );
```

Arguments

`key`	the target key
`propName`	the name of the desired property, which recognizes `kPGPKeyProp…` values (see Table 2-4)
`availLength`	the length of the receiving field for the desired property data
`propData`	the receiving field for the desired property data
`usedLength`	the receiving field for the resultant length of the desired property data

Description

Retrieves the arbitrary binary data associated with the specified property of the specified key.

Errors

`kPGPError_BufferTooSmall`

Notes, Warnings, and Tips

For a `propName` value of `kPGPPropPreferredAlgorithm`, a return value of `kPGPError_NoErr` with a resultant `usedLength` of zero indicates that no preferred algorithm is set.

Sample Code

```
PGPSize          usedLength;
PGPByte          keyPropBuffer[ 256 ];

err = PGPGetKeyNumber( key,
                       kPGPKeyPropPreferredAlgorithm,
                       (PGPSize)sizeof(keyPropBuffer),
                       &keyPropBuffer[ 0 ],
                       &usedLength );

if ( IsPGPError( err )
{
     return( err );
}

if ( usedLength == 0 )
{
     /*
     ** Handle no preferred algorithm set
     */
}
```

PGPGetKeyTime

```
PGPError          PGPGetKeyTime(
                       PGPKeyRef           key,
                       PGPKeyPropName      propName,
                       PGPTime             *propData );
```

Arguments

`key`	the target key
`propName`	the name of the desired property, which recognizes `kPGPKeyProp…` values (see Table 2-4)
`propData`	the receiving field for the desired property value

Description

Retrieves the value of the specified date/time property of the specified key.

PGPGetSubKeyBoolean

```
PGPError          PGPGetSubKeyBoolean(
                       PGPSubKeyRef        subkey,
                       PGPKeyPropName      propName,
                       PGPBoolean          *propData );
```

Arguments

`subkey`	the target sub-key
`propName`	the name of the desired property, which recognizes `kPGPKeyProp…` values (see Table 2-4)
`propData`	the receiving field for the desired property data

Description

Retrieves the value of the specified boolean property of the specified sub-key.

Notes, Warnings, and Tips

Keys and sub-keys share the same `propName` values.

PGPGetSubKeyNumber

```
PGPError          PGPGetSubKeyNumber(
                       PGPSubKeyRef        subkey,
                       PGPKeyPropName      propName,
                       PGPUInt32           *propData );
```

Arguments

`subkey`	the target sub-key
`propName`	which property you want to retrieve, which recognizes `kPGPKeyProp…` values (see Table 2-4)
`propData`	the receiving field for the desired property

Description

Retrieves the value of the specified numeric property of the specified sub-key.

Notes, Warnings, and Tips

Keys and sub-keys share the same `propName` values.

PGPGetSubKeyPropertyBuffer

```
PGPError          PGPGetSubKeyPropertyBuffer(
                       PGPSubKeyRef        subkey,
                       PGPKeyPropName      propName,
                       PGPSize             availLength,
                       void                *propData,
                       PGPSize             *usedLength );
```

Arguments

`subkey`	the target sub-key
`propName`	the name of the desired property, which recognizes `kPGPKeyProp…` values (see Table 2-4)
`availLength`	the length of the receiving field for the desired property data
`propData`	the receiving field for the desired property data
`usedLength`	the receiving field for the resultant length of the desired property data

Description

Retrieves the arbitrary binary data associated with the specified property of the specified sub-key.

Errors

`kPGPError_BufferTooSmall`

Notes, Warnings, and Tips

Keys and sub-keys share the same `propName` values.

For a `propName` value of `kPGPPropPreferredAlgorithm`, a return value of `kPGPError_NoErr` with a resultant `usedLength` of zero indicates that no preferred algorithm is set.

PGPGetSubKeyTime

```
PGPError          PGPGetSubKeyTime(
                       PGPSubKeyRef        subkey,
                       PGPKeyPropName      propName,
                       PGPTime             *propData );
```

Arguments

`subkey`	the target sub-key
`propName`	the name of the desired property, which recognizes `kPGPKeyProp…` values (see Table 2-4)
`propData`	the receiving field for the desired property value

Description

Retrieves the value of the specified date/time property of the specified sub-key.

Notes, Warnings, and Tips

Keys and sub-keys share the same `propName` values.

PGPGetUserIDNumber

```
PGPError         PGPGetUserIDNumber(
                      PGPUserIDRef        userID,
                      PGPUserIDPropName   propName,
                      PGPUInt32           *propData );
```

Arguments

`userID`	the target user ID
`propName`	the name of the desired property, which recognizes `kPGPUserIDProp…` values (see Table 2-5)
`propData`	the receiving field for the desired property value

Description

Retrieves the value of the specified numeric property of the specified key.

Notes, Warnings, and Tips

Keys and sub-keys share the same `propName` values.

PGPGetUserIDStringBuffer

```
PGPError         PGPGetUserIDStringBuffer(
                      PGPUserIDRef        userID,
                      PGPUserIDPropName   propName,
                      PGPSize             availLength,
                      char                *propString,
                      PGPSize             *usedLength );
```

Arguments

`userID`	the target user ID
`propName`	the name of the desired property, which recognizes `kPGPUserIDProp…` values (see Table 2-5)
`availLength`	the length of the receiving field for the desired property data
`propString`	the receiving field for the desired property data
`usedLength`	the receiving field for the resultant length of the desired property data

Description

Retrieves the *C* language string associated with the specified property of the specified user ID.

Errors

`kPGPError_BufferTooSmall`

Notes, Warnings, and Tips

`propString` should be a minimum of 256 bytes.

`usedLength` does *not* include the terminating `NUL`.

PGPGetSigBoolean

```
PGPError         PGPGetSigBoolean(
                      PGPSigRef           sig,
                      PGPSigPropName      propName,
                      PGPBoolean          *propData );
```

Arguments

`sig`	the target signature
`propName`	the name of the desired property, which recognizes `kPGPSigProp…` values (see Table 2-6)
`propData`	the receiving field for the desired property data

Description

Retrieves the value of the specified boolean property of the specified signature.

PGPGetSigNumber

```
PGPError         PGPGetSigNumber(
                      PGPSigRef           sig,
                      PGPSigPropName      propName,
                      PGPUInt32           *propData );
```

Arguments

`sig`	the target signature
`propName`	the name of the desired property, which recognizes `kPGPSigProp…` values (see Table 2-6)
`propData`	the receiving field for the desired property data

Description

Retrieves the value of the specified numeric property of the specified signature.

PGPGetSigTime

```
PGPError        PGPGetSigTime(
                     PGPSigRef          sig,
                     PGPSigPropName     propName,
                     PGPTime            *propData );
```

Arguments

`sig`	the target signature
`propName`	the name of the desired property, which recognizes `kPGPSigProp…` values (see Table 2-6)
`propData`	the receiving field for the desired property data

Description

Retrieves the value of the specified date/time property of the specified signature.

Convenience Property Functions

The “convenience property functions” encapsulate code that

- creates an iterator on the associated item
- applies it to the specified key
- outputs the associated property value
- frees the iterator

PGPGetPrimaryUserID

```
PGPError        PGPGetPrimaryUserID(
                     PGPKeyRef          key,
                     PGPUserIDRef       *userID );
```

Arguments

`key`	the target key
`userID`	the receiving field for the associated primary user ID

Description

Obtains the primary user ID of the specified key.

PGPGetPrimaryUserIDNameBuffer

```
PGPError         PGPGetPrimaryUserIDNameBuffer(
                      PGPKeyRef          key,
                      PGPSize            availLength,
                      char               *nameBuf,
                      PGPSize            *usedLength );
```

Arguments

`key`	the target key
`availLength`	the length of the receiving field for the associated primary user ID name
`nameBuf`	the receiving field for the associated primary user ID name
`usedLength`	the receiving field for the resultant length of the primary user ID name

Description

Retrieves the primary user ID name associated with the specified key, which is assumed to be a *C* language string.

Errors

`kPGPError_BufferTooSmall`

Notes, Warnings, and Tips

The `nameBuf` argument length must be at least (`kPGPMaxUserIDLength + 1`).

`usedLength` does *not* include the terminating NUL.

PGPGetPrimaryUserIDValidity

```
PGPError         PGPGetPrimaryUserIDValidity(
                      PGPKeyRef          key,
                      PGPValidity        *validity );
```

Arguments

`key`	the target key
`validity`	the receiving field for the validity value associated with the user ID of the target key, which recognizes `kPGPValidity_`… values (see Table 3-8)

Description

Obtains the validity of the primary user ID associated with the specified key.

Errors

`kPGPError_ItemNotFound`
`kPGPValidity_Unknown`

Default Private Key Functions

PGPSetDefaultPrivateKey

```
PGPError          PGPSetDefaultPrivateKey(
                       PGPKeyRef          key );
```

Arguments

`key`	the target key

Description

Sets the default private key (nominally used for signing) to the specified key.

Errors

```
kPGPError_KeyExpired
kPGPError_KeyRevoked
kPGPError_ItemIsReadOnly
```

Notes, Warnings, and Tips

The target key must be a **secret key** (`kPGPKeyPropIsSecret`), and must be able to sign (`kPGPKeyPropCanSign`).

The target key is forced to be axiomatically trusted (no passphrase is required).

PGPGetDefaultPrivateKey

```
PGPError          PGPGetDefaultPrivateKey(
                       PGPKeySetRef       keySet,
                       PGPKeyRef          *key );
```

Arguments

`keySet`	the target key set
`key`	the receiving field for the associated default private key

Description

Obtains the default private key, which is used for signing, for the *key database associated with* the specified key set.

Errors

```
kPGPError_ItemNotFound
```

Notes, Warnings, and Tips

The current implementation treats the look-up key set as an indirect parameter that references a key database, rather than as an explicit destination.

The indirect nature of this interface is likely to change in a future version, and will almost certainly involve changes to this function’s parameterization.

Key User-Defined Data Functions

The PGPsdk provides the PGPsdk developer with a mechanism by which arbitrary data may be associated with keys and key elements. This data is of type `PGPUserValue`, and can be used for housekeeping, as pointers to data structures, or for any other user-defined purpose. When a key is first imported, all of these values are initialized to zero. These values are not saved with the key - they are only valid while the key or key element is in-memory.

PGPSetKeyUserVal

```
PGPError        PGPSetKeyUserVal(
                    PGPKeyRef           key,
                    PGPUserValue        userValue );
```

Arguments

`key`	the key with which the user value will be associated
`userValue`	the user data

Description

Associates a user-defined value or data structure with the specified key, provided that key is still in memory.

PGPSetSubKeyUserVal

```
PGPError        PGPSetSubKeyUserVal(
                    PGPSubKeyRef        subkey,
                    PGPUserValue        userValue );
```

Arguments

`subkey`	the sub-key with which the user value will be associated
`userValue`	the user data

Description

Associates a user-defined value or data structure with the specified sub-key, provided that sub-key is still in memory.

PGPSetSigUserVal

```
PGPError        PGPSetSigUserVal(
                    PGPSigRef           sig,
                    PGPUserValue        userValue );
```

Arguments

`sig`	the signature with which the user value will be associated
`userValue`	the user data

Description

Associates a user-defined value or data structure with the specified signature, provided that signature is still in memory.

PGPSetUserIDUserVal

```
PGPError          PGPSetUserIDUserVal(
                       PGPUserIDRef        userID,
                       PGPUserValue        userValue );
```

Arguments

`userID`	the user ID with which the user value will be associated
`userValue`	the user data

Description

Associates a user-defined value or data structure with the specified user ID, provided that user ID is still in memory.

PGPGetKeyUserVal

```
PGPError          PGPGetKeyUserVal(
                       PGPKeyRef           key,
                       PGPUserValue        *userValue );
```

Arguments

`key`	the target key
`userValue`	the receiving field for the user data

Description

Obtains the user data associated with the specified key (if any), and places it into the specified field.

Notes, Warnings, and Tips

Any associated user data is always initialized to zeroes upon creation of a PGP data type instance.

PGPGetSubKeyUserVal

```
PGPError          PGPGetSubKeyUserVal(
                       PGPSubKeyRef        subkey,
                       PGPUserValue        *userValue );
```

Arguments

`subkey`	the target sub-key
`userValue`	he receiving field for the user data

Description

Obtains the user data associated with the specified sub-key (if any), and places it into the specified field.

Notes, Warnings, and Tips

Any associated user data is always initialized to zeroes upon creation of a PGP data type instance.

PGPGetSigUserVal

```
PGPError         PGPGetSigUserVal(
                      PGPSigRef           sig,
                      PGPUserValue        *userValue );
```

Arguments

`sig`	the target signature
`userValue`	the receiving field for the user data

Description

Obtains the user data associated with the specified signature (if any), and places it into the specified field.

Notes, Warnings, and Tips

Any associated user data is always initialized to zeroes upon creation of a PGP data type instance.

PGPGetUserIDUserVal

```
PGPError         PGPGetUserIDUserVal(
                      PGPUserIDRef        userID,
                      PGPUserValue        *userValue );
```

Arguments

`userID`	the target user ID
`userValue`	the receiving field for the user data

Description

Obtains the user data associated with the specified User ID (if any), and places it into the specified field.

Notes, Warnings, and Tips

Any associated user data is always initialized to zeroes upon creation of a PGP data type instance.

KeyID Functions

PGPImportKeyID

```
PGPError         PGPImportKeyID(
                      void const          *data,
                    PGPKeyID           *keyID );
```

Arguments

`data`	the key ID data to import
`keyID`	the receiving field for the resultant key ID

Description

Imports the key ID.

Notes, Warnings, and Tips

`data` must be in the format produced by `PGPExportKeyID,` and must reference a buffer of at least `kPGPMaxExportedKeyIDSize` bytes in length

PGPExportKeyID

```
PGPError         PGPExportKeyID(
                    PGPConstKeyIDRef   keyID,
                    PGPByte            exportedData[
                                            kPGPMaxExportedKeyIDSize ],
                    PGPSize            *exportedLength );
```

Arguments

`keyID`	the key ID to be exported
`exportedData`	the receiving field for the exported key ID data
`exportedLength`	the receiving field for the resultant length of the exported key ID data

Description

Exports the specified key ID.

PGPGetKeyIDString

```
PGPError         PGPGetKeyKeyIDString(
                    PGPConstKeyIDRef   keyID,
                    PGPKeyIDStringType type,
                    char               outString[
                                            kPGPMaxKeyIDStringSize ] );
```

Arguments

`keyID`	the target key ID
`type`	the type of key ID string to return, which recognizes `kPGPKeyIDString_`… values (see Table 2-7)
`outString`	the receiving field for the associated key ID string

Description

Retrieves the string associated with the specified key ID.

Errors

`kPGPError_BufferTooSmall`

PGPGetKeyIDFromString

```
PGPError          PGPGetKeyIDFromString(
                       const char          *string,
                       PGPKeyID            *keyID );
```

Arguments

`string` the target string

`keyID` the receiving field for the resultant key ID

Description

Creates a key ID corresponding to the specified key string.

Notes, Warnings, and Tips

The `string` argument length must not exceed `kPGPMaxKeyIDStringSize`.

PGPGetKeyByKeyID

```
PGPError          PGPGetKeyByKeyID(
                       PGPKeySetRef        keySet,
                       PGPKeyID const      *keyID,
                       PGPPublicKeyAlgorithm
                                           pubKeyAlgorithm,
                       PGPKeyRef           *key );
```

Arguments

`keySet` the look-up key set

`keyID` the target keyID

`pubKeyAlgorithm` the public key algorithm used to generate the target keyID, which recognizes `kPGPPublicKeyAlgorithm_`… values (see Table 3-5)

`key` the receiving field for the resultant key

Description

Searches the *key database associated with* the specified key set for the key whose keyID and public key algorithm match those specified. This is especially useful for finding the keys of signing users.

Notes, Warnings, and Tips

Specifying the public key algorithm as `kPGPPublicKeyAlgorithm_Invalid` causes it to be ignored as a selection criteria.

The current implementation treats the look-up key set as an indirect parameter that references a key database, rather than as an explicit destination. Because of key filtering and the “live” nature of its resultant view-style key sets, the resultant key may or may not appear in the specified look-up key set, depending upon its key filtering criteria.

The indirect nature of this interface is likely to change in a future version, and will almost certainly involve changes to this function's parameterization.

Sample Code

```
PGPKeyID                    keyID;
PGPKeyRef                   key;
PGPInt32                    tmpPubKeyAlg;
PGPPublicKeyAlgorithm       pubKeyAlg;

if  ( IsntPGPError( PGPGetKeyIDOfCertifier( sigRef, &keyID ) ) )
{
    PGPGetSigNumber( sigRef, kPGPSigPropAlgID, &tmpPubKeyAlg );
    pubKeyAlg   = ( PGPPublicKeyAlgorithm )tmpPubKeyAlg;
    if  ( IsntPGPError( PGPGetKeyByKeyID(allKeys, &keyID, pubKeyAlg, &key )
    {
          return( key );
    }
}
return( kPGPError_ItemNotFound );
```

PGPGetKeyIDFromKey

```
PGPError         PGPGetKeyIDFromKey(
                      PGPKeyRef           key,
                      PGPKeyID            *keyID );
```

Arguments

`key`	the target key
`keyID`	the receiving field for the resultant key ID

Description

Creates a key ID corresponding to the specified key.

PGPGetKeyIDFromSubKey

```
PGPError         PGPGetKeyIDFromSubKey(
                      PGPKeyRef           subkey,
                      PGPKeyID            *keyID );
```

Arguments

`subkey`	the target sub-key
`keyID`	the receiving field for the resultant key ID

Description

Creates a key ID corresponding to the specified sub-key.

PGPGetKeyIDOfCertifier

```
PGPError         PGPGetKeyIDOfCertifier(
                      PGPSigRef           sig,
                    PGPKeyID          *keyID );
```

Arguments

`sig`	the target signature
`keyID`	the receiving field for the associated KeyID

Description

Retrieves the KeyID of the specified signature.

PGPGetIndexedAdditionalRecipientRequestKeyID

```
PGPError         PGPGetIndexedAdditionalRecipientRequestKeyID(
                      PGPKeyRef           baseKey,
                      PGPUInt32           index,
                      PGPKeyID            *arrKeyID,
                      PGPByte             *arrKeyClass );
```

Arguments

`baseKey`	the target key
`index`	the index (zero-based) of the desired additional recipient request key
`arrKeyID`	the receiving field for the *n*th additional recipient request key ID
`arrKeyClass`	the receiving field for the class of the *n*th additional recipient request key

Description

Provides a means of indexing through the available additional recipient request key, and retrieving each additional recipient request key ID and its class. The resultant key ID, however, may not be used to access the additional recipient request key directly since key ID values are not unique.

Errors

```
kPGPError_ItemNotFound
kPGPError_OutOfRings
```

Notes, Warnings, and Tips

Either of `arrKeyID` and `arrKeyClass` may be `NULL` to indicate that the associated value should not be retrieved, but not both.

The class of the additional recipient request key is currently reserved for internal use, and the caller should not infer anything from its value.

Sample Code

(see the sample code for `PGPGetIndexedAdditionalRecipientRequest`)

PGPCompareKeyIDs

```
PGPInt32    PGPCompareKeyIDs(
                      PGPConstKeyIDRef    keyID1,
                      PGPConstKeyIDRef    keyID2);
```

Arguments

`keyID1`	key ID
`keyID2`	key ID

Description

Compares the key IDs, and returns `–1`, `0`, or `1` depending upon whether `keyID1` is less than `keyID2`, `keyID1` equals `keyID2`, or `keyID1` is greater than `keyID2`.

Key Item Context Retrieval Functions

PGPGetKeySetContext

```
PGPContextRef    PGPGetKeySetContext(
                      PGPKeySetRef       keySet );
```

Arguments

`keySet`	the target keySet

Description

Returns the context associated with the specified key set.

Notes, Warnings, and Tips

If the specified key set is invalid, then the returned context reference value is set to `kInvalidPGPContextRef`.

PGPGetKeyListContext

```
PGPContextRef    PGPGetKeyListContext(
                      PGPKeyListRef      keyList );
```

Arguments

`keyList`	the target key list

Description

Returns the context associated with the specified key list.

Notes, Warnings, and Tips

If the specified key list is invalid, then the returned context reference value is set to `kInvalidPGPContextRef`.

PGPGetKeyIterContext

```
PGPContextRef    PGPGetKeyIterContext(
                      PGPKeyIterRef      keyIter );
```

Arguments

`keyIter`	the target key iterator

Description

Returns the context associated with the specified key iterator.

Notes, Warnings, and Tips

If the specified key iterator is invalid, then the returned context reference value is set to `kInvalidPGPContextRef`.

PGPGetKeyContext

```
PGPContextRef    PGPGetKeyContext(
                      PGPKeyRef           key );
```

Arguments

`key`	the target key

Description

Returns the context associated with the specified key.

Notes, Warnings, and Tips

If the specified key is invalid, then the returned context reference value is set to `kInvalidPGPContextRef`.

PGPGetSubKeyContext

```
PGPContextRef    PGPGetSubKeyContext(
                      PGPSubKeyRef        subKey );
```

Arguments

`subKey`	the target sub-key

Description

Returns the context associated with the specified sub-key.

Notes, Warnings, and Tips

If the specified sub-key is invalid, then the returned context reference value is set to `kInvalidPGPContextRef`.

PGPGetUserIDContext

```
PGPContextRef    PGPGetUserIDContext(
                      PGPUserIDRef        userID );
```

Arguments

`userID`	the target user ID

Description

Returns the context associated with the specified user ID.

Notes, Warnings, and Tips

If the specified user ID is invalid, then the returned context reference value is set to `kInvalidPGPContextRef`.

Chapter 3: Function Reference - Ciphering and Authentication Functions

Introduction

The PGPsdk provides high-level, algorithm-independent cryptographic functions for encrypting, decrypting, hashing, signing, and verifying messages and data. These not only free applications from having to be aware of the particular algorithm being used, but also allow new algorithms to be supported as they become available. Function prototypes are listed in the public header file `pgpEncode.h`. In most cases, inputs and outputs can be specified as any arbitrary combination of memory buffers and/or data files.

The PGPsdk also provides low-level cryptographic functions for developers who have special requirements, or require greater control over ciphering and authentication activities. Function prototypes are listed in the public header files `pgpCBC.h`, `pgpCFB.h`, `pgpHash.h`, `pgpPublicKey.h`, and `pgpSymmetricCipher.h`. `pgpCFB.h`, `pgpHash.h`, and `pgpSymmetricCipher.h` also appear as `#include` directives in `pgpEncode.h`, since the high-level functions are based on **cipher feedback mode** methodology.

Certain PGPsdk functions – most notably decryption and key generation (see Chapter 2) – require a significant amount of time to complete. To facilitate control and progress tracking, these functions support an event and callback mechanism. This same mechanism also provides for prompting of required information when required for example, file specifications, passphrases.

Header Files

```
pgpEncode.h
pgpCBC.h
pgpCFB.h
pgpHash.h
pgpPublicKey.h
pgpSymmetricCipher.h
```

Constants and Data Structures

Table 3-1: Event Type Values.

Event Type Constant	Event Description
`kPGPEvent_NullEvent`	Progress notification
`kPGPEvent_InitialEvent`	Initial event
`kPGPEvent_FinalEvent`	Final event
`kPGPEvent_ErrorEvent`	An error occurred
`kPGPEvent_WarningEvent`	Warning event
`kPGPEvent_EntropyEvent`	More entropy is needed
`kPGPEvent_PassphraseEvent`	A passphrase is needed
`kPGPEvent_AnalyzeEvent`	Initial analysis event, before any output
`kPGPEvent_RecipientsEvent`	Recipient list report, before any output
`kPGPEvent_KeyFoundEvent`	Key packet found
`kPGPEvent_OutputEvent`	Output specification needed

kPGPEvent_SignatureEvent	Signature status report
kPGPEvent_BeginLexEvent	Initial event per lexical unit
kPGPEvent_EndLexEvent	Final event per lexical unit
kPGPEvent_DetachedSignatureEvent	Need input for verification of detached signature
kPGPEvent_KeyGenEvent	Key generation progress
kPGPEvent_KeyServerEvent	Key server progress
kPGPEvent_KeyServerSignEvent	Key server passphrase

Table 3-2: Lexical Section Type Values.

Section Type Constant	Section Type Description
kPGPAnalyze_DetachedSignature	Detached signature
kPGPAnalyze_Encrypted	Encrypted message
kPGPAnalyze_Key	Key data
kPGPAnalyze_Signed	Signed message
kPGPAnalyze_Unknown	Non-PGP message

Table 3-3: Hash Algorithm Selection Values.

Hash Algorithm Constant
kPGPHashAlgorithm_Invalid
kPGPHashAlgorithm_MD5
kPGPHashAlgorithm_RIPEMD160
kPGPHashAlgorithm_SHA
kPGPHashAlgorithm_SHADouble

Table 3-4a: Symmetric Cipher Algorithm Selection Values.

Symmetric Cipher Algorithm Constant
kPGPCipherAlgorithm_None
kPGPCipherAlgorithm_CAST5
kPGPCipherAlgorithm_IDEA
kPGPCipherAlgorithm_3DES

Table 3-4b: Symmetric Cipher Algorithm Key Size Values.

Symmetric Cipher Algorithm	Key Size (Bits)	Key Size (Bytes)
kPGPCipherAlgorithm_CAST5	128***	16
kPGPCipherAlgorithm_IDEA	128	16
kPGPCipherAlgorithm_3DES	192	24

***The PGPsdk does *not* support the **CAST** 40- and 80-bit key sizes, as these are considered to yield **cipher text** of less than acceptable cryptographic strength.

Table 3-5: Public Key Algorithm Selection Values.

Public Key Algorithm Constant	
kPGPPublicKeyAlgorithm_Invalid	
kPGPPublicKeyAlgorithm_ElGamal	(a.k.a.Diffie-Hellman)
kPGPPublicKeyAlgorithm_DSA	
kPGPPublicKeyAlgorithm_RSA	
kPGPPublicKeyAlgorithm_RSAEncryptOnly	
kPGPPublicKeyAlgorithm_RSASignOnly	

Table 3-6: Public Key Message Format Values

Public Key Message Format Constant
kPGPPublicKeyMessageFormat_PGP
kPGPPublicKeyMessageFormat_PKCS1

Table 3-7a: Key Trust Values.

Trust Value Constant
`kPGPKeyTrust_Mask`
`kPGPKeyTrust_Undefined`
`kPGPKeyTrust_Unknown`
`kPGPKeyTrust_Never`
`kPGPKeyTrust_Marginal`
`kPGPKeyTrust_Complete`
`kPGPKeyTrust_Ultimate`

Table 3-7b: Name Trust Values.

Trust Value Constant
`kPGPNameTrust_Mask`
`kPGPNameTrust_Unknown`
`kPGPNameTrust_Untrusted`
`kPGPNameTrust_Marginal`
`kPGPNameTrust_Complete`

Table 3-8: Validity Level Values.

Validity Level Constant
`kPGPValidity_Unknown`
`kPGPValidity_Invalid`
`kPGPValidity_Marginal`
`kPGPValidity_Complete`

Table 3-9: Line Ending Option Values.

Line Ending Option Constant
`kPGPLineEnd_CR`
`kPGPLineEnd_CRLF`
`kPGPLineEnd_Default`
`kPGPLineEnd_LF`

Table 3-10: Local Encoding Option Values.

Local Encoding Constant
`kPGPLocalEncoding_Auto***`
`kPGPLocalEncoding_Force***`
`kPGPLocalEncoding_NoMacBinCRCOkay`
`kPGPLocalEncoding_None`

`***kPGPLocalEncoding_Force` and `kPGPLocalEncoding_Auto` are mutually exclusive.

Table 3-11: Valid `PGPOptionListRef` Options for Ciphering and Authentication Functions.

Option Function	PGPEncode	PGPDecode	PGPExportKeySet	PGPImportKeySet
PGPOAdditionalRecipientRequestKeySet				
PGPOAllocatedOutputBuffer	●	●	●	
PGPOAppendOutput	●	●		
PGPOArmorOutput	●			
PGPOAskUserForEntropy	●			
PGPOCipherAlgorithm	●			
PGPOClearSign	●			
PGPOCommentString	●		●	
PGPOCompression	●			
PGPOConventionalEncrypt	●			
PGPODataIsASCII	●			

PGPODetachedSig	●	●		
PGPODiscardOutput	●	●	●	
PGPOEncryptToKey	●			
PGPOEncryptToKeySet	●			
PGPOEncryptToUserID	●			
PGPOEventHandler	●	●	●	●
PGPOExpiration				
PGPOExportable				
PGPOExportPrivateKeys			●	
PGPOFailBelowValidity	●	●		
PGPOForYourEyesOnly	●			
PGPOHashAlgorithm	●			
PGPOImportKeysTo		●		
PGPOInputBuffer	●	●		●
PGPOInputFile	●	●		●
PGPOInputFileFSSpec	●	●		●
PGPOKeyGenFast				
PGPOKeyGenMasterKey				
PGPOKeyGenName				
PGPOKeyGenParams				
PGPOKeySetRef		●		
PGPOLastOption	●	●	●	●
PGPOLocalEncoding	●	●		●
PGPONullOption	●	●	●	●
PGPOOmitMIMEVersion	●			
PGPOOutputBuffer	●	●	●	
PGPOOutputFile	●	●	●	
PGPOOutputFileFSSpec	●	●	●	
PGPOOutputLineEndType	●	●		
PGPOPassphrase	●	●		
PGPOPassphraseBuffer	●	●		
PGPOPassThroughIfUnrecognized		●		
PGPOPGPMIMEEncoding	●			
PGPOPreferredAlgorithms	●			
PGPORawPGPInput	●			
PGPOSendEventIfKeyFound		●		
PGPOSendNullEvents	●	●	●	●
PGPOSignWithKey	●			
PGPOSigRegularExpression				
PGPOSigTrust				
PGPOVersionString	●		●	
PGPOWarnBelowValidity	●	●		

Events and Callbacks

The `PGPOEventHandler` option allows the calling application to request callbacks when various events occur, and to define the function (event handler) that is the target of the callback. While an event handler is usually not needed for encryption operations, it is often needed for decryption operations.

An event handler serves two purposes – it provides notification to the calling application that an event has occurred, and provides a mechanism for the calling application to affect processing (in a pre-defined manner). Notification includes a pointer to a `PGPEvent` data type that, depending on the type of event, provides detailed information about the cause of the event. The calling application can then respond appropriately, which may or may not intervene and affect the course of further processing. If the calling application wishes to intervene, then it can abort the job by returning an error code (a value other than `kPGPError_NoErr`, except in the cases of `kPGPEvent_ErrorEvent and kPGPEvent_AnalyzeEvent`). Additionally, depending on the type of event, it can modify the processing context by invoking `PGPAddJobOptions`.

All event handlers are declared as

```
    PGPError myEvents( PGPContextRef pgpContext,
                    PGPEvent *event,
                    PGPUserValue userValue );
```

The pgpContext argument is the reference to the context of the job posting the event. The event argument references a PGPEvent data type as follows:

```
struct PGPEvent_
{
    PGPVersion              version;
    struct PGPEvent_        *nextEvent;
    PGPJobRef               job;
    PGPEventType            type;
    PGPEventData            data;
};
typedef struct PGPEvent_ PGPEvent;
```

The version and nextEvent members are currently reserved for internal use. The job member references the currently active encode or decode activity. The type member identifies the event being posted, and recognizes kPGPEvent_… values (see Table 3-1). The data member is a union of the event-specific data structures, which are described with their corresponding event (some events have no associated event-specific data).

The calling application can modify the processing context by invoking PGPAddJobOptions as

```
PGPError   PGPAddJobOptions( PGPJobRef job, ... );
```

The value of the job argument is that of the PGPEvent argument's job member. Additional PGPOptionListRef arguments are specified similarly to the way they are passed to PGPEncode and PGPDecode. However, only certain options can be set after each type of event, and these are listed for each event.

Figure 3-1: Encode Processing Event Sequence.

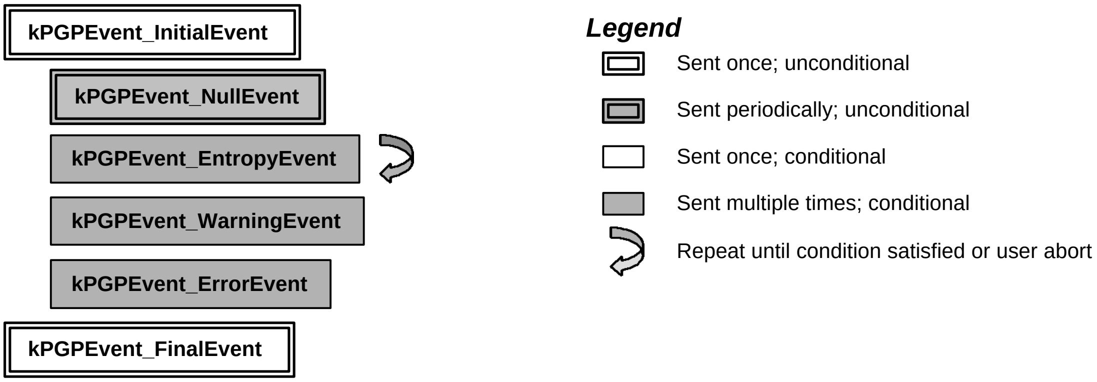
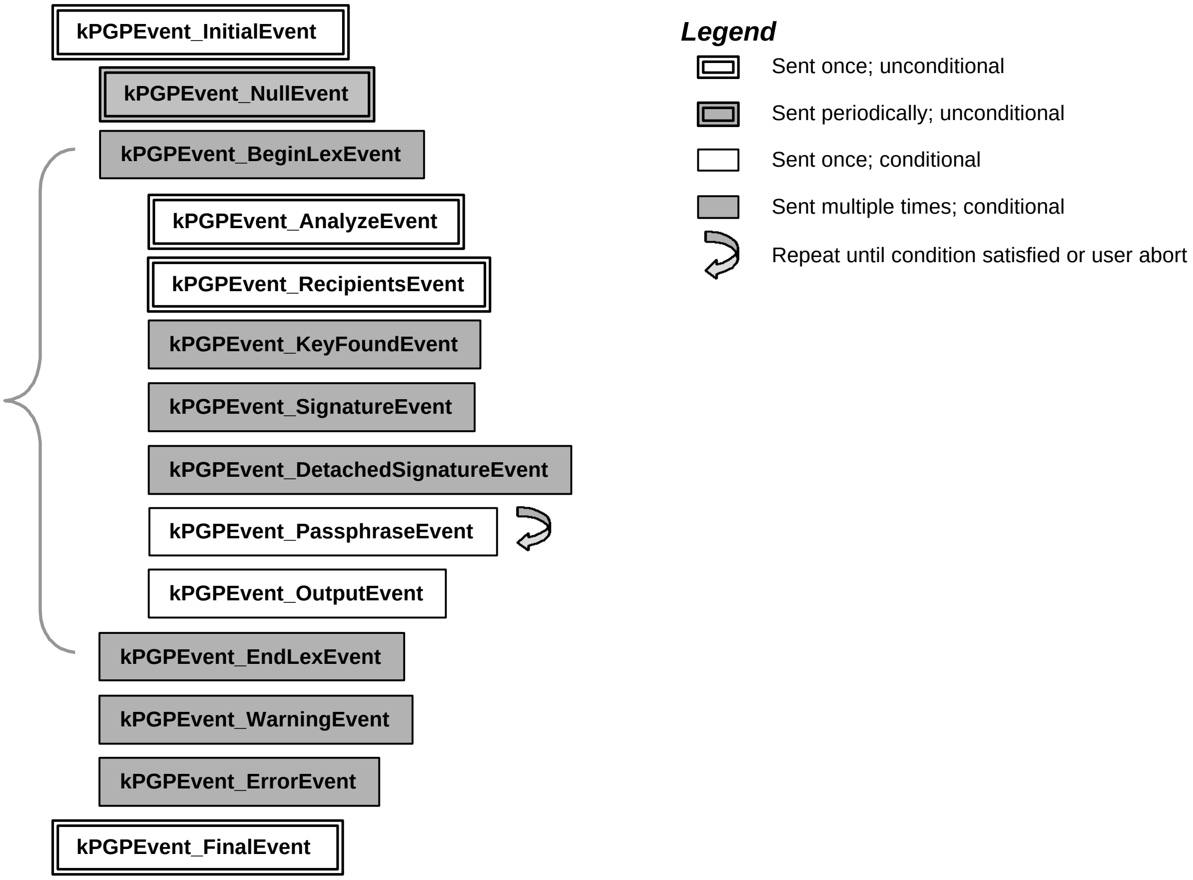


Figure 3-2: Decode Processing Event Sequence.



Common Cipher Events

kPGPEvent_InitialEvent

Sent before all other events. Implies initiation of the job.

Data

None

Options

None

kPGPEvent_NullEvent

Sent during the course of encode/decode processing if explicitly requested with `PGPOSendNullEvents` (see `PGPEncode` and `PGPDecode`).

The event data allows the PGPsdk developer to determine the sending function’s progress and completion percentage. Its members should be treated as relative, unscaled quantities – they are not necessarily byte quantities. In all cases, the completion percentage is calculated as follows:

```
double    completionPercent;

if  ( event->type = kPGPEvent_NullEvent )
{
    if  ( event->nullData.bytesTotal != 0 )
    {
          completionPercent = ( 100 * event->nullData.bytesWritten ) /
                                   event->nullData.bytesTotal;
    }
    else
    {
          completionPercent = 100;
    }
}
```

Progress tracking that involves compressed input files is rarely linear, since it tracks access of the compressed data, and not the decompression and processing of the resultant expanded data.

Data

```
typedef struct PGPEventNullData_
{
    PGPFileOffset     bytesWritten;
    PGPFileOffset     bytesTotal;
} PGPEventNullData;
```

Options

None

kPGPEvent_WarningEvent

Sent whenever a non-fatal error occurs during processing. The associated event data always includes the error code, and for certain warnings includes an error-specific argument. Unlike `kPGPEvent_ErrorEvent,` the value returned by the event handler is not ignored, and so a value other than `kPGPError_NoErr` will abort the job.

Data

```
typedef struct PGPEventWarningData_
{
    PGPError          warning;
    void              *warningArg;
} PGPEventWarningData;
```

Options

None

kPGPEvent_ErrorEvent

Sent whenever a fatal error occurs during processing. The associated event data always includes the error code, and for certain errors includes an error-specific argument. Upon return from the event handler, the job will always abort and return the initial error code – the value returned by the event handler is ignored.

Data

```
typedef struct PGPEventErrorData_
{
    PGPError          error;
    void              *errorArg;
} PGPEventErrorData;
```

Options

None

kPGPEvent_FinalEvent

Sent after all other events. Implies termination of the job.

Data

None

Options

None

PGPEncode-only Events

kPGPEvent_EntropyEvent

Sent if more entropy is needed for signing or encrypting, and indicates the minimum number of entropy bits that the event handler should add to the random pool (see Chapter 5 for descriptions of the available random number pool management functions). For example:

```
while ( !PGPGlobalRandomPoolHasMinimumEntropy( void ) )
{
    PGPGlobalRandomPoolAddKeystroke( myGetKeystrokeFunction( void ) );
}
```

Data

```
typedef struct PGPEventEntropyData_
{
    PGPUInt32         entropyBitsNeeded;
} PGPEventEntropyData;
```

Options

None

PGPDecode-only Events

kPGPEvent_BeginLexEvent

Sent whenever a new **lexical section** is encountered in the input. A lexical section is a block of data delimited by `---BEGIN PGP` and `---END PGP` (ASCII input; binary input has only one section). The zero-based `sectionNumber` value indicates which section has been encountered.

Data

```
typedef struct PGPEventBeginLexData_
{
    PGPUInt32         sectionNumber;
    PGPSize           sectionOffset;
} PGPEventBeginLexData;
```

Options

None

kPGPEvent_AnalyzeEvent

Sent immediately after a `BeginLexEvent` to identify the type of the current lexical section. This allows the event handler to decide if it should skip this lexical section, but not abort the whole job, by returning the special error value `kPGPError_SkipSection`.

`sectionType` recognizes `kPGPAnalyze_`… values (see Table 3-2).

Data

```
typedef struct PGPEventAnalyzeData_
{
    PGPAnalyzeType          sectionType;
} PGPEventAnalyzeData;
```

Options

None

kPGPEvent_RecipientsEvent

Sent immediately after an `AnalyzeEvent` to describe the recipient(s) of the message. Generally, there can be three types of recipients:

- keys that are on the active key ring
- keys that are *not* on the active key ring
- conventional encryption passphrases

Determination of which keys are present is based upon a search of the key set specified in the `PGPOKeySetRef` option passed to `PGPDecode`. Generally, this key set will have resulted from opening the default key ring (see `PGPOpenDefaultKeyRings`, `PGPOpenKeyRing`, and `PGPOpenKeyRingPair`).

`recipientSet` identifies the set of keys required to decrypt the message, and which are currently available. `conventionalPassphraseCount` indicates how many different passphrases the message is encrypted to (typically zero or one). `keyCount` indicates the number of keys required to decrypt the message that are not currently available, and these are identified by `keyID` in the referenced `keyIDArray`.

Data

```
typedef struct PGPEventRecipientsData_
{
    PGPKeySetRef      recipientSet;
    PGPUInt32         conventionalPassphraseCount;
    PGPUInt32         keyCount;
    PGPKeyID const    *keyIDArray;
} PGPEventRecipientsData;
```

Options

None

kPGPEvent_KeyFoundEvent

Sent whenever all of the following are `TRUE`:

- a key is found in the input data
- the `PGPOImportKeysTo` option was *not* specified, telling the job where to put the key
- the `PGPOSendEventIfKeyFound` option was specified

`keySet` holds the key found in the input data, and this key set is automatically freed upon return. The event handler code can process the key in anyway it sees fit, but will usually choose to merge the key into some key set (see `PGPAddKeys`).

Data

```
typedef struct PGPEventKeyFoundData_
{
    PGPKeySetRef                  keySet;
} PGPEventKeyFoundData;
```

Options

`PGPOImportKeysTo`
`PGPOSendEventIfKeyFound(…,FALSE)`

kPGPEvent_SignatureEvent

Sent for signed messages to provide information about the signature status.

`signingKeyID` always contains the key ID of the signing key. `signingKey` contains the signing key itself if it is on the local key ring.

The key validity flags increase monotonically, that is, if one is `TRUE`, then the flags preceding it must also be `TRUE`:

- `checked` indicates that the key is available, and that the message is properly formatted
- `verified` indicates that the signature validated correctly
- `keyRevoked`, `keyDisabled`, and `keyExpired` indicate that the signing key is no longer active
- `keyValidity` indicates the validity level of the signing key

The `keyValidity` flag is set based on the signing key's validity in relation to the thresholds set by the `PGPDecode` options `PGPOWarnBelowValidity` and `PGPOFailBelowValidity`.

`creationTime` indicates when the key was signed.

Data

```
typedef struct PGPEventSignatureData_
{
    PGPKeyID          signingKeyID;
    PGPKeyRef         signingKey;
    PGPBoolean        checked;
    PGPBoolean        verified;
    PGPBoolean        keyRevoked;
    PGPBoolean        keyDisabled;
    PGPBoolean        keyExpired;
    PGPBoolean        keyMeetsValidityThreshold;
    PGPValidity       keyValidity;
    PGPTime           creationTime;
} PGPEventSignatureData;
```

Options

PGPOWarnBelowValidity
PGPOFailBelowValidity

kPGPEvent_DetachedSigEvent

Sent to notify the event handler that the input file contains a detached signature (a signature that is not attached to the file it signs). The event handler must provide an input source to be signature-checked against the detached signature. This can be any of the forms of input described among the options. The event handler should set a `PGPODetachedSig` option with the input data to be checked as a sub-option.

Data

None

Options

`PGPODetachedSig` with a sub-option of one of:

- `PGPOInputFile`
- `PGPOInputFSSpec` (MacOS platforms only)
- `PGPOInputBuffer`

kPGPEvent_PassphraseEvent

Sent if a passphrase is needed for decrypting (posted by `PGPDecode`), either to unlock a decryption key or to decrypt a conventionally encrypted message. The event handler should invoke `PGPAddJobOptions` specifying the `PGPOPassphrase` or `PGPOPassphraseBuffer` option, or return `kPGPError_UserAbort` if no passphrase is available.

If a passphrase is needed for a conventionally encrypted message, then the `fConventional` flag is `TRUE`, and `keyset` is ignored. Otherwise, `keyset` includes the key(s) for which a passphrase is needed.

If a passphrase is needed for decryption, then `keyset` will hold multiple keys if the message can be decrypted by multiple secret keys on the key ring. However, any passphrase which unlocks any of these secret keys is acceptable as a response.

This event is sent repeatedly until a valid passphrase is received, or until the event handler requests abort of the job. This allows the event handler to enforce a limit on the number of passphrase attempts.

Data

```
typedef struct PGPEventPassphraseData_
{
    PGPBoolean         fConventional;
    PGPKeySetRef       keyset;
} PGPEventPassphraseData;
```

Options

PGPOPassphrase
PGPOPassphraseBuffer

kPGPEvent_OutputEvent

If the initial call to PGPDecode did not include an output specification option, then this event will be sent whenever a new section of the message is encountered. This allows the application total flexibility in routing each output section.

If the initial call to PGPDecode did include an output specification option, then this event will not be sent and all output will go to the specified location. However, keys are handled as described in kPGPEvent_KeyFoundEvent.

The messageType indicates whether the section is text, data, or non-PGP. The suggestedName argument specifies the name the encrypted or signed file had when it was encrypted. The forYourEyesOnly flag is TRUE if the encryption specified the PGPOForYourEyesOnly option.

The event handler should use this information to specify a processing option appropriate for the output of the section. These options include:

- write the output to a file
- write the output to a buffer
- discard the output

The event handler should return an error if it cannot set an output option.

Data

```
typedef struct PGPEventOutputData_
{
    PGPUInt32          messageType;
    char               *suggestedName;
    PGPBoolean         forYourEyesOnly;
} PGPEventOutputData;
```

Options

Write the output to a file:

- PGPOOutputFile
- PGPOOutputFSSpec (MacOS platforms only)
- PGPOAppendOutput

Write the output to a buffer:

- PGPOAllocatedOutputBuffer
- PGPOOutputBuffer
- PGPOAppendOutput

Discard the output:

- PGPODiscardOutput

kPGPEvent_EndLexEvent

Sent whenever a lexical section is completed (see the `BeginLexEvent` description for how sections are defined). The zero-based `sectionNumber` value indicates which section has been completed.

Data

```
typedef struct PGPEventEndLexData_
{
    PGPUInt32         sectionNumber;
} PGPEventEndLexData;
```

Options

None

Encode and Decode Functions

PGPEncode

```
PGPError         PGPEncode(
                      PGPContextRef      pgpContext,
                      PGPOptionListRef   firstOption,
                      ...,
                      PGPOLastOption( void ) );
```

Arguments

`pgpContext`	the target context
`firstOption`	the initial option list instance
`...`	subsequent option list instances
`PGPOLastOption( void )`	must always appear as the final argument to terminate the argument list

Description

Encrypts a block of text according to the target context and specified options.

Errors

```
kPGPError_RedundantOptions
kPGPError_KeyInvalid
kPGPError_KeyExpired
kPGPError_KeyDisabled
kPGPError_KeyRevoked
kPGPError_KeyUnusableForEncryption
kPGPError_KeyUnusableForSignature
kPGPError_MissingPassphrase
kPGPError_InconsistentEncryptionAlgorithms
kPGPError_CombinedConventionalAndPublicEncryption
kPGPError_NoInputOptions
kPGPError_MultipleInputOptions
kPGPError_InputFile
kPGPError_NoOutputOptions
```

```
kPGPError_MultipleOutputOptions
kPGPError_OutputBufferTooSmall
kPGPError_MissingEventHandler
kPGPError_MissingKeySet
kPGPError_TooManyARRKs
```

PGPDecode

```
PGPError        PGPDecode(
                     PGPContextRef      pgpContext,
                     PGPOptionListRef   firstOption,
                     ...,
                     PGPOLastOption( void ) );
```

Arguments

`pgpContext` the target context

`firstOption` the initial option list instance

`...` subsequent option list instances

`PGPOLastOption( void )`
must always appear as the final argument to terminate the argument list

Description

Decrypts a block of text according to the target context and specified options.

Errors

```
kPGPError_RedundantOptions
kPGPError_MissingPassphrase
kPGPError_DetachedSignatureFound
kPGPError_DetachedSignatureWithoutSigningKey
kPGPError_DetachedSignatureWithEncryption
kPGPError_NoInputOptions
kPGPError_MultipleInputOptions
kPGPError_InputFile
kPGPError_NoOutputOptions
kPGPError_MultipleOutputOptions
kPGPError_OutputBufferTooSmall
kPGPError_MissingEventHandler
kPGPError_MissingKeySet
kPGPError_NoDecryptionKeyFound
kPGPError_SkipSection
```

Low-Level Cipher Functions

PGPNewHashContext

```
PGPError        PGPNewHashContext(
                     PGPContextRef      pgpContext,
                     PGPHashAlgorithm   algID,
                     PGPHashContextRef  *hashContext );
```

Arguments

`pgpContext`	the target context
`algID`	the hash algorithm to use, which recognizes `kPGPHashAlgorithm_…` values (see Table 3-3)
`hashContext`	the receiving field for the resultant hash context

Description

Creates a new hash context that utilizes the specified algorithm.

Errors

`kPGPError_AlgorithmNotAvailable`

PGPCopyHashContext

```
PGPError          PGPCopyHashContext(
                       PGPHashContextRef  hashContextOrig,
                       PGPHashContextRef  *hashContextCopy );
```

Arguments

`hashContextOrig`	the source hash context
`hashContextCopy`	the receiving field for the copy of the hash context

Description

Creates an exact copy of the source hash context.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant hash context copy with `PGPFreeHashContext`.

PGPFreeHashContext

```
PGPError          PGPFreeHashContext(
                       PGPHashContextRef  hashContext );
```

Arguments

`hashContext`	the target hash context

Description

Frees the specified hash context.

Notes, Warnings, and Tips

Hash contexts do *not* have associated reference counts – the context is always deallocated.

PGPGetHashSize

```
PGPError         PGPGetHashSize(
                      PGPHashContextRef  hashContext,
                      PGPSize            *hashSize );
```

Arguments

`hashContext`	the target hash context
`hashSize`	the receiving field for the hash size

Description

Determines the resultant size of the associated hash in bytes, for example, a 160-bit hash may yield 20 bytes of resultant data.

Notes, Warnings, and Tips

Used for generic code that may not know the size of the hash being produced.

PGPContinueHash

```
PGPError         PGPContinueHash(
                      PGPHashContextRef  hashContext,
                      const void         *hashIn,
                      PGPSize            numBytes );
```

Arguments

`hashContext`	the target hash context
`hashIn`	the current hash data
`numBytes`	the length of the current hash data

Description

Continues the hash, accumulating an intermediate result.

Notes, Warnings, and Tips

Normally, `numBytes` should be passed as the value received from `PGPGetHashSize`**.**

Sample Code

```
const void      *hashIn;
PGPSize         numBytes;

PGPGetHashSize( hashContext,
               &numBytes );
PGPContinueHash( hashContext,
                hashIn,
                numBytes );
```

PGPFinalizeHash

```
PGPError         PGPFinalizeHash(
                      PGPHashContextRef  hashContext,
                      void               *hashOut );
```

Arguments

`hashContext`	the target hash context
`hashOut`	the receiving buffer for the resultant hash data

Description

Finalizes the hash, placing the result into `hashOut`. The hash context is then automatically reset via `PGPResetHash`.

Notes, Warnings, and Tips

Use `PGPGetHashSize` to ensure that the result buffer is of adequate size.

To obtain an intermediate result, use `PGPCopyHashContext` and then finalize the copy.

Sample Code

```
PGPError        err;
PGPSize         hashSize;
void            *hashOut;

if ( IsntPGPError( ( err = PGPGetHashSize( hashContext, &hashSize ) ) ) )
{
    hashOut = ( void * )malloc( hashSize );
    if ( hashOut != ( void * )NULL )
    {
          err = PGPFinalizeHash( hashContext, hashOut );
    }
}
return( err );
```

PGPResetHash

```
PGPError         PGPResetHash(
                      PGPHashContextRef  hashContext );
```

Arguments

`hashContext`	the target hash context

Description

Resets a hash context as if it had been created anew. Any existing intermediate hash is lost.

PGPNewSymmetricCipherContext

```
PGPError         PGPNewSymmetricCipherContext(
                      PGPContextRef      pgpContext,
                      PGPCipherAlgorithm algID,
                      PGPSize            keySize,
                      PGPSymmetricCipherContextRef
                                         *cipherContext );
```

Arguments

`pgpContext`	the target context
`algID`	the desired symmetric cipher algorithm, which recognizes `kPGPCipherAlgorithm_`… values (see Table 3-4a)
`keySize`	the desired key size (in bytes; see Table 3-4b)
`cipherContext`	the receiving field for the resultant symmetric cipher context

Description

Creates a new symmetric cipher based upon the specified algorithm.

Errors

`kPGPError_AlgorithmNotAvailable`

Notes, Warnings, and Tips

Currently, all supported symmetric cipher algorithms have only one key size. Specifying the key size as `kPGPSymmetricCipherDefaultKeySize` will not only simplify coding, but also avoid errors. This is especially true if the PGPsdk developer avoids any specification of key size, and instead always obtains the effective key size from `PGPGetSymmetricCipherSizes`.

The resultant symmetric cipher context cannot be used until it has been initialized with `PGPInitSymmetricCipher`.

The caller is responsible for deallocating the resultant symmetric cipher context with `PGPFreeSymmetricCipherContext` *unless* the copy is passed to a function which assumes ownership, for example `PGPNewCBCipherContext` or `PGPNewCFBCipherContext`.

PGPInitSymmetricCipher

```
PGPError         PGPInitSymmetricCipher(
                     PGPSymmetricCipherContextRef
                                         cipherContext,
                     const void          *key );
```

Arguments

`cipherContext`	the target symmetric cipher context
`key`	the desired key

Description

Establishes the key for the symmetric cipher context.

Notes, Warnings, and Tips

The key size is determined by the choice of symmetric cipher, and may be obtained with `PGPGetSymmetricCipherSizes`.

Since the key is copied into the symmetric cipher context and so is no longer needed, the caller is strongly encouraged to clear the key's memory upon successful return.

A symmetric cipher can be repeatedly reset and reused with different keys, which avoids having to create and destroy new contexts each time.

PGPCopySymmetricCipherContext

```
PGPError    PGPCopySymmetricCipherContext(
                 PGPSymmetricCipherContextRef
                                                  cipherContextOrig,
                 PGPSymmetricCipherContextRef
                                                  *cipherContextCopy );
```

Arguments

`cipherContextOrig`	the source symmetric cipher context
`cipherContextCopy`	the receiving field for the copy of the symmetric cipher context

Description

Creates an exact copy of the source symmetric cipher context, including its key.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant symmetric cipher context copy with `PGPFreeSymmetricCipherContext` *unless* the copy is passed to a function which assumes ownership, for example `PGPNewCBCipherContext` or `PGPNewCFBCipherContext`.

PGPFreeSymmetricCipherContext

```
PGPError         PGPFreeSymmetricCipherContext(
                      PGPSymmetricCipherContextRef
                                                  cipherContext );
```

Arguments

`cipherContext`	the target symmetric cipher context

Description

Frees the specified symmetric cipher context.

Notes, Warnings, and Tips

This function should only be called for those symmetric cipher contexts that are *not* passed to functions which assumes ownership, for example `PGPNewCBCipherContext` or `PGPNewCFBCipherContext`.

Symmetric cipher contexts do *not* have associated reference counts – the context is always deallocated.

Before deallocating the context, the function erases all sensitive in-memory data.

PGPGetSymmetricCipherSizes

```
PGPError          PGPGetSymmetricCipherSizes(
                       PGPSymmetricCipherContextRef
                                         cipherContext,
                       PGPSize           *keySize,
                       PGPSize           *blockSize );
```

Arguments

`cipherContext`	the target symmetric cipher context
`keySize`	the receiving field for the associated cipher's key size (in bytes)
`blockSize`	the receiving field for the associated cipher's block size (in bytes)

Description

Returns the key and block sizes (in bytes) for the associated symmetric cipher.

PGPSymmetricCipherEncrypt

```
PGPError          PGPSymmetricCipherEncrypt(
                       PGPSymmetricCipherContextRef
                                         cipherContext,
                       const void        *plainText,
                       void              *cipherText );
```

Arguments

`cipherContext`	the target symmetric cipher context
`plainText`	the source buffer for the input **plain text**
`cipherText`	the receiving buffer for the output cipher text

Description

Encrypts one block of data, whose size is determined by the cipher (see `PGPGetSymmetricCipherBlockSize`).

Notes, Warnings, and Tips

This function should not be used to encrypt multiple blocks of data unless the key is changed for each block (usually through a chaining or feedback scheme), since it is considered bad cryptographic practice to reuse a key in a block cipher.

PGPSymmetricCipherDecrypt

```
PGPError          PGPSymmetricCipherDecrypt(
                       PGPSymmetricCipherContextRef
                                         cipherContext,
                       const void        *cipherText,
                       void              *plainText );
```

Arguments

`cipherContext`	the target symmetric cipher context
`cipherText`	the source buffer for the input cipher text
`plainText`	the receiving buffer for the output plain text

Description

Decrypts one block of data, whose size is determined by the target cipher context (see `PGPGetSymmetricCipherBlockSize`).

PGPWashSymmetricCipher

```
PGPError         PGPWashSymmetricCipher(
                      PGPSymmetricCipherContextRef
                                           cipherContext,
                      void const           *washData,
                      PGPSize              washLength );
```

Arguments

`cipherContext`	the target symmetric cipher context
`washData`	the wash data
`washLength`	the length of the wash data

Description

Hashes the current key of the specified symmetric cipher with the specified wash data to produce a new key.

PGPWipeSymmetricCipher

```
PGPError         PGPWipeSymmetricCipher(
                      PGPSymmetricCipherContextRef
                                           cipherContext );
```

Arguments

`cipherContext`	the target symmetric cipher context

Description

Wipes any sensitive data in the cipher. The cipher context remains “alive”, but its key must be reset before any data can be encrypted.

PGPNewCBCContext

```
PGPError         PGPNewCBCContext(
                      PGPSymmetricCipherContextRef
                                           cipherContext,
                      PGPCBCContextRef     *chainingContext );
```

Arguments

`cipherContext`	the underlying symmetric cipher context
`chainingContext`	the receiving field for the resultant CBC context

Description

Creates a **cipher block chaining** context based upon the specified symmetric cipher.

Notes, Warnings, and Tips

A cipher block chaining context requires use of a symmetric cipher that has been created and whose key has been set. This key may be set explicitly with `PGPInitSymmetricCipher`, or set implicitly with `PGPInitCBC`.

Upon creation of the context, the `CBCRef` "owns" the symmetric `cipherContext` and will dispose of it properly (even if an error occurs). The caller should no longer reference it.

Sample Code

```
PGPError                    err;
PGPSymmetricCipherContextRef
                            cipherContext;
PGPSize                     keySize;
PGPSize                     blockSize;
PGPBoolean                  didLock;
PGPCFBContextRef            chainingContext;
void                        *key;
void                        *initVector;

if  ( IsPGPError( err = PGPNewSymmetricCipherContext( pgpContext,
                                                     kPGPCipherAlgorithm_CAST5,
                                                     kPGPSymmetricCipherDefaultKeySize,
                                                     &cipherContext ) ) )
{
    return( err );
}

if  ( IsPGPError( err = PGPGetSymmetricCipherSizes( cipherContext,
                                                   &keySize,
                                                   &blockSize ) ) )
{
    return( err );
}

/*
** Allocate “sensitive data” memory for the key buffer
*/
if  ( ( key = PGPNewSecureData( pgpContext,
                               keySize,
                               &didLock ) ) == (void *)NULL )
{
    return( kPGPError_OutofMemory );
}

myKeySetFunction( key,
                   keySize );
err = PGPInitSymmetricCipher( cipherContext,
                              ( const void * )key );
PGPFreeData( key );
if  ( IsPGPError( err ) )
{
    return( err );
}

if  ( IsPGPError( err = PGPNewCBCContext( cipherContext,
                                          &chainingContext ) ) )
{
    return( err );
}

if  ( ( initVector = PGPNewSecureData( pgpContext,
                                        blockSize,
                                        &didLock ) ) == (void *)NULL )
{
    return( kPGPError_OutofMemory );
}

myInitVectorSetFunction( initVector,
                         blockSize );
err = PGPInitCFB( cipherContext,
                  (const void *)NULL,
                  initVector );
PGPFreeData( initVector );

if  ( ( IsPGPError( err ) )
{
    return( err );
}

/*
** CBC encrypt/decrypt loop code…
*/
```

PGPInitCBC

```
PGPError        PGPInitCBC(
                     PGPCBCContextRef   chainingContext,
                     const void         *key,
                     const void         *initializationVector );
```

Arguments

`chainingContext`	the target CBC context
`key`	the desired key
`initializationVector`	the desired initialization vector data

Description

Establishes the key and/or intialization vector for the cipher chaining context. One of `key` and `initializationVector` may be `NULL`, but not both.

Notes, Warnings, and Tips

The **initialization vector** (IV) size is assumed to be the same as the symmetric cipher block size.

Since both arguments are copied into the cipher chaining context, the caller is encouraged to clear their memory upon successful return.

Both `key` and `initializationVector` must be set prior to any cipher operations. However, as a convenience to the PGPsdk developer, these may be set in separate calls to `PGPInitCBC` and/or `PGPInitSymmetricCipher` since these values are commonly obtained from different sources at different times.

If the PGPsdk developer neglects to call `PGPInitCBC` to set the initialization vector (for example, always sets the key via `PGPInitSymmetricCipher`), then the initialization vector defaults to zeroes. Generally, it is better cryptographic practice to set the initialization vector to random data.

PGPCopyCBCContext

```
PGPError         PGPCopyCBCContext(
                      PGPCBCContextRef   chainingContextOrig,
                      PGPCBCContextRef   *chainingContextCopy );
```

Arguments

`chainingContextOrig`
the source CBC context

`chainingContextCopy`
the receiving field for the copy of the CBC context

Description

Creates an exact copy of the source chaining cipher context.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant chaining cipher context copy with `PGPFreeCBCCipherContext`.

PGPFreeCBCContext

```
PGPError         PGPFreeCBCContext(
                      PGPCBCContextRef   chainingContext );
```

Arguments

`chainingContext`	the target cipher block chaining context

Description

Decrements the reference count for the specified cipher block chaining context, and frees the context if the reference count reaches zero.

Notes, Warnings, and Tips

Before deallocating the context, the function erases all associated in-memory data.

PGPCBCEncrypt

```
PGPError         PGPCBCEncrypt(
                      PGPCBCContextRef   chainingContext,
                      const void         *plainText,
                      PGPSize            plainTextLength,
                      void               *cipherText );
```

Arguments

`chainingContext`	the target CBC context
`plainText`	the data to encrypt
`plainTextLength`	the length of the data to encrypt (in bytes)
`cipherText`	the receiving buffer for the resultant encrypted data

Description

Encrypts the specified data according to the specified cipher block chaining context.

Notes, Warnings, and Tips

Since cipher block chaining effectively changes the key for each block of plain text, `PGPCBCEncrypt` can be called repeatedly to encrypt arbitrary amounts of data.

Sample Code

```
PGPSymmetricCipherContext    cipherContext;
PGPSize                      keySize;
PGPSize                      blockSize;

if  ( IsPGPError( err = PGPCBCGetSymmetricCipher( chainingContext,
                                                  &cipherContext );
{
    return( err );
}

if  ( IsPGPError( err = PGPGetSymmetricCipherSizes( cipherContext,
                                                     &keySize,
                                                     &blockSize );
{
    return( err );
}
```

```
while (plainTextLength >= blockSize )
{
    if  ( IsPGPError( err = PGPCBCEncrypt( chainingContext,
                                           plainText,
                                           blockSize,
                                           cipherText ) ) )
    {
          return( err );
    }
    plainText += blockSize;
    cipherText += blockSize;
    plainTextLength -= blockSize;
}
if  ( IsPGPError( err = PGPCBCEncrypt( chainingContext,
                                       plainText,
                                       plainTextLength,
                                       cipherText ) ) )
{
    return( err );
}
```

PGPCBCDecrypt

```
PGPError        PGPCBCDecrypt(
                     PGPCBCContextRef   chainingContext,
                     const void         *cipherText,
                     PGPSize            cipherTextLength,
                     void               *plainText );
```

Arguments

`chainingContext`	the target CBC context
`cipherText`	the data to decrypt
`cipherTextLength`	the length of the data to decrypt (in bytes)
`plainText`	the receiving buffer for the resultant plain text

Description

Decrypts the specified data according to the specified chaining context.

PGPCBCGetSymmetricCipher

```
PGPError        PGPCBCGetSymmetricCipher(
                     PGPCBCContextRef   chainingContext,
                     PGPSymmetricCipherContextRef
                                        *cipherContext );
```

Arguments

`chainingContext`	the target CBC context
`cipherContext`	the receiving field for the symmetric cipher context

Description

Get the symmetric cipher context being used by the specified cipher block chaining context.

Notes, Warnings, and Tips

`cipherContext` is the actual `PGPSymmetricCipherContext`, and *not* a copy. Since the chaining context “owns” the symmetric cipher, the caller should neither free nor dereference it, but may copy it.

Once obtained, the symmetric cipher reference can be used to obtain attributes of the underlying cipher, for example, its block size.

PGPNewCFBContext

```
PGPError         PGPNewCFBContext(
                      PGPSymmetricCipherContextRef
                                          cipherContext,
                      PGPUInt16           interleaveFactor,
                      PGPCFBContextRef    *feedbackContext );
```

Arguments

`cipherContext`	the underlying symmetric cipher context
`interleaveFactor`	the desired number of cipher blocks in the feedback loop
`feedbackContext`	the receiving field for the resultant CFB context

Description

Creates a new feedback context based upon the specified symmetric cipher. The specified interleave factor determines the number of cipher blocks through which the feedback mechanism will cycle.

Notes, Warnings, and Tips

A cipher feedback context requires use of a symmetric cipher that has been created and whose key has been set. This key may be set explicitly with `PGPInitSymmetricCipher`, or set implicitly with `PGPInitCFB`.

After the call, the `CFBRef` "owns" the symmetric `cipherContext` and will dispose of it properly (even if an error occurs). The caller should no longer reference it.

The choice of interleave factor affects the size of the resultant feedback context, but does not affect its performance. However, while the PGPsdk API currently supports interleaving, it is not yet fully implemented. As such, the interleave factor should always be specified as one.

Sample Code

```
PGPError                    err;
PGPSymmetricCipherContextRef
                            cipherContext;
PGPSize                     keySize;
PGPSize                    blockSize;
#define kMyCipherAlg       kPGPCipherAlgorithm_CAST5
/*
** The choice of key size must be compatible
** with the choice of cipher algorithm!
*/
#define kMyKeySize          ( ( PGPSize )64 ) /* Bits!    */
PGPCFBContextRef            feebackContext;
#define kMyInterleaveFactor 1
void                        key[ ( ( kMyKeySize / 8 ) * kMyInterleaveFactor ) ];
void                        *initVector;

if  ( IsPGPError( err = PGPNewSymmetricCipherContext( pgpContext,
                                                      kMyCipherAlg,
                                                      kMyKeySize,
                                                      &cipherContext ) ) )
{
    return( err );
}

if  ( IsPGPError( err = PGPGetSymmetricCipherSizes( cipherContext,
                                                    &keySize,
                                                    &blockSize ) ) )
{
    return( err );
}

if  ( IsPGPError( err = PGPNewCFBContext( cipherContext,
                                          kMyInterleaveFactor,
                                          &feebackContext ) ) )
{
    return( err );
}

if  ( ( initVector = PGPNewSecureData( pgpContext,
                                       ( blockSize * kMyInterleaveFactor ),
                                        &didLock ) ) == (void *)NULL )
{
    return( err );
}

myKeySetFunction( &key[ 0 ],
                 ( kMyKeyBitSize / (PGPSize)8 ),
                  kMyInterleaveFactor );
myInitVectorSetFunction( initVector,
                        ( blockSize * kMyInterleaveFactor ) );

if  ( IsPGPError( err = PGPInitCFB( cipherContext,
                                    &key[ 0 ],
                                    &initVector ) ) )
myMemClearFunction( &key[ 0 ],
                   ( kMyKeyBitSize / (PGPSize)8 ),
                  kMyInterleaveFactor );
```

```
PGPFreeData( initVector );

if  ( IsPGPError( err ) )
{
    return( err );
}

/*
** CFB encrypt/decrypt loop code…
*/
```

PGPInitCFB

```
PGPError         PGPInitCFB(
                      PGPCFBContextRef   feedbackContext,
                      const void         *key,
                      const void         *initializationVector );
```

Arguments

`feedbackContext`	the target CFB context
`key`	the desired key data
`initializationVector`	the desired initialization vector data

Description

Establishes the key(s) and/or intialization vector(s) for the cipher feeback context. One of `key` and `initializationVector` may be `NULL`, but not both.

Notes, Warnings, and Tips

The key data size is assumed to be the key size of the associated symmetric cipher, times the feedback context's interleave factor; the initialization vector (IV) data size is assumed to be the block size of the associated symmetric cipher, times the feedback context's interleave factor.

Since both arguments are copied into the cipher feedback context, the caller is encouraged to clear their memory upon successful return.

Both `key` and `initializationVector` must be set prior to any cipher operations. However, as a convenience to the PGPsdk developer, these may be set in separate calls to `PGPInitCFB` and/or `PGPInitSymmetricCipher` since these values are commonly obtained from different sources at different times.

If the PGPsdk developer neglects to call `PGPInitCFB` to set the initialization vector (for example, always sets the key via `PGPInitSymmetricCipher`), then the initialization vector defaults to zeroes. Generally, it is better cryptographic practice to set the initialization vector to random data.

PGPCopyCFBContext

```
PGPError         PGPCopyCFBContext(
                      PGPCFBContextRef   feebackContextOrig,
                      PGPCFBContextRef   *feebackContextCopy );
```

Arguments

`feebackContextOrig`	the source CFB context
`feebackContextCopy`	the receiving field for the copy of the CFB context

Description

Creates an exact copy of the source feedback cipher context.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant feedback cipher context copy with `PGPFreeCFBCipherContext`.

PGPFreeCFBContext

```
PGPError          PGPFreeCFBContext(
                       PGPCFBContextRef   feedbackContext );
```

Arguments

`feedbackContext`	the target cipher feedback context

Description

Decrements the reference count for the specified cipher feedback context, and frees the context if the reference count reaches zero.

Notes, Warnings, and Tips

Before deallocating the context, the function erases all associated in-memory data.

PGPCFBEncrypt

```
PGPError          PGPCFBEncrypt(
                       PGPCFBContextRef   feedbackContext,
                       const void         *plainText,
                       PGPSize            plainTextLength,
                       void               *cipherText );
```

Arguments

`feedbackContext`	the target CFB context
`plainText`	the data to encrypt
`plainTextLength`	the length of the data to encrypt (in bytes)
`cipherText`	the receiving buffer for the resultant encrypted data

Description

Encrypts the specified data according to the specified feedback context.

Notes, Warnings, and Tips

Call repeatedly to encrypt arbitrary amounts of data.

PGPCFBDecrypt

```
PGPError          PGPCFBDecrypt(
                       PGPCFBContextRef   feedbackContext,
                       const void         *cipherText,
                       PGPSize            cipherTextLength,
                       void               *plainText );
```

Arguments

`feedbackContext`	the target CFB context
`cipherText`	the data to decrypt
`cipherTextLength`	the length of the data to decrypt (in bytes)
`plainText`	the receiving buffer for the resultant plain text

Description

Decrypts the specified data according to the specified feedback context.

PGPCFBGetSymmetricCipher

```
PGPError          PGPCFBGetSymmetricCipher(
                       PGPCFBContextRef   feedbackContext,
                       PGPSymmetricCipherContextRef
                                          *cipherContext );
```

Arguments

`feedbackContext`	the target CFB context
`cipherContext`	the receiving field for the context of the associated symmetric cipher

Description

Get the symmetric cipher context associated with the specified cipher feedback context.

Notes, Warnings, and Tips

`cipherContext` is the actual `PGPSymmetricCipherContext`, and *not* a copy. Since the feedback context “owns” the symmetric cipher, the caller should neither free nor dereference it, but may copy it.

Once obtained, the symmetric cipher reference can be used to obtain attributes of the underlying cipher, for example, its block size.

PGPCFBGetRandom

```
PGPError          PGPCFBGetRandom(
                       PGPCFBContextRef   feedbackContext,
                       PGPSize            requestCount,
                       void               *randomData,
                       PGPSize            *randomDataCount );
```

Arguments

`feedbackContext`	the target CFB context
`requestCount`	the maximum number of pseudo-random bytes to fetch
`randomData`	the receiving buffer for the pseudo-random bytes
`randomDataCount`	the receiving field for the actual number of pseudo-random bytes fetched

Description

Fetches pseudo-random bytes from the specified cipher feedback context, and indicates the actual number of pseudo-random bytes obtained. A maximum of `requestCount` bytes are fetched.

Notes, Warnings, and Tips

The receiving buffer must be at least `requestCount` bytes in length.

PGPCFBRandomCycle

```
PGPError         PGPCFBRandomCycle(
                      PGPCFBContextRef   feedbackContext,
                      const void         *salt );
```

Arguments

`feedbackContext`	the target CFB context
`salt`	the additional random byte data

Description

Makes more pseudo-random bytes available by iterating through the existing random number pool, and applying the supplied **salt**.

Notes, Warnings, and Tips

The number of salt bytes is assumed to equal to the block size of the associated symmetric cipher.

This function will assert if the associated interleave factor is not exactly one.

PGPCFBRandomWash

```
PGPError         PGPCFBRandomWash(
                      PGPCFBContextRef   feedbackContext,
                      const void         *washData,
                      PGPSize            washDataLength );
```

Arguments

`feedbackContext`	the target CFB context
`washData`	the wash data
`washDataLength`	the length of the wash data

Description

Hashes the associated specified symmetric cipher’s key and initialization vector with the specified wash data to produce a new key and a new initialization vector.

Notes, Warnings, and Tips

If washDataLength is less than the symmetric cipher block size, then padding bytes are used. If washDataLength is greater than the symmetric cipher block size, then multiple iterations occur. Passing “extra” wash data never reduces the resultant cryptographic strength of the resultant cipher text, and often increases it.

This function will assert if the associated interleave factor is not exactly one.

PGPCFBSync

```
PGPError         PGPCFBSync(
                      PGPCFBContextRef    feedbackContext );
```

Arguments

feedbackContext	the target CFB context

Description

Reset the feedback mechanism to use the currently available data plus an additional number of previous bytes, such that the resultant data length equals the cipher block size.

Notes, Warnings, and Tips

This effectively changes the cipher block boundary.

PGPNewPublicKeyContext

```
PGPError         PGPNewPublicKeyContext(
                      PGPKeyRef           key,
                      PGPPublicKeyMessageFormat
                                          messageFormat,
                      PGPPublicKeyContextRef
                                          *publicKeyContext );
```

Arguments

pgpContext	the target key
messageFormat	the desired message format, which recognizes kPGPPublicKeyMessageFormat_… values (see Table 3-6)
publicKeyContext	the receiving field for the resultant public key context

Description

Creates a context for public key operations based on the specified key and using the specified message format.

PGPFreePublicKeyContext

```
PGPError         PGPFreePublicKeyContext(
                      PGPPublicKeyContextRef
                                          publicKeyContext );
```

Arguments

`publicKeyContext` the target public key context

Description

Decrements the reference count for the specified public key context, and frees the context if the reference count reaches zero.

PGPGetPublicKeyOperationsSizes

```
PGPError        PGPGetPublicKeyOperationsSizes(
                     PGPPublicKeyContextRef
                                         publicKeyContext,
                     PGPSize             *maxDecryptedBufferSize,
                     PGPSize             *maxEncryptedBufferSize,
                     PGPSize             *maxSignatureSize );
```

Arguments

`publicKeyContext` the target public key context

`maxDecryptedBufferSize`
the receiving field for the decryption buffer size (in bytes)

`maxEncryptedBufferSize`
the receiving field for the encryption buffer size (in bytes)

`maxSignatureSize` the receiving field for the signature size (in bytes)

Description

Returns the sizes associated with the specified public key context. A resultant value of zero indicates that the associated operation is not available, for example if `maxSignatureSize` is zero, then signing is not a supported operation.

PGPPublicKeyEncrypt

```
PGPError        PGPPublicKeyEncrypt(
                     PGPPublicKeyContextRef
                                         publicKeyContext,
                     void const          *plainText,
                     PGPSize             plainTextLength,
                     void                *cipherText,
                     PGPSize             *cipherTextLength );
```

Arguments

`publicKeyContext`	the target public key context
`plainText`	the buffer containing the input plain text
`plainTextLength`	the length of the input plain text
`cipherText`	the receiving buffer for the output cipher text, which must be at least `maxEncryptedBufferSize` (obtained from `PGPGetPublicKeyOperationsSize`)
`cipherTextLength`	the receiving field for the resultant length of the output cipher text

Description

Encrypts one block of data, using **PKCS**-1 padding.

PGPPublicKeyVerifySignature

```
PGPError         PGPPublicKeyVerifySignature(
                      PGPPublicKeyContextRef
                                          publicKeyContext,
                      PGPHashContextRef   hashContext,
                      void const          *signature,
                      PGPSize             signatureSize );
```

Arguments

`publicKeyContext`	the target public key context
`hashContext`	the target hash context
`signature`	the target signature
`signatureSize`	the length of the target signature

Description

Verifies a signature on a message hash, which is finalized by this call.

Notes, Warnings, and Tips

The message hash should not have been finalized prior to the call.

PGPNewPrivateKeyContext

```
PGPError         PGPNewPrivateKeyContext(
                      PGPKeyRef           key,
                      PGPPrivateKeyMessageFormat
                                          messageFormat,
                      char const          *passphrase,
                      PGPPrivateKeyContextRef
                                          *privateKeyContext );
```

Arguments

`pgpContext`	the target key, which must be a public/private key pair
`messageFormat`	the desired message format, which recognizes `kPGPPublicKeyMessageFormat_`… values (see Table 3-6)
`passphrase`	the passphrase associated with the target key
`privateKeyContext`	the receiving field for the resultant private key context

Description

Creates a context for private key operations based on the specified key and using the specified message format.

PGPFreePrivateKeyContext

```
PGPError         PGPFreePrivateKeyContext(
                      PGPPrivateKeyContextRef
                                          privateKeyContext );
```

Arguments

`privateKeyContext`	the target `private` key context

Description

Decrements the reference count for the specified private key context, and frees the context if the reference count reaches zero.

Notes, Warnings, and Tips

Before deallocating the context, the function erases all sensitive in-memory data.

PGPGetPrivateKeyOperationsSizes

```
PGPError         PGPGetPrivateKeyOperationsSizes(
                      PGPPrivateKeyContextRef
                                          privateKeyContext,
                      PGPSize             *maxDecryptedBufferSize,
                      PGPSize             *maxEncryptedBufferSize,
                      PGPSize             *maxSignatureSize );
```

Arguments

`privateKeyContext`	the target private key context
`maxDecryptedBufferSize`	the receiving field for the decryption buffer size (in bytes)
`maxEncryptedBufferSize`	the receiving field for the encryption buffer size (in bytes)
`maxSignatureSize`	the receiving field for the signature size (in bytes)

Description

Returns the sizes associated with the specified private key context. A resultant value of zero indicates that the associated operation is not available.

PGPPrivateKeyDecrypt

```
PGPError         PGPPrivateKeyDecrypt(
                      PGPPrivateKeyContextRef
                                          privateKeyContext,
                      void const          *cipherText,
                      PGPSize             cipherTextLength,
                      void                *plainText,
                      PGPSize             *plainTextLength );
```

Arguments

`privateKeyContext`	the target private key context
`cipherText`	the buffer containing the input cipher text
`cipherTextLength`	the length of the input cipher text
`plainText`	the receiving buffer for the output plain text, which must be at least `maxDecryptedBufferSize` (obtained from `PGPGetPrivateKeyOperationsSize`)
`plainTextLength`	the receiving field for the resultant length of the output plain text

Description

Decrypts one block of data.

Errors

`kPGPError_FeatureNotAvailable`

PGPPrivateKeySign

```
PGPError         PGPPrivateKeySign(
                      PGPPrivateKeyContextRef
                                          privateKeyContext,
                      PGPHashContextRef   hashContext,
                      void                *signature,
                      PGPSize             *signatureSize );
```

Arguments

`privateKeyContext`	the target private key context
`hashContext`	the target hash context
`signature`	the receiving field for the signature, which must be at least `maxSignatureSize` (obtained from `PGPGetPrivateKeyOperationsSize`)
`signatureSize`	the receiving field for the resultant length of the signature

Description

Obtains the signature associated with the specified private key context, as well as its length (in bytes).

Errors

`kPGPError_BadSignature`

```
kPGPError_FeatureNotAvailable
```

Notes, Warnings, and Tips

The message hash should not have been finalized prior to the call.

Chapter 4: Function Reference – Option List Functions

Introduction

The PGPsdk provides a flexible and extensible mechanism for presenting arbitrary option specifications and data to functions accepting this mechanism.

Since the option list mechanism was originally developed for encode/decode operations, the function prototypes are listed in the public header file `pgpEncode.h`. Future PGPsdk versions may provide a separate public header file for these functions.

The option list management functions and the individual option functions use copy semantics. That is, they create their own copy of the arguments, and so allow the caller to delete the argument data upon return. This is very important in the case of passphrase and other sensitive data. In these cases, the caller should not only free the memory occupied by the argument, but also ensure that the memory is first erased. Additionally, the individual option functions allocate `PGPOptionListRef` instances that are automatically deallocated once they are used in an option list management function (for example, `PGPBuildOptionList`), or as a sub-option (for example, `PGPOSignWithKey( …, PGPOPassphrase ( … ), … )`).

The individual option functions do *not* return the data type `PGPError`; instead they always return the data type `PGOptionListRef`. However, an error may have occurred, and the resultant option list may not be valid (this is almost always due to `kPGPError_BadParams`, but may also be `kPGPError_OutOfMemory`). Since this condition can not be detected reliably until the resultant option list is actually used, the PGPsdk developer should always consider these option list functions as being a potential failure point for functions accepting option list arguments.

The macro `PGPOptionListRefIsValid` can be used to determine validity, but only checks against the value `kInvalidPGPOptionListRef`. Other values that are also invalid but not currently exported include:

```
kPGPEndOfArgsOptionListRef        ((PGPOptionListRef) -1L)
kPGPOutOfMemoryOptionListRef      ((PGPOptionListRef) -2L)
kPGPBadParamsOptionListRef        ((PGPOptionListRef) -3L)
kPGPNullOptionListRef             ((PGPOptionListRef) -4L)
```

These constants may be exported in future versions of the PGPsdk, and their values are subject to change.

Header Files

`pgpEncode.h`

Option List Management Functions

Option list management functions create and act upon persistent option lists, which must later be explicitly freed.

PGPNewOptionList

```
PGPError         PGPNewOptionList(
                      PGPContextRef       pgpContext,
                      PGPOptionListRef    *outList );
```

Arguments

`pgpContext` the target context
`outList` the receiving field for the resultant option list

Description

Creates an empty, persistent option list, which may then be the output target for subsequent `PGPAppendOptionList` and `PGPBuildOptionList` function calls.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant option list via `PGPFreeOptionList`.

PGPBuildOptionList

```
PGPError         PGPBuildOptionList(
                      PGPContextRef       pgpContext,
                      PGPOptionListRef    *outList,
                      PGPOptionListRef    firstOption,
                      ...,
                      PGPOLastOption( void ) );
```

Arguments

`pgpContext` the target context
`outList` the receiving field for the resultant option list
`firstOption` the initial option list instance
`...` subsequent option list instances
`PGPOLastOption( void )`
must always appear as the final argument to terminate the argument list

Description

Populates a persistent option list, replacing any previous content. Argument option list instances may be embedded option list function calls and/or previously built `PGPOptionListRef` instances, thus supporting modular assembly of option lists.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant option list via `PGPFreeOptionList`.

PGPCopyOptionList

```
PGPError         PGPCopyOptionList(
                      PGPConstOptionListRef
                                          optionListOrig,
                        PGPOptionListRef   *optionListCopy );
```

Arguments

`optionListOrig`	the source option list
`optionListCopy`	the receiving field for the copy of the option list

Description

Creates a persistent, exact copy of the source option list.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant copy of the option list via `PGPFreeOptionList`.

PGPAppendOptionList

```
PGPError          PGPAppendOptionList(
                        PGPContextRef      pgpContext,
                        PGPOptionListRef   outList,
                        PGPOptionListRef   firstOption,
                        ...,
                        PGPOLastOption( void ) );
```

Arguments

`pgpContext`	the target context
`outList`	the existing option list to which the specified option list instances will be appended
`firstOption`	the initial option list instance
`...`	subsequent option list instances
`PGPOLastOption( void )`	must always appear as the final argument to terminate the argument list

Description

Augments a persistent option list by appending the specified option(s) to any existing content. Argument option list instances may be embedded option list function calls and/or previously built `PGPOptionListRef` instances, thus supporting modular assembly of option lists.

PGPFreeOptionList

```
PGPError          PGPFreeOptionList(
                        PGPOptionListRef   optionList );
```

Arguments

`optionList`	the existing option list to be deallocated

Description

Decrements the reference count for the specified option list (created by `PGPNewOptionList`, `PGPBuildOptionList`, or `PGPCopyOptionList`), and frees the option list if the reference count reaches zero.

Notes, Warnings, and Tips

Option lists that result from the inclusion of PGPO… functions in an argument list are automatically deallocated upon return from the employing function. Such employing functions include:

- PGPEncode
- PGPDecode
- PGPBuildOptionList
- PGPAppendOptionList
- PGPAddJobOptionList

PGPAddJobOptions

```
PGPError          PGPAddJobOptions(
                       PGPJobRef           theJob,
                       PGPOptionListRef    firstOption,
                       ...,
                       PGPOLastOption( void ) );
```

Arguments

theJob	the current job
firstOption	the initial option list instance
...	subsequent option list instances
PGPOLastOption(void)	must always appear as the final argument to terminate the argument list

Description

Pass new option information to the job upon receipt of certain events. The job argument should be passed as `event->job`. Additional `PGPOptionListRef` arguments can be specified similarly to the way they are passed to `PGPEncode` and `PGPDecode`. However, only certain options can be set after each type of event. The legal options are described for each event, as well as enumerated in the preceding Constants and Data Structures section.

Common Encode/Decode Option List Functions

The following functions are used to create `PGPOptionListRef` instances for specifying the various common options to either `PGPDecode` or `PGPEncode`. These functions can be used as temporary inline arguments, or presented to `PGPAppendOptionList` and `PGPBuildOptionList` to augment or create existing persistent lists.

PGPOInputBuffer

```
PGPOptionListRef PGPOInputBuffer(
                       PGPContextRef       pgpContext,
                       void const          *inBuf,
                       PGPSize             inBufSize );
```

Arguments

`pgpContext`	the target context
`inBuf`	the desired input buffer
`inBufSize`	the length of the input data in the desired input buffer

Description

Specifies that input is to be taken from the referenced buffer.

Notes, Warnings, and Tips

One of `PGPOInputBuffer`, `PGPOInputFile`, and `PGPOInputFileFSSpec` is required to specify an input source for functions that accept this option.

If this option is specified in addition to an input file option, then the operation will fail with `kPGPError_BadParams`.

PGPOInputFile

```
PGPOptionListRef PGPOInputFile(
                      PGPContextRef       pgpContext,
                      PGPConstFileSpecRef
                                          fileSpec );
```

Arguments

`pgpContext`	the target context
`fileSpec`	the desired input file specification

Description

Specifies that input is to be taken from the indicated file.

Notes, Warnings, and Tips

One of `PGPOInputBuffer`, `PGPOInputFile`, and `PGPOInputFileFSSpec` is required to specify an input source for functions that accept this option.

If this option is specified in addition to an input buffer option, then the operation will fail with `kPGPError_BadParams`.

PGPOInputFileFSSpec (*MacOS platforms only*)

```
PGPOptionListRef PGPOInputBuffer(
                      PGPContextRef       pgpContext,
                      void const          *buffer,
                      const FSSpec        *fileSpec );
```

Arguments

`pgpContext`	the target context
`fileSpec`	the desired input FS specification

Description

Specifies that input is to be taken from the indicated file.

Notes, Warnings, and Tips

One of `PGPOInputBuffer`, `PGPOInputFile`, and `PGPOInputFileFSSpec` is required to specify an input source for functions that accept this option.

If this option is specified in addition to an input buffer option, then the operation will fail with `kPGPError_BadParams`.

PGPODiscardOutput

```
PGPOptionListRef PGPODiscardOutput(
                      PGPContextRef      pgpContext,
                      PGPBoolean         discardOutput );
```

Arguments

`pgpContext`	the target context
`discardOutput`	TRUE if the output is to be discarded

Description

Specifies whether or not the output should be discarded, for example, sent to the null device.

Notes, Warnings, and Tips

One of `PGPODiscardOutput`, `PGPOOutputFile`, `PGPOOutputBuffer`, and `PGPOOutputFileFSSpec` is required to specify an output destination for functions that accept this option.

If this option is specified with either an output file or an output buffer option, then the operation will fail with `kPGPError_BadParams`.

PGPOAllocatedOutputBuffer

```
PGPOptionListRef PGPOAllocatedOutputBuffer(
                      PGPContextRef      pgpContext,
                      void               **outputBuffer,
                      PGPSize            maximumBufferSize,
                      PGPSize            *actualBufferSize );
```

Arguments

`pgpContext`	the target context
`outputBuffer`	the receiving field for a pointer to the allocated buffer
`maximumBufferSize`	the maximum size to which the buffer may grow
`actualBufferSize`	the receiving field for the actual size of the buffer

Description

Specifies that output should be placed in a dynamically allocated buffer. Upon completion of the operation, `outputBuffer` will contain a pointer to the buffer, and `actualBufferSize` will contain the length of the data in the output buffer.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant buffer with `PGPFreeData`.

Sample Code

```
PGPError   err;
void       *outputBuffer;
PGPSize    actualBufferSize;

err = PGPDecode( pgpContext,
                 PGPOInputFile( pgpContext,
                                fileSpec ),
                 PGPOAllocatedOutputBuffer( pgpContext,
                                            &outputBuffer,
                                            (PGPSize)( 1024 * 1024 ),
                                            &actualBufferSize ),
                 PGPOPassphrase( pgpContext,
                                 passphraseBuf ),
                 PGPOLastOption( void ) );

if ( IsntPGPError( err ) )
{
    myProcessFunction( outputBuffer,
                       actualBufferSize );
    PGPFreeData( outputBuffer,
                 actualBufferSize );
}
```

PGPOOutputBuffer

```
PGPOptionListRef PGPOOutputBuffer(
                         PGPContextRef      pgpContext,
                         void               *outBuf,
                         PGPSize            outBufSize,
                         PGPSize            *outBufDataLength );
```

Arguments

`pgpContext`	the target context
`outBuf`	the desired output buffer
`outBuf`	the available size of the desired output buffer
`outBufDataLength`	the receiving field for the actual length of the data output

Description

Specifies that output should be placed in a statically allocated buffer. Upon completion of the operation, `outBufDataLength` will contain the actual size of the output.

Notes, Warnings, and Tips

If `outputDataLength` is less than or equal to `bufferSize`, then all the output was successfully collected. If not, then some of the output data was lost.

One of `PGPODiscardOutput`, `PGPOOutputFile`, `PGPOOutputBuffer`, and `PGPOOutputFileFSSpec` is required to specify an output destination for functions that accept this option.

If this option is specified with either a discard output or an output file option, then the operation will fail with `kPGPError_BadParams`.

PGPOOutputFile

```
PGPOptionListRef PGPOOutputFile(
                      PGPContextRef       pgpContext,
                      PGPConstFileSpecRef
                                          fileSpec );
```

Arguments

`pgpContext`	the target context
`fileSpec`	the specification of the desired output file

Description

Specifies that output should be directed to the indicated file.

Notes, Warnings, and Tips

One of `PGPODiscardOutput`, `PGPOOutputFile`, `PGPOOutputBuffer`, and `PGPOOutputFileFSSpec` is required to specify an output destination for functions that accept this option.

If this option is specified with either a discard output or an output buffer option, then the operation will fail with `kPGPError_BadParams`.

PGPOOutputFileFSSpec **(*MacOS platforms only*)**

```
PGPOptionListRef PGPOOutputFileFSSpec(
                      PGPContextRef       pgpContext,
                      const FSSpec        *fileSpec );
```

Arguments

`pgpContext`	the target context
`fileSpec`	the FS specification of the desired output file

Description

Specifies that output should be directed to the indicated file.

Notes, Warnings, and Tips

One of `PGPODiscardOutput`, `PGPOOutputFile`, `PGPOOutputBuffer`, and `PGPOOutputFileFSSpec` is required to specify an output destination for functions that accept this option.

If this option is specified with either a discard output or an output buffer option, then the operation will fail with `kPGPError_BadParams`.

PGPOAppendOutput

```
PGPOptionListRef PGPOAppendOutput(
                      PGPContextRef      pgpContext,
                      PGPBoolean         appendOutput );
```

Arguments

`pgpContext`	the target context
`appendOutput`	`TRUE` if the output is to be appended to any associated file or buffer

Description

Specifies whether or not output should be appended to any associated file or buffer, or should overwrite it.

PGPOPGPMIMEEncoding

```
PGPOptionListRef PGPOPGPMIMEEncoding(
                      PGPContextRef      pgpContext,
                      PGPBoolean         mimeEncoding,
                      PGPSize            *mimeBodyOffset,
                      char               mimeSeparator[
                                              kPGPMimeSeparatorSize ] );
```

Arguments

`pgpContext` the target context

`mimeEncoding` `TRUE` if the output should be in **MIME** format

`mimeBodyOffset` a field that will be used by the encoding process to hold the offset of the MIME body text, which is ignored if `mimeEncoding` is `FALSE`

`mimeSeparator` a buffer that will be used by the encoding process to hold the MIME separator text, which must have a minimum length of `kPGPMimeSeparatorSize`, which is ignored if `mimeEncoding` is `FALSE`

Description

Specifies whether or not the output should be in MIME format. If `mimeEncoding` is `TRUE`, then `mimeBodyOffset` is initialized to zero, and `mimeSeparator` is initialized to an empty string, assuming that they are non-`NULL`.

Notes, Warnings, and Tips

This option forcibly sets `PGPOArmorOutput`.

PGPOOmitMIMEVersion

```
PGPOptionListRef PGPOOmitMIMEVersion(
                     PGPContextRef      pgpContext,
                     PGPBoolean         omitMIMEVersion );
```

Arguments

`pgpContext` the target context

`omitMIMEVersion` `TRUE` if the MIME version should *not* be included in the output

Description

Specifies whether or not the MIME version should be included in the output, since some mailers automatically add the MIME version to their output. By specifying `TRUE`, the PGPsdk developer can avoid inclusion of two MIME version entries.

Notes, Warnings, and Tips

This option is only meaningful in conjunction with a `PGPOPGPMIMEEncoding` instance that enables MIME format.

PGPOLocalEncoding

```
PGPOptionListRef PGPOLocalEncoding(
                     PGPContextRef      pgpContext,
                     PGPLocalEncodingFlags
                                        localEncode );
```

Arguments

`pgpContext`	the target context
`localEncode`	the encoding to use, which recognizes `kPGPLocalEncoding_…` values (see Table 3-10)

Description

Specifies the conditions under which the output should be converted to a platform-specific encoding. Currently, the PGPsdk only supports conversion to MacOS MacBinary format, and this function effectively does nothing on non-MacOS platforms. The local encoding flag values have the following meanings:

- `kPGPLocalEncoding_Auto` - effect conversion depending upon the output MacOS `OSType` file type
- `kPGPLocalEncoding_Force` - always effect conversion
- `kPGPLocalEncoding_NoMacBinCRCOkay` - flag the converted output such that a subsequent decode or signature verification ignores a failed CRC check
- `kPGPLocalEncoding_None` - no-op

The `kPGPLocalEncoding_Auto` and `kPGPLocalEncoding_Force` options are considered “main” options, and are mutually exclusive. `kPGPLocalEncoding_NoMacBinCRCOkay` and `kPGPLocalEncoding_None` are considered “modifier” options, and are intended to be `OR`’ed with one of the main options.

Notes, Warnings, and Tips

`kPGPLocalEncoding_NoMacBinCRCOkay` is primarily intended to provide compatibility with *PGP Version 2.6.2*.

When specified for `PGPDecode,` the option applies only to any detached signatures.

Generally, the PGPsdk developer should always specify `kPGPLocalEncoding_Force` since this:

- ensures that no data will be lost
- is ignored for output on non-MacOS platforms
- is recognized for input by PGP version 5.5 software products on non-MacOS platforms

Sample Code

```
tOptListRef = PGPOLocalEncoding( pgpContext,
                                 ( kPGPLocalEncoding_Force |
                                   kPGPLocalEncoding_NoMacBinCRCOkay ) );
```

PGPOOutputLineEndType

```
PGPOptionListRef PGPOOutputLineEndType(
                        PGPContextRef       pgpContext,
                        PGPLineEndType      lineEndType );
```

Arguments

`pgpContext`	the target context
`lineEndType`	the line ending to use, which recognizes `kPGPLineEnd_`... values (see Table 3-9)

Description

Specifies the type of line endings to use when generating text output.

Notes, Warnings, and Tips

This option is only meaningful in conjunction with `PGPOArmorOutput`.

If not specified, then the default line endings for the local platform is used.

PGPODetachedSig

```
PGPOptionListRef PGPODetachedSig(
                      PGPContextRef      pgpContext,
                      PGPOptionListRef   firstOption,
                      ...,
                      PGPOLastOption( void ) );
```

Arguments

`pgpContext`	the target context
`firstOption`	the initial option list instance
`...`	subsequent option list instances
`PGPOLastOption( void )`	must always appear as the final argument to terminate the argument list

Description

For `PGPEncode,` creates a detached signature for the message. No sub-options are defined at this time.

For `PGPDecode,` specifies the input source to be used to verify any associated detached signature. In this case, one of `PGPOInputBuffer`, `PGPOInputFile`, and `PGPOInputFileFSSpec` is required.

Common Encrypting and Signing Option List Functions

PGPOConventionalEncrypt

```
PGPOptionListRef PGPOConventionalEncrypt(
                      PGPContextRef      pgpContext,
                      PGPOptionListRef   firstOption,
                      ...,
                      PGPOLastOption( void ) );
```

Arguments

`pgpContext`	the target context
`firstOption`	the initial option list instance
`...`	subsequent option list instances
`PGPOLastOption( void )`	must always appear as the final argument to terminate the argument list

Description

Conventionally encrypt the message.

Notes, Warnings, and Tips

This requires a `PGPOPassphrase` sub-option to specify the conventional encryption key. The operation will fail if one is not specified.

PGPOCipherAlgorithm

```
PGPOptionListRef PGPOCipherAlgorithm(
                      PGPContextRef      pgpContext,
                      PGPCipherAlgorithm algorithm );
```

Arguments

`pgpContext`	the target context
`algorithm`	the cipher algorithm to use, which recognizes `kPGPCipherAlgorithm_`... values (see Table 3-4a)

Description

Specifies the algorithm to use for encryption. This is currently meaningful only in conjunction with conventional encryption; otherwise the choice of encryption algorithm is based on the encrypt-to keys.

PGPOEncryptToKey

```
PGPOptionListRef PGPOEncryptToKey(
                      PGPContextRef      pgpContext,
                      PGPKeyRef          keyRef );
```

Arguments

`pgpContext`	the target context
`keyRef`	the target key

Description

Encrypt the plain text to the specified key.

Notes, Warnings, and Tips

To encrypt the plain text with multiple keys, include an instance of this option in the `PGPEncode` option list for each key. There is no preset limit to the number of instances.

If the number of individual encrypt-to keys is large or if multiple data instances are to be encrypted, then it may be simpler to collect the keys as a key set and use `PGPOEncryptToKeySet`.

PGPOEncryptToKeySet

```
PGPOptionListRef PGPOEncryptToKeySet(
                      PGPContextRef      pgpContext,
                      PGPKeySetRef       keySet );
```

Arguments

`pgpContext`	the target context
`keySet`	the target key set

Description

Encrypt the plain text to each key in the key set. This option may be used multiple times in one call.

Notes, Warnings, and Tips

To encrypt the plain text to each key in multiple key sets, include an instance of this option in the `PGPEncode` option list for each key set. There is no preset limit to the number of instances.

PGPOEncryptToUserID

```
PGPOptionListRef PGPOEncryptToUserID(
                      PGPContextRef      pgpContext,
                      PGPUserIDRef       userIDRef );
```

Arguments

`pgpContext`	the target context
`userIDRef`	the target user ID

Description

Encrypt the plain text to the key associated with the specified user ID.

Notes, Warnings, and Tips

To encrypt the plain text with the keys associated with multiple user IDs, include an instance of this option in the `PGPEncode` option list for each user ID. There is no preset limit to the number of instances.

This function is believed to be of limited use, and may not be supported in future versions of the PGPsdk.

PGPOHashAlgorithm

```
PGPOptionListRef PGPOHashAlgorithm(
                      PGPContextRef      pgpContext,
                      PGPHashAlgorithm   algorithm );
```

Arguments

`pgpContext`	the target context
`algorithm`	the desired hash algorithm, which recognizes `kPGPHashAlgorithm_…` values (see Table 3-3)

Description

Use the specified algorithm as the hash algorithm for signatures. For example, force the use of the SHA-1 algorithm in an RSA signature.

Notes, Warnings, and Tips

DSS keys unconditionally use the SHA-1 algorithm, and are unaffected by this option.

PGPOSignWithKey

```
PGPOptionListRef PGPOSignWithKey(
                      PGPContextRef      pgpContext,
                      PGPKeyRef          sigKey,
                      PGPOptionListRef   firstOption,
                      ...,
                      PGPOLastOption( void ) );
```

Arguments

`pgpContext`	the target context
`sigKey`	the desired signing key
`firstOption`	the initial option list instance
`...`	subsequent option list instances
`PGPOLastOption( void )`	must always appear as the final argument to terminate the argument list

Description

Sign the message or file with the specified key. Any required passphrase should be specified with a `PGPOPassphrase` sub-option. A passphrase event is posted if all of the following conditions exist:

- no passphrase sub-option is specified
- the target key requires a passphrase
- an event handler is defined (see `PGPOEventHandler`)

This function additionally accepts a `PGPOCompression` sub-option.

PGPOWarnBelowValidity

```
PGPOptionListRef PGPOWarnBelowValidity(
                      PGPContextRef      pgpContext,
                      PGPValidity        minValidity );
```

Arguments

`pgpContext` the target context
`minValidity` the desired validity threshold, which recognizes `kPGPValidity_…` values (see Table 3-8)

Description

For encryption and signature verification, specifies that a warning event be sent for any encryption or signing key having a validity level less than that specified.

PGPOFailBelowValidity

```
PGPOptionListRef PGPOFailBelowValidity(
                      PGPContextRef      pgpContext,
                      PGPValidity        minValidity );
```

Arguments

`pgpContext` the target context
`minValidity` the desired validity threshold, which recognizes `kPGPValidity_…` values (see Table 3-8)

Description

For encryption, specifies that a fatal error be recognized for an encryption key having a validity level less than that specified. For signature verification, specifies that the generated signature event `keyValidity` member be set to `kPGPValidity_Invalid`.

Encode-only Option List Functions

PGPOAskUserForEntropy

```
PGPOptionListRef PGPOAskUserForEntropy(
                      PGPContextRef      pgpContext,
                      PGPBoolean         askUserForEntropy );
```

Arguments

`pgpContext` the target context
`askUserForEntropy` `TRUE` if the user should be prompted for additional entropy

Description

Specifies whether or not the user should be prompted to provide additional entropy if the global random pool entropy level drops below its minimum.

Notes, Warnings, and Tips

If the user is not to be prompted and the entropy drops below minimum, then the operation will fail with `kPGPError_OutOfEntropy`.

PGPODataIsASCII

```
PGPOptionListRef PGPODataIsASCII(
                      PGPContextRef      pgpContext,
                      PGPBoolean         dataIsASCII );
```

Arguments

`pgpContext`	the target context
`dataIsASCII`	`TRUE` if the input data should be interpreted as ASCII

Description

Force all line endings to <`CR`><`LF`> pairs prior to encoding or signing. This flags the cipher text such that `PGPDecrypt` will generate the plain text with output line endings appropriate to the decoding platform.

PGPORawPGPInput

```
PGPOptionListRef PGPORawPGPInput(
                      PGPContextRef      pgpContext,
                      PGPBoolean         isRawPGPInput );
```

Arguments

`pgpContext`	the target context
`isRawPGPInput`	`TRUE` if the input is assumed to be in raw PGP format

Description

Indicates whether or not the input is already in binary PGP format. This simplifies decryption of messages that are multiply signed and/or multiply encrypted.

Notes, Warnings, and Tips

`PGPORawPGPInput` is intended for internal use by the PGPsdk routines, and client code should specify this option rarely, if ever.

PGPOForYourEyesOnly

```
PGPOptionListRef PGPOForYourEyesOnly(
                      PGPContextRef      pgpContext,
                      PGPBoolean         forYourEyesOnly );
```

Arguments

`pgpContext`	the target context
`forYourEyesOnly`	`TRUE` to enable "for your eyes only" encryption mode

Description

Encrypt in "for your eyes only" mode. This flags the cipher text such that the output events generated during decoding will reflect `TRUE` for the `forYourEyesOnly` member of the `PGPEventOutputData`. This in turn alerts the client to the fact that the resultant plain text should not be saved to disk, or otherwise made available to other recipients.

Notes, Warnings, and Tips

This option is not enforcible by the encrypting client - the decrypting client may always choose to ignore events entirely or simply ignore this indicator.

PGPOArmorOutput

```
PGPOptionListRef PGPOArmorOutput(
                      PGPContextRef       pgpContext,
                      PGPBoolean          armorOutput );
```

Arguments

pgpContext	the target context
armorOutput	TRUE if the resultant output should be ASCII encoded

Description

Ensures that all output is encoded as 7-bit ASCII. For example, a 32-bit binary numeric value of 688,798,386 would be rendered as the ASCII text string “290E3AB2”, assuming big-endian encoding.

PGPOClearSign

```
PGPOptionListRef PGPOClearSign(
                      PGPContextRef       pgpContext,
                      PGPBoolean          clearSign );
```

Arguments

pgpContext	the target context
clearSign	TRUE if the resultant output should be clear-signed

Description

Clear-sign the message, that is, output the text as lexical section with the appropriate PGP delimeters, but do not encrypt the plain text. In this way, messages can be sent “in the clear” while still providing for authentication. This option forcibly sets both PGPOArmorOutput and PGPODataIsASCII.

Decode-only Option List Functions

PGPOImportKeysTo

```
PGPOptionListRef PGPOImportKeysTo(
                      PGPContextRef       pgpContext,
                      PGPKeySetRef        keySet );
```

Arguments

`pgpContext`	the target context
`keySet`	the target key set

Description

If any keys are found in the input, add them to the specified key set.

PGPOPassThroughIfUnrecognized

```
PGPOptionListRef PGPOPassThroughIfUnrecognized(
                      PGPContextRef      pgpContext,
                      PGPBoolean         passThrough );
```

Arguments

`pgpContext`	the target context
`passThrough`	`TRUE` if unrecognized lexical sections should *not* post an error

Description

Indicate whether or not unrecognized lexical sections should post an error.

PGPOSendEventIfKeyFound

```
PGPOptionListRef PGPOSendEventIfKeyFound(
                      PGPContextRef      pgpContext,
                      PGPBoolean         sendEventIfKeyFound );
```

Arguments

`pgpContext`	the target context
`sendEventIfKeyFound`	`TRUE` to enable sending of `kPGPEvent_KeyFound` `events`

Description

Enable or disable sending `kPGPEvent_KeyFound` events, which allows an event handler to decide what to do with keys in the input.

(Sub-)Key Generation, Augmentation, and Revocation Option List Functions

The following functions are used to create `PGPOptionListRef` instances for specifying the various common options to `PGPGenerateKey`, `PGPGenerateSubKey`, `PGPGetKeyEntropyNeeded`, `PGPAddUserID`, and `PGPSignUserID`. These functions can be used as temporary inline arguments, or used with `PGPAppendOptionList` and `PGPBuildOptionList` to augment or create existing persistent lists.

PGPOAdditionalRecipientRequestKeySet

```
PGPOptionListRef PGPOAdditionalRecipientRequestKeySet(
                      PGPContextRef      pgpContext,
                      PGPKeySetRef       arrKeySet,
                      PGPByte            arrKeyClass );
```

Arguments

`pgpContext`	the target context
`arrKeySet`	the key set containing the additional recipient request keys
`arrKeyClass`	the class of the additional recipient request keys

Description

Establish the specified key(s) as additional recipient request key(s) when generating keys with `PGPGenerateKey`.

Notes, Warnings, and Tips

This option is valid for `PGPGenerateKey` only.
`arrKeyClass` is currently ignored, and should be specified as `( PGPByte )0`.

PGPOKeyGenName

```
PGPOptionListRef PGPOKeyGenName(
                      PGPContextRef      pgpContext,
                      const void         *name,
                      PGPSize            nameLength );
```

Arguments

`pgpContext`	the target context
`name`	the desired name
`nameLength`	the length (in bytes) of the desired name, which must be between one and 256

Description

Establish the name to be used when generating keys with `PGPGenerateKey`.

Notes, Warnings, and Tips

This option is valid for `PGPGenerateKey` only.

PGPOKeyGenMasterKey

```
PGPOptionListRef PGPOKeyGenMasterKey(
                      PGPContextRef      pgpContext,
                      PGPKeyRef          masterKey );
```

Arguments

`pgpContext`	the target context
`masterKey`	the “parent” key

Description

Specifies the key on which a sub-key will be generated.

Notes, Warnings, and Tips

This option is valid for `PGPGenerateSubKey` only.

PGPOExportPrivateKeys

```
PGPOptionListRef PGPOExportPrivateKeys(
                      PGPContextRef      pgpContext,
                      PGPBoolean         exportPrivateKeys );
```

Arguments

`pgpContext`	the target context
`exportPrivateKeys`	TRUE to include private keys in exported key sets

Description

Indicate whether or not private keys should be included when exporting key sets.

PGPOKeyGenFast

```
PGPOptionListRef PGPOKeyGenFast(
                      PGPContextRef      pgpContext,
                      PGPBoolean         fastGen );
```

Arguments

`pgpContext`	the target context
`fastGen`	TRUE to enable “fast” key generation mode

Description

Indicate whether or not keys should be generated in “fast” mode, that is, based on “known” primes instead of dynamically generated primes.

PGPOKeyGenParams

```
PGPOptionListRef PGPOKeyGenParams(
                      PGPContextRef      pgpContext,
                      PGPPublicKeyAlgorithm
                                         pubKeyAlg,
                      PGPUInt32          keySize );
```

Arguments

`pgpContext`	the target context
`pubKeyAlg`	the desired public key algorithm, which recognizes `kPGPPublicKeyAlgorithm_…` values (see Table 3-5)
`keySize`	the desired key size (in bits), which must be at least 512

Description

Establishes the public key algorithm and key size (in bits) to be used when generating keys or sub-keys, as well as when determining the entropy required to generate such keys or sub-keys.

Notes, Warnings, and Tips

The permissible key size values depend upon the choice of algorithm.

This option is required by those functions which accept it.

PGPOExpiration

```
PGPOptionListRef PGPOExpiration(
                        PGPContextRef      pgpContext,
                        PGPUInt32          expirationDays );
```

Arguments

`pgpContext`	the target context
`expirationDays`	the desired expiration date, expressed as days from “now”

Description

Sets the expiration date of keys and their component items generated for the specified context. Whenever a key or component is actually generated, the PGPsdk adds the specified number of days to the current system time, which establishes the key’s expiration date.

Notes, Warnings, and Tips

To ensure that a key or component item has no expiration date, specify `expirationDays` as having the special value `kPGPExpirationTime_Never`.

PGPOExportable

```
PGPOptionListRef PGPOExportable(
                        PGPContextRef      pgpContext,
                        PGPBoolean         canExport );
```

Arguments

`pgpContext`	the target context
`canExport`	`TRUE` if the item is exportable

Description

Indicate whether or not export of the key item in question is allowed. Currently, this only applies to signatures (see `PGPSignUserID`).

PGPOSigRegularExpression

```
PGPOptionListRef PGPOSigRegularExpression(
                        PGPContextRef      pgpContext,
                        char const         *regExpr );
```

Arguments

`pgpContext`	the target context
`regExpr`	the regular expression string

Description

Establishes the specified regular expression for use by `PGPSignUserID`.

Notes, Warnings, and Tips

This option is valid for `PGPSignUserID` only.

PGPOSigTrust

```
PGPOptionListRef PGPOSigTrust(
                        PGPContextRef       pgpContext,
                        PGPUInt32           trustLevel,
                        PGPUInt32           validity );
```

Arguments

`pgpContext`	the target context
`trustLevel`	the desired trust level for signatures, which asumes `kPGPNameTrust_…` values (see Table 3-7b)
`validity`	the desired trust value for signatures, which asumes `kPGPValidity_…` values (see Table 3-8)

Description

Establishes the specified signature validity for use by `PGPSignUserID`.

Notes, Warnings, and Tips

This option is valid for `PGPSignUserID` only.

Misc. Option List Functions

PGPONullOption

```
PGPOptionListRef PGPONullOption( void );
```

Arguments

Description

Returns a special `PGPOptionListRef` that is always ignored.

Notes, Warnings, and Tips

While this function is useful for providing a placeholder or default value in dynamically constructed option lists, the same results can be achieved by assemblying the dynamic option list from modular, persistent lists.

Sample Code

```
switch(encryptToOption)
{
case kEncryptToKey:
     encryptToOptionRef = PGPOEncryptToKey( pgpContext,
                                            key );
     break;
case kEncryptToKeySet:
     encryptToOptionRef = PGPOEncryptToKeySet( pgpContext,
                                               keySet );
     break;
case kEncryptToUserID:
     encryptToOptionRef = PGPOEncryptToUserID( pgpContext,
                                               userID );
     break;
default:
     encryptToOptionRef = PGPONullOption ( void );
     break;
}
err = PGPOAppendOptionList( pgpContext,
                            baseOptionList,
                            encryptToOptionRef,
                            PGPOLastOption  ) );
```

PGPOCompression

```
PGPOptionListRef PGPOCompression(
                       PGPContextRef       pgpContext,
                       PGPBoolean          isCompressed );
```

Arguments

`pgpContext`	the target context
`isCompressed`	`TRUE` to indicate compress plain text before encrypting or signing

Description

Indicates whether or not the input plain text should be compressed prior to encrypting or signing in binary format.

Notes, Warnings, and Tips

This option should routinely be specified as `TRUE`, since prior compression will not only reduce the size of the resultant cipher text, but also will increase the strength of the cipher text in most cases. This increase in the strength is partially a result of the reduction in plain text character frequency, and partially a result of the reduction in the amount of resultant cipher text.

Strong cipher text is essentially immune to compression, since it has large numbers of distinct “characters” that rarely if ever form repeating sequences.

PGPOCommentString

```
PGPOptionListRef PGPOCommentString(
                       PGPContextRef       pgpContext,
                        char const          *commentString );
```

Arguments

`pgpContext`	the target context
`commentString`	the comment text

Description

Indicates that the specified comment string should be included in the message blocks.

PGPOVersionString

```
PGPOptionListRef PGPOVersionString(
                        PGPContextRef       pgpContext,
                        char const          *versionString );
```

Arguments

`pgpContext`	the target context
`versionString`	the desired version string

Description

Indicates that the specified version string should be included in the message blocks.

Sample Code

```
char                    versionString[256];
PGPOptionListRef        tmpOptListRef;

PGPGetSDKString( &versionString[ 0 ] );
tmpOptListRef = PGPOVersionString( pgpContext,
                                   &versionString );
```

PGPOPassphrase

```
PGPOptionListRef PGPOPassphrase(
                        PGPContextRef       pgpContext,
                        const char          *passphraseBuf );
```

Arguments

`pgpContext`	the target context
`passphraseBuf`	the passphrase string

Description

Specifies the passphrase to be used for signing, conventional encrypting, and decrypting.

Notes, Warnings, and Tips

For signing and conventional encryption, this option must be specified as a sub-option (see `PGPOSignWIthKey` and `PGPOConventionalEncrypt`).

PGPOPassphraseBuffer

```
PGPOptionListRef PGPOPassphraseBuffer(
                      PGPContextRef      pgpContext,
                      const void         *passphraseBuf,
                      PGPSize            passphraseLength );
```

Arguments

`pgpContext`	the target context
`passphraseBuf`	the passphrase data
`passphraseLength`	the length of the passphrase data

Description

Specifies the passphrase to be used for signing, conventional encrypting, and decrypting. This differs from `PGPOPassphrase` in that the passphrase data and length are arbitrary, rather than being constrained to a *C* language string.

Notes, Warnings, and Tips

For signing and conventional encryption, this option must be set as a sub-option (see `PGPOSignWIthKey` and `PGPOConventionalEncrypt`).

PGPOPreferredAlgorithms

```
PGPOptionListRef PGPOPreferredAlgorithms(
                      PGPContextRef      pgpContext,
                      PGPCipherAlgorithm const *cipherKeyAlg,
                      PGPUInt32          cipherKeyAlgCount );
```

Arguments

`pgpContext`	the target context
`cipherKeyAlg`	an array of the prefered symmetric cipher algorithms, which recognizes `kPGPCipherAlgorithm_`... values (see Table 3-4a)
`cipherKeyAlgCount`	the number of symmetric cipher algorithms in the ordered array

Description

Establishes the specified symmetric cipher algorithm(s) as the preferred algorithm(s) to use when generating keys and their sub-items, as well as when encrypting and signing. The order of the array determines the relative preferences, with the first element in the array being the most preferred algorithm.

Notes, Warnings, and Tips

The number of symmetric cipher algorithms in the ordered array must be between one and the number of available symmetric cipher algorithms (see `PGPCountSymmetricCiphers`).

No assumption is made regarding the actual availability of the symmetric cipher algorithm(s) listed in the array.

The actual choice of algorithm involves availability and acceptability considerations. This function simply adds a preference consideration.

Sample Code

```
PGPCipherAlgorithm      cipherKeyAlg[ 3 ] =
                        {
                              kPGPCipherAlgorithm_CAST5, /* Most preferred    */
                              kPGPCipherAlgorithm_3DES,
                              kPGPCipherAlgorithm_IDEA   /* Least preferred   */
                        };

prefCipherAlg = PGPOPreferredAlgorithms( PGPContext pgpContext,
                              &cipherKeyAlg[ 0 ],
                              ( sizeof( cipherKeyAlg ) /
                                sizeof( PGPCipherAlgorithm ) ) );
```

PGPOKeySetRef

```
PGPOptionListRef PGPOKeySetRef(
                           PGPContextRef       pgpContext,
                           PGPKeySetRef        keySet );
```

Arguments

`pgpContext`	the target context
`keySet`	the desired key set

Description

For signature **validation** and decryption operations, use the *key database associated with* the specified key set as the look-up source for signature and decryption keys.

For key generation operations, use the *key database associated with* the specified key set as the destination for newly generated keys.

This option is required by those functions accepting it.

Notes, Warnings, and Tips

The current implementation treats the specified key set as an indirect parameter that references a key database, rather than as an explicit destination.

The indirect nature of this interface is likely to change in a future version, and will almost certainly involve changes to the function’s semantics and usage.

PGPOSendNullEvents

```
PGPOptionListRef PGPOSendNullEvents(
                           PGPContextRef       pgpContext,
                           PGPTimeInterval     approxInterval );
```

Arguments

`pgpContext`	the target context
`approxInterval`	the desired time interval (in milliseconds) between event postings

Description

Post a null event at each specified interval. This interval is approximate, but is guaranteed never to be less than that specified.

Notes, Warnings, and Tips

These events provide a mechanism and a data source for implementing progress bars, as well as a window of opportunity to pause, modify, or terminate the job.

PGPOEventHandler

```
PGPOptionListRef PGPOEventHandler(
                      PGPContextRef       pgpContext,
                      PGPEventHandlerProcPtr
                                          eventHandler,
                      PGPUserValue        eventHandlerArg );
```

Arguments

`pgpContext`	the target context
`eventHandler`	the desired event handler
`eventHandlerArg`	the user-defined data to be passed as an argument to the event handler

Description

Establish the specified function as the user event handler.

Notes, Warnings, and Tips

For greatest flexibility, the PGPsdk developer should consider establishing `eventHandlerArg` as a pointer to a user-defined data type, for example a *C* `struct`.

Specify `eventHandlerArg` as `( PGPUserData )0` to indicate a dummy argument.

PGPOLastOption

```
PGPOptionListRef PGPOLastOption( void );
```

Arguments

Description

All functions having a variable number of arguments must include a special argument to indicate the end of the argument list. This function provides that argument, and *must* appear at the end of every variable argument list.

Chapter 5: Function Reference – Global Random Number Pool Management Functions

Introduction

Since the PGPsdk cryptographic functions require random numbers to operate correctly, the PGPsdk includes functions to manage a global pool of random numbers seeded from keystrokes and mouse movements. The SHA-1 hash function is used to distill entropy from incoming events and to spread it throughout the random pool.

The PGPsdk provides both cryptographically strong pseudo-random numbers as well as true random numbers based on external events. An internal fixed-size random pool holds random bits acquired from events passed in by the caller, and the PGPsdk estimates the entropy content (that is, the amount of true randomness) of the events, and tracks the total entropy available in the random pool at any time.

Random numbers are made available via an internal pseudo-random number generator (RNG) based on ANSI X9.17, and fed from the random pool. When there is sufficient entropy in the pool, the generator produces cryptographically strong true random numbers; when the entropy in the random pool is exhausted, the generator produces cryptographically strong pseudo-random numbers.

The **ANSI X9.17** -compliant PGPsdk random number package includes the following functionality:

- acquiring randomness from environmental events passed in by the application
- filling buffers with random data as requested
- tracking the number of true random bits available

The random number functions support the following arguments and features to control their actions:

- random seeding from keystrokes and mouse movements
- a cryptographically strong pseudo-random number generator based on ANSI X9.17
- saving of the random pool state in persistent storage with reload on library initialization
- soft degrade from true environmental random bits to cryptographically strong pseudo-random bits

Header Files

```
pgpRandomPool.h
```

Random Number Pool Management Functions

PGPGlobalRandomPoolAddKeystroke

```
PGPUInt32        PGPGlobalRandomPoolAddKeystroke(
                      PGPInt32            keyCode );
```

Arguments

`keyCode`	the key code of the captured keystroke value

Description

Augments the random number pool based upon the value of the captured keystroke. A non-zero return value indicates that the operation increased the entropy of the random number pool.

PGPGlobalRandomPoolAddMouse

```
PGPUInt32        PGPGlobalRandomPoolAddMouse(
                      PGPUInt32          x,
                      PGPUInt32          y );
```

Arguments

x	the mouse x-coordinate value
y	the mouse y-coordinate value

Description

Augments the random number pool based upon the values of the captured mouse coordinates. A non-zero return value indicates that the operation increased the entropy of the random number pool.

Entropy Estimation Functions

PGPGlobalRandomPoolGetEntropy

```
PGPUInt32        PGPGlobalRandomPoolGetEntropy( void );
```

Arguments

Description

Returns a measure of the current entropy of the global random number pool.

PGPGlobalRandomPoolGetMinimumEntropy

```
PGPUInt32        PGPGlobalRandomPoolGetMinimumEntropy( void );
```

Arguments

Description

Returns the minimum allowable entropy of the global random number pool that will support generation of random or cryptographically strong pseudo-random numbers.

PGPGlobalRandomPoolHasMinimumEntropy

```
PGPBoolean       PGPGlobalRandomPoolHasMinimumEntropy( void );
```

Arguments

Description

Returns TRUE if the current entropy of the global random number pool is sufficient to generate random or cryptographically strong pseudo-random numbers. This is a convenience function, and is the equivalent of:

```
if  ( PGPGlobalRandomPoolGetEntropy( void ) >=
      PGPGlobalRandomPoolGetMinimumEntropy( void ) )
{
      return( TRUE );
}
else
{
      return( FALSE );
}
```

PGPGlobalRandomPoolGetSize

```
PGPUInt32        PGPGlobalRandomPoolGetSize( void );
```

Arguments

Description

Returns the current size of the global random number pool in bytes.

Chapter 6: Function Reference - Utility Toolbox

Introduction

The PGPsdk includes miscellaneous utility functions that relate to multiple functional areas, such as:

- context creation and management
- memory management
- file specification
- preferences
- date/time
- error code to error string conversion

Header Files

```
pgpsdkPrefs.h
pgpUtilities.h
```

Constants and Data Structures

Table 6-1: Memory Management Option Values.

Memory Management Flag Constant
kPGPMemoryFlags_Clear

Table 6-2a: MacOS File Creator Values.

MacOS File Creator Constant	MacOS OSType Value
kPGPMacFileCreator_DecryptedBinary	????
kPGPMacFileCreator_DecryptedText	ttxt
kPGPMacFileCreator_Keys	pgpK
kPGPMacFileCreator_Tools	pgpM

Table 6-2b: MacOS File Type Values.

MacOS File Type Constant	MacOS OSType Value
kPGPMacFileTypeArmorFile	TEXT
kPGPMacFileTypeDecryptedBinary	BINA
kPGPMacFileTypeDecryptedText	TEXT
kPGPMacFileTypeDetachedSig	pgDS
kPGPMacFileTypeEncryptedData	pgEF
kPGPMacFileTypeExportedKeys	TEXT
kPGPMacFileTypePref	pref
kPGPMacFileTypePrivRing	pgRR
kPGPMacFileTypePubRing	pgPR
kPGPMacFileTypeRandomSeed	pgRS
kPGPMacFileTypeSignedData	pgSF

Table 6-3: Preference Selector Values.

Preference Selector Constant
kPGPsdkPref_DefaultKeyID

kPGPsdkPref_PrivateKeyring
kPGPsdkPref_PublicKeyring
kPGPsdkPref_RandomSeedFile

Figure 6-1: PGPNewContextStruct typedef.

```
typedef struct PGPNewContextStruct
{
                  /*
                  ** sizeofStruct must be initialized
                  ** to sizeof( PGPNewContextStruct )
                  */
    PGPUInt32                        sizeofStruct;
    PGPMemoryAllocationProc          allocProc;
    PGPMemoryReallocationProc        reallocProc;
    PGPMemoryDeallocationProc        deallocProc;
    PGPUserValue                     allocUserValue;
} PGPNewContextStruct;
```

Context Creation and Management Functions

PGPNewContext

```
PGPError         PGPNewContext(
                      PGPUInt32          clientAPIVersion,
                      PGPContextRef      *pgpContext );
```

Arguments

`clientAPIVersion`	the version of the current PGPsdk client API
`pgpContext`	the receiving field for the new context

Description

Creates a context that employs the default PGPsdk memory management functions.

Errors

```
kPGPError_IncompatibleAPI
kPGPError_InvalidRef
```

Notes, Warnings, and Tips

`clientAPIVersion` should always be specified as the special value `kPGPsdkVersion`.

PGPNewContextCustom

```
PGPError         PGPNewContextCustom(
                      PGPUInt32          clientAPIVersion,
                      PGPNewContextStruct const
                                         *pgpContextStruct,
                      PGPContextRef      *pgpContext );
```

Arguments

`clientAPIVersion`	the version of the current PGPsdk client API
`pgpContextStruct`	the custom context information
`pgpContext`	the receiving field for the new context

Description

Creates a `PGPContext` that employs user-defined memory management functions. The custom information is passed as a `PGPNewContextStruct` (see Table 7-4).

Errors

```
kPGPError_IncompatibleAPI
kPGPError_InvalidRef
```

Notes, Warnings, and Tips

`clientAPIVersion` should always be specified as the special value `kPGPsdkVersion`.

The `PGPNewContextStruct` member `sizeofStruct` must be specified as the special value `sizeof( PGPNewContextStruct )`.

The custom memory allocation function should expect to receive the following arguments in the following order:

```
PGPContextRef     context
PGPSize           allocationSize
PGPMemoryFlags    flags
PGPUserValue      userValue
```

If the `flags` argument is specified as `kPGPMemoryFlags_Clear`, then the function should initialize the resultant memory to zeroes.

The custom memory re-allocation function should expect to receive the following arguments in the following order:

```
PGPContextRef     context
void              **allocation
PGPSize           newAllocationSize
PGPMemoryFlags    flags
PGPUserValue      userValue
```

If the `flags` argument is specified as `kPGPMemoryFlags_Clear`, then the function should initialize the resultant memory to zeroes.

The custom memory de-allocation function should expect to receive the following arguments in the following order:

```
PGPContextRef     context
void              *allocation
PGPUserValue      userValue
```

PGPFreeContext

```
PGPError          PGPFreeContext(
                        PGPContextRef       pgpContext );
```

Arguments

`pgpContext`	the target context

Description

Decrements the reference count for the specified context (created by either `PGPNewContext` or `PGPNewContextCustom`), and frees the context if the reference count reaches zero.

Notes, Warnings, and Tips

A `PGPContext` must *not* be freed until and unless all data items allocated using that context have been explicitly freed.

PGPSetContextUserValue

```
PGPError         PGPSetContextUserValue(
                      PGPContextRef       pgpContext,
                      PGPUserValue        userValue );
```

Arguments

`pgpContext`	the target context
`userValue`	the associated (replacement) user-defined data

Description

Sets the user-defined data associated with the specified context to that specified by `userValue`.

PGPGetContextUserValue

```
PGPError         PGPGetContextUserValue(
                      PGPContextRef       pgpContext,
                      PGPUserValue        *userValue );
```

Arguments

`pgpContext`	the target context
`userValue`	the receiving field for the associated user-defined data

Description

Retrieves the user-defined data associated with the specified context.

PGPContextGetRandomBytes

```
PGPError         PGPContextGetRandomBytes(
                      PGPContextRef       pgpContext,
                      void                *dataBuf,
                      PGPSize             availLength );
```

Arguments

`pgpContext`	the target context
`dataBuf`	the receiving buffer for the associated pseudo-random bytes
`availLength`	the length of the receiving buffer

Description

Places the pseudo-random bytes associated with the specified context into the specified buffer. A maximum of `availLength` bytes is retrieved. The function returns `kPGPError_OutOfEntropy` if the specified context's global random pool does not have sufficient entropy.

Errors

`kPGPError_OutOfEntropy`

Notes, Warnings, and Tips

The size of the global random pool and its entropy are independent of one another.

Memory Management Functions

PGPNewData

```
void            *PGPNewData(
                     PGPContextRef       pgpContext,
                     PGPSize             allocationSize );
```

Arguments

`pgpContext`	the target context
`allocationSize`	the number of 8-bits bytes to be allocated

Description

Allocates the specified number of 8-bit bytes of memory, using the memory allocation function associated with the specified context.

Notes, Warnings, and Tips

`PGPNewData` is used internally by the PGPsdk `PGPNew…` functions. Client code should rarely, if ever, have a reason to use this function.

Memory allocated with `PGPNewData` should always be deallocated with `PGPFreeData`.

A return value of `( void * )NULL` indicates failure.

PGPNewSecureData

```
void            *PGPNewSecureData(
                     PGPContextRef       pgpContext,
                     PGPSize             allocationSize,
                     PGPBoolean          *didLock );
```

Arguments

`pgpContext` the target context

`allocationSize` the number of 8-bit bytes to be allocated

`didLock` set to `TRUE` upon return if the memory allocated is guaranteed not to be swapped to secondary storage (virtual memory implementations)

Description

Allocates the specified number of 8-bit bytes of memory, using the memory allocation function associated with the specified context. The allocated memory is intended to store sensitive data such as passphrases, and so:

- the function attempts to preclude the allocated memory being swapped to secondary storage, thus simplifying later clearing of the memory
- memory allocated with this function is automatically cleared just prior to its deallocation

Notes, Warnings, and Tips

Memory allocated with `PGPNewSecureData` should always be deallocated with `PGPFreeData`.

A return value of `( void * )NULL` indicates failure.

Not all platforms support page locking, and those that do may restrict it to certain classes of users, for example, the superuser. Still, the PGPsdk will utilize whatever facilities do exist for the platform, and will ensure erasure of the resident memory upon deallocation.

PGPFreeData

```
void            PGPFreeData(
                     void                 *allocation );
```

Arguments

`allocation` the target data in memory

Description

Frees memory allocated with `PGPNewData` and `PGPNewSecureData`. Memory allocated with `PGPNewSecureData` is cleared prior to its being freed.

Notes, Warnings, and Tips

The operation will fail silently if `allocation` is `NULL`, or if the associated internal header control block is corrupted.

File Specification Functions

PGPNewFileSpecFromFSSpec (*MacOS platforms only*)

```
PGPError        PGPNewFileSpecFromFSSpec(
                     PGPContextRef        pgpContext,
                     const FSSpec         *spec,
                     PGPFileSpecRef       *fileRef );
```

Arguments

`pgpContext`	the target context
`spec`	the source Macintosh FS specification
`fileRef`	the receiving field for the resultant file specification

Description

Creates a file specification from the specified Macintosh FS specification.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant file specification with `PGPFreeFileSpec`.

PGPNewFileSpecFromFullPath (*Non-MacOS platforms only*)

```
PGPError         PGPNewFileSpecFromFullPath(
                      PGPContextRef       pgpContext,
                      char const          *pathname,
                      PGPFileSpecRef      *fileRef );
```

Arguments

`pgpContext`	the target context
`pathname`	the source pathname
`fileRef`	the receiving field for the resultant file specification

Description

Creates a file specification from a pathname.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant file specification with `PGPFreeFileSpec`.

PGPCopyFileSpec

```
PGPError         PGPCopyFileSpec(
                      PGPConstFileSpecRef
                                          fileSpecOrig,
                      PGPFileSpecRef      *fileSpecCopy );
```

Arguments

`fileSpecOrig`	the source file specification
`fileSpecCopy`	the receiving field for the copy of the file specification

Description

Creates an exact copy of the source file specification.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant file specification copy with `PGPFreeFileSpec`.

PGPFreeFileSpec

```
PGPError         PGPFreeFileSpec(
                      PGPFileSpecRef     fileSpecRef );
```

Arguments

`fileSpecRef`	the target file specification

Description

Decrements the reference count for the specified file specification, and frees the file specification if the reference count reaches zero.

PGPGetFSSpecFromFileSpec (*MacOS platforms only*)

```
PGPError         PGPGetFSSpecFromFileSpec(
                      PGPConstFileSpecRef
                                         fileSpec,
                      FSSpec             *fsSpec );
```

Arguments

`fileSpec`	the source file specification
`fsSpec`	the receiving field for the resultant Macintosh FS specification

Description

Converts the specified file specification to a Macintosh FS specification.

Errors

`kPGPError_FileNotFound`

PGPGetFullPathFromFileSpec (*Non-MacOS platforms only*)

```
PGPError         PGPGetFullPathFromFileSpec(
                      PGPConstFileSpecRef
                                         fileSpec,
                      char               **fullPathPtr );
```

Arguments

`fileSpec`	the target file specification
`fullPathPtr`	the receiving field for a pointer to the resultant full pathname

Description

Converts the specified file specification to a file pathname, and places it into dynamically allocated memory.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant pathname with `PGPFreeData`.

PGPMacBinaryToLocal **(*MacOS platforms only*)**

```
PGPError         PGPMacBinaryToLocal(
                      PGPFileSpecRef      inSpec,
                      PGPFileSpecRef      *outSpec,
                      PGPUInt32           *macCreator,
                      PGPUInt32           *macTypeCode );
```

Arguments

`inSpec`	the source file specification, which is assumed to reference a MacOS MacBinary file
`outSpec`	the receiving field for the file specification to the converted file
`macCreator`	the receiving field for the MacOS `OSType` of the creating application, which is always one of the `kPGPMacFileCreator_`… values (see Table 6-2a)
`macType`	the receiving field for the MacOS `OSType` of the file type, which is always one of the `kPGPMacFileType_`… values (see Table 6-2b)

Description

Converts a MacOS MacBinary file to files containing its data fork and resource fork. The source file is deleted upon conversion.

A return value of `kPGPError_NoMacBinaryTranslationAvailable` indicates that while the conversion did suceed and that the source file was deleted, either:

- the `macCreator` and/or `macType` values were not recognized, and so the file suffix was defaulted to `.bin`
- the source file had no data fork

A return value of `kPGPError_NotMacBinary` indicates that the source file specification does not reference a MacOS MacBinary file. The source file is unaltered.

Errors

```
kPGPError_NoMacBinaryTranslationAvailable
kPGPError_NotMacBinary
```

Notes, Warnings, and Tips

The `macCreator` and `macType` arguments are optional. If specified as `NULL`, then the corresponding data item is not returned.

No assumption should be made regarding the name of the resultant file. The PGPsdk chooses the most appropriate extension for the encoded file type.

Preference Functions

PGPsdkInit

```
PGPError         PGPsdkInit( void );
```

Arguments

Description

Initializes the PGPsdk global state. This function must be called before using using any part of the PGPsdk.

Notes, Warnings, and Tips

Multiple calls to this function will *not* re-initialize the global variables. Instead, a mechanism similar to the opaque data type reference count mechanism tracks the calls. This frees the PGPsdk developer from having to worry about whether or not the global state has already been initialized, since a subsequent initialization will not adversely affect the global state.

The caller is responsible for freeing any and all resources held by the PGPsdk with `PGPsdkCleanup`.

Calling this function is redundant for Windows and MacOS platforms, since it is called by the PGPsdk library initial entry point.

PGPsdkCleanup

```
PGPError         PGPsdkCleanup( void );
```

Arguments

Description

Releases any and all resources held by the PGPsdk.

Notes, Warnings, and Tips

This function should be called only after freeing the last `PGPContext`. Any subsequent usage of the PGPsdk must first call `PGPsdkInit`.

Calling this function is redundant for Windows and MacOS platforms, since the PGPsdk library automatically calls this function upon exit.

PGPsdkLoadDefaultPrefs

```
PGPError         PGPsdkLoadDefaultPrefs(
                      PGPContextRef       pgpContext );
```

Arguments

`pgpContext`	the target context

Description

Loads the preferences from the default preference file.

PGPsdkLoadPrefs

```
PGPError         PGPsdkLoadPrefs(
                      PGPContextRef       pgpContext,
                         PGPFileSpecRef     prefSpec );
```

Arguments

`pgpContext`	the target context
`prefSpec`	the file containing the stored preferences

Description

Loads the preferences from the specified preference file.

PGPsdkSavePrefs

```
PGPError         PGPsdkSavePrefs(
                     PGPContextRef      pgpContext );
```

Arguments

`pgpContext`	the target context

Description

Saves any changed preference to its associated source file.

Notes, Warnings, and Tips

The `PGPContext` “remembers” the source file from which each preference was loaded, and so the preference information is saved to that file.

PGPsdkPrefSetData

```
PGPError         PGPsdkPrefSetData(
                     PGPContextRef      pgpContext,
                     PGPsdkPrefSelector prefSelector,
                     void const         *prefBuf,
                     PGPSize            prefLength );
```

Arguments

`pgpContext`	the target context
`prefSelector`	the target preference, which recognizes `kPGPsdkPref_`... values (see Table 6-3)
`prefBuf`	the associated (replacement) preference data
`prefLength`	the length of the associated (replacement) preference data

Description

Sets the data associated with the specified preference to the specified (replacement) preference data.

Notes, Warnings, and Tips

The caller must additionally call `PGPsdkSavePrefs` to make the change permanent.

PGPsdkPrefSetFileSpec

```
PGPError         PGPsdkPrefSetFileSpec(
                      PGPContextRef      pgpContext,
                      PGPsdkPrefSelector prefSelector,
                      PGPConstFileSpec   fileSpec );
```

Arguments

`pgpContext`	the target context
`prefSelector`	the target preference, which recognizes `kPGPsdkPref_`... values (see Table 6-3)
`fileSpec`	the (replacement) file specification

Description

Establishes the specified file as the persistent store for the specified preference.

Notes, Warnings, and Tips

The caller must additionally call `PGPsdkSavePrefs` to make the change permanent.

PGPsdkPrefGetData

```
PGPError         PGPsdkPrefGetData(
                      PGPContextRef      pgpContext,
                      PGPsdkPrefSelector prefSelector,
                      void               **prefBuf,
                      PGPSize            *prefLength );
```

Arguments

`pgpContext`	the target context
`prefSelector`	the target preference, which recognizes `kPGPsdkPref_`... values (see Table 6-3)
`prefBuf`	the receiving field for a pointer to the requested preference data
`prefLength`	the receiving field for the resultant length of the requested preference data

Description

Retrieves the data associated with the specified preference into dynamically allocated memory.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant preference data with `PGPFreeData`.

PGPsdkPrefGetFileSpec

```
PGPError         PGPsdkPrefGetFileSpec(
                      PGPContextRef      pgpContext,
                      PGPsdkPrefSelector prefSelector,
                      PGPFileSpecRef     *fileSpec );
```

Arguments

`pgpContext`	the target context
`prefSelector`	the target preference, which recognizes `kPGPsdkPref_`... values (see Table 6-3)
`fileSpec`	the receiving field for the associated file specification

Description

Retrieves the file specification associated with the specified preference.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant file specification with `PGPFreeFileSpec`.

Date/Time Functions

PGPGetTime

```
PGPTime         PGPGetTime( void );
```

Arguments

Description

Returns the current system time as a `PGPTime` format time value.

PGPGetPGPTimeFromStdTime

```
PGPTime         PGPGetPGPTimeFromStdTime(
                     time_t             theTime );
```

Arguments

`theTime`	the time in Standard *C* Library time format

Description

Returns the specified time as a `PGPTime` format time value.

Notes, Warnings, and Tips

The data type `time_t` is that used by many of the Standard *C* Library time functions, for example `time()`.

PGPGetStdTimeFromPGPTime

```
time_t          PGPGetStdTimeFromPGPTime(
                     PGPTime            theTime );
```

Arguments

`theTime`	the time as a `PGPTime` data type

Description

Returns the specified `PGPTime` value as a `time_t` format time value.

Notes, Warnings, and Tips

The data type `time_t` is that used by many of the Standard *C* Library time functions, for example `time()`.

PGPGetYMDFromPGPTime

```
void            PGPGetYMDFromPGPTime(
                     PGPTime             theTime,
                     PGPUInt16           *year,
                     PGPUInt16           *month,
                     PGPUInt16           *day );
```

Arguments

`theTime`	the time as a `PGPTime` data type
`year`	the receiving field for the year component
`month`	the receiving field for the month component
`day`	the receiving field for the day component

Description

Extracts the year, month, and day components from the specified `PGPTime` time value.

Notes, Warnings, and Tips

The `year`, `month`, and `day` arguments are optional. If specified as `NULL`, then the corresponding data item is not returned.

The `year` component includes the century.

The `month` and `day` components are one-based.

Sample Code

```
PGPUInt16       year;       /* Includes century           */
PGPUInt16       month;      /* January = 1; December = 12 */
PGPUInt16       day;        /* Assumes values 1 - 31      */

/*
** Output the current date as YYYY.MM.DD
*/
PGPGetYMDFromPGPTime( PGPGetTime( void ),
                      &year,
                      &month,
                      &day );

printf(“%.4d.%.2d.%.2d\n”,
            year,
            month,
            day );
```

PGPTimeFromMacTime (*MacOS platforms only*)

```
PGPTime         PGPTimeFromMacTime(
                     PGPUInt32          theTime );
```

Arguments

`theTime` the time as a MacOS format time value

Description

Returns the specified MacOS format time value as a `PGPTime` format time value.

Sample Code

```
PGPUInt32  macTime;

err = PGPNewKeyCreationTimeFilter( pgpContext,
                                   PGPTimeFromMacTime( macTime ),
                                   kPGPMatchLessOrEqual,
                                   *outFilter );
```

PGPTimeToMacTime (*MacOS platforms only*)

```
PGPUInt32       PGPTimeToMacTime(
                     PGPTime            theTime );
```

Arguments

`theTime` the time as a `PGPTime` format time value

Description

Returns the specified `PGPTime` format `time` value as a MacOS format time value.

Sample Code

```
PGPUInt32  macTime;

macTime = PGPTimeToMacTime( PGPGetTime ( void ) );
```

Error Look-Up Functions

PGPGetErrorString

```
PGPError        PGPGetErrorString(
                     PGPError           theErrorCode,
                     PGPsize            availLength,
                     char               *theErrorText );
```

Arguments

`theErrorCode`	the encoded error value
`availLength`	the available length of the receiving buffer
`theErrorText`	the receiving buffer

Description

Looks-up the encoded error value, and places the corresponding error text formated as a *C* language string into the receiving buffer .

Errors

`kPGPError_BufferTooSmall`

Notes, Warnings, and Tips

The error text is truncated as required, and results in `kPGPError_BufferTooSmall` being returned.

Chapter 7: Function Reference - Feature (Capability) Query Functions

Introduction

When one considers the present state of U.S. export law and the continuously evolving set of cryptographic standards, algorithms, and formats, the simultaneous existence of multiple versions of the PGPsdk becomes a very real possibility. For example, one version of the PGPsdk library may support encryption, while another supports signing but not encryption. By including functions that return version numbers and the availability of specific features (capabilities), the PGPsdk provides applications with a measure of version independence, as well as a specific and extensible mechanism for determining feature availability.

The feature query functions that allow the caller to determine the availablity of a specific feature before attempting to use it are the only supported means for determining such availability. The PGPsdk version number should *not* be used to determine feature availability. As the PGPsdk library evolves and adopts a more customized, modular build model that may include “stub” functions that do nothing except return an appropriate error code, the presence and use of these feature query functions can only increase in importance.

Header Files

```
pgpFeatures.h
```

Constants and Data Structures

Table 7-1: `PGPGetFeatureFlags` Feature (Capability) Selector Values.

Feature (Capability) Selector Constants
kPGPFeatures_GeneralSelector
kPGPFeatures_ImplementationSelector

Table 7-2: `kPGPFeatures_GeneralSelector` Mask Values.

General Selector Mask Constants
kPGPFeatureMask_CanDecrypt
kPGPFeatureMask_CanEncrypt
kPGPFeatureMask_CanSign
kPGPFeatureMask_CanVerify

Table 7-3: `kPGPFeatures_GeneralSelector` Mask Values.

Implementation Selector Mask Constants
kPGPFeatureMask_HasTimeout
kPGPFeatureMask_IsDebugBuild

Figure 7-1: `PGPAlgorithmInfo typedef`.

```
typedef struct PGPAlgorithmInfo
{
     char                    shortName[ 32 ];
     char                    longName[ 96 ];
     char                    copyright[ 128 ];
     PGPFlags                flags;
     PGPUInt32               reserved[ 16 ];
} PGPAlgorithmInfo;
```

Figure 7-2: `PGPPublicKeyAlgorithmInfo` typedef.

```
    typedef struct PGPPublicKeyAlgorithmInfo
    {
        PGPAlgorithmInfo        info;
        PGPPublicKeyAlgorithm   algID;
        PGPBoolean              canEncrypt;
        PGPBoolean              canSign;
        PGPBoolean              reserved1;
        PGPBoolean              reserved2;
        PGPUInt32               reserved[ 8 ];
    } PGPPublicKeyAlgorithmInfo;
```

Figure 7-3: `PGPSymmetricCipherInfo` typedef.

```
    typedef struct PGPSymmetricCipherInfo
    {
        PGPAlgorithmInfo        info;
        PGPCipherAlgorithm      algID;
        PGPUInt32               reserved[ 8 ];
    } PGPSymmetricCipherInfo;
```

Feature (Capability) Query Functions

PGPGetFeatureFlags

```
PGPError        PGPGetFeatureFlags(
                     PGPFeatureSelector featureSelector,
                     PGPFlags           *featureFlags );
```

Arguments

`featureSelector`	the feature flags to obtain, which recognizes `kPGPFeatures_…Selector` values (see Table 7-1)
`featureFlags`	the receiving field for the feature flags

Description

Retrieves the flags associated with the specified feature selector. A return value of `kPGPError_ItemNotFound` indicates that the `featureSelector` value is not recognized.

Errors

```
kPGPError_ItemNotFound
```

Notes, Warnings, and Tips

Since `flags` is an encoded value, individual features should always be extracted by presenting the `PGPFeatureExists` macro (defined in `pgpFeatures.h`) with the appropriate `kPGPFeatureMask_`... value (see Table 7-2 and Table 7-3).

Sample Code

```
PGPFlags        featureFlags;

if ( ( PGPFeatureExists( featureFlags,
                         ( kPGPFeatureMask_CanSign |
                           kPGPFeatureMask_CanEncrypt ) ) ) );
{
    /* features-are-available code */
}
```

PGPCountPublicKeyAlgorithms

```
PGPError         PGPCountPublicKeyAlgorithms(
                      PGPUInt32           *numPKAlgs );
```

Arguments

`numPKAlgs`	the receiving field for the number of available public key algorithms

Description

Provides the number of available public key algorithms.

Notes, Warnings, and Tips

Use this count as the exclusive upper limit when indexing through the available algorithms (see the sample code for `PGPGetIndexedPublicKeyAlgorithmInfo`).

PGPGetIndexedPublicKeyAlgorithmInfo

```
PGPError         PGPGetIndexedPublicKeyAlgorithmInfo(
                      PGPUInt32           index,
                      PGPPublicKeyAlgorithmInfo
                                          *info );
```

Arguments

`index`	the index (zero-based) of the desired public key algorithm
`info`	the receiving field for the associated algorithm information

Description

Provides a means of indexing through the available public key algorithms and accessing their associated information, which is of type `PGPPublicKeyAlgorithmInfo` (see Figure 7-1 and Figure 7-2).

Sample Code

```
PGPError                    err;
PGPUInt32                   index;
PGPUInt32                   numPKAlgs;
PGPPublicKeyAlgorithm       targetPKAlg;
PGPPublicKeyAlgorithmInfo   info;

if  ( IsPGPError( err = PGPCountPublicKeyAlgorithms( &numPKAlgs ) ) )
{
    return( err );
}
targetPKAlg = kPGPPublicKeyAlgorithm_ElGamal;
for ( index = 0; index < numPKAlgs; index++ )
{
    if  ( IsPGPError( err = PGPGetIndexedPublicKeyAlgorithmInfo( index, &info ) ) )
    {
          return( err );
    }
    if  ( info.algID == targetPKAlg )
    {
          break;
    }
}

if  ( index >= numPKAlgs )
{
    return( kPGPError_UnknownPublicKeyAlgorithm );
}

return( kPGPError_NoErr );
```

PGPCountSymmetricCiphers

```
PGPError          PGPCountSymmetricCiphers(
                       PGPUInt32           *numSymmetricCiphers );
```

Arguments

`numSymmetricCiphers`

the receiving field for the number of available symmetric ciphers

Description

Provides the number of available symmetric ciphers.

Notes, Warnings, and Tips

Use this count as the exclusive upper limit when indexing through the available symmetric ciphers (see the sample code for `PGPGetIndexedSymmetricCipherInfo`).

PGPGetIndexedSymmetricCipherInfo

```
PGPError          PGPGetIndexedSymmetricCipherInfo(
                       PGPUInt32           index,
                       PGPSymmetricCipherInfo
                                           *info );
```

Arguments

`index`	the index (zero-based) of the desired symmetric cipher
`info`	the receiving field for the associated information

Description

Provides a means of indexing through the available symmetric ciphers and accessing the associated information, which is of type `PGPSymmetricCipherInfo` (see Figure 7-1 and Figure 7-3).

Sample Code

```
PGPError                    err;
PGPUInt32                   index;
PGPUInt32                   numSymmetricCiphers;
PGPCipherAlgorithm          targetSymmetricCipher;
PGPSymmetricCipherInfo      info;

if  ( IsPGPError( err = PGPCountSymmetricCiphers( &numSymmetricCiphers ) ) )
{
    return( err );
}

targetSymmetricCipher = kPGPCipherAlgorithm_3DES;
for ( index = 0; index < numSymmetricCiphers; index++ )
{
    if  ( IsPGPError( err = PGPGetIndexedSymmetricCipherInfo( index, &info ) ) )
    {
          return( err );
    }
    if  ( info.algID == targetSymmetricCipher )
    (
          break;
    }
}

if  ( index >= numSymmetricCiphers )
{
    return( kPGPError_UnknownSymmetricCipher );
}
```

```
    return( kPGPError_NoErr );
```

PGPGetSDKVersion

```
void            PGPGetSDKVersion(
                     PGPUInt32          *version );
```

Arguments

`version`	the receiving field for the version number value

Description

Places the PGPsdk API version number into the referenced field. Since the version number is encoded, its components should always be extracted using the `PGPMajorVersion`, `PGPMinorVersion`, and `PGPRevVersion` macros defined in `pgpUtilities.h`.

Notes, Warnings, and Tips

The version number reflects the API version, and not the release version of the packaged software developer’s kit. Generally speaking, the API version is independent of the version of the PGPsdk.

Sample Code

```
PGPUInt32       completeVersionNumber;
char            versionString[ 256 ];

PGPGetSDKVersion( &completeVersionNumber );

sprintf(&versionString[ 0 ],
        “PGPsdk Version %d.%d.%d (c) 1997 Pretty Good Privacy, Inc.\n”,
        PGPMajorVersion( completeVersionNumber ),
        PGPMinorVersion( completeVersionNumber ),
        PGPRevVersion( completeVersionNumber ) );

printf(“%s\n”,
      &versionString[ 0 ] );
```

PGPGetSDKString

```
void            PGPGetSDKString(
                     char               theString[ 256 ] );
```

Arguments

`theString[ 256 ]`	a buffer having a minimum length of 256 bytes to receive the PGPsdk API version string

Description

A convenience function that yields a *C* language string of the form:

```
PGPsdk Version 1.1.0 (c) 1997 Pretty Good Privacy, Inc.
```

This function is the equivalent of the sample code included for `PGPGetSDKVersion`.

Sample Code

```
char      versionString[ 256 ];

PGPGetSDKString( &versionString[ 0 ] );
printf(“%s\n”,
      &versionString[ 0 ] );
```

Chapter 8: Function Reference – Key Server Functions

Introduction

The PGPsdk includes functions that support communication with HTTP and LDAP key servers, and allow developers to search for, add, disable, and delete keys on those servers.

Key server search operations support the same key filter mechanism described in Chapter 2, and yield a key set of the keys on the server that satisfy the filter criteria. LDAP servers essentially support the entire set of available primative filters; HTTP servers support a significantly more limited set (see Table 8-5).

Key server add, disable, and delete operations accept a key set that specifies input, and yield a resultant key set that contains the keys that could not be added, disabled, or deleted.

Key server search for, add, disable, and delete operations include a user-defined callback function. This function gains control periodically, and so allows the developer to look for a pending user cancel request, effect other processing as required, or perform whatever operations the developer wishes. It is important to note that the intent and functionality of this callback mechanism is quite different from that of the event handler mechanism provided for key generation and encrypt/decrypt operations. No event is sent and no event-specific data is included – the callback function simply assumes control and executes until it returns. If the callback function returns a value other than `kPGPError_NoErr`, then the associated key server operation is aborted.

Header Files

```
pgpKeyServer.h
pgpKeyServerTypes.h
```

Constants and Data Structures

Table 8-1: Key Server State Values.

Key Server State Constants
`kPGPKeyServerStateConnect`
`kPGPKeyServerStateDisconnect`
`kPGPKeyServerStateReceive`
`kPGPKeyServerStateSend`
`kPGPKeyServerStateWait`

Table 8-2: Key Server Query Completion **Values.**

Key Server Query Completion Constants
`kPGPKeyServerQuery_PartialResults`

Table 8-3: Key Server Space Values.

PGPKeyServerKeySpace
`kPGPKSKeySpaceNormal`

`kPGPKSKeySpacePending`
`kPGPKSKeySpaceDefault***`

*** `kPGPKSKeySpaceDefault` is currently equivalent to `kPGPKSKeySpaceNormal`.

Table 8-4: Key Server Access Values.

PGPKeyServerAccessType
`kPGPKSAccess_Normal`
`kPGPKSAccess_Administrator`
`kPGPKSAccess_Default`

*** `kPGPKSAccess_Default` is currently equivalent to `kPGPKSAccess_Normal`.

Table 8-5: Valid `PGPQueryKeyServer` Filters for Key Server Protocols.

Filter Function	HTTP	LDAP
PGPIntersectFilters		●
PGPNegateFilter		●
PGPNewKeyCreationTimeFilter		●
PGPNewKeyDisabledFilter		●
PGPNewKeyEncryptAlgorithmFilter		●
PGPNewKeyEncryptKeySizeFilter		●
PGPNewKeyExpirationTimeFilter		●
PGPNewKeyFingerPrintFilter		
PGPNewKeyIDFilter	●	●
PGPNewKeyRevokedFilter		●
PGPNewKeySigAlgorithmFilter		●
PGPNewKeySigKeySizeFilter		
PGPNewSigKeyIDFilter		●
PGPNewSubKeyIDFilter		●
PGPNewUserIDEmailFilter	●	●
PGPNewUserIDNameFilter	●	●
PGPNewUserIDStringFilter	●	●
PGPUnionFilters		●

Figure 8-1: `PGPKeyServerMonitor` typedef.

```
typedef struct PGPKeyServerMonitor
{
     PGPUInt32                   magic;
     char                        *monitorTag;
     char                        **monitorValues;
     struct PGPKeyServerMonitor  *next;
} PGPKeyServerMonitor;
```

Events and Callbacks

A number of the key server functions allow the calling application to request callbacks to track the progress of the request. These functions generally require a perceptible amount of execution time, regardless of the size of their target key set.

An event handler serves two purposes – it provides notification to the calling application that an event has occurred, and provides a mechanism for the calling application to affect processing (in a pre-defined manner). Notification includes a pointer to a `PGPEvent` data type that, depending on the type of event, provides detailed information about the cause of the event. The calling application can then respond appropriately, which may or may not intervene and affect the course of further processing. If the calling application wishes to intervene, then it can abort the request by returning an error code (a value other than kPGPError_NoErr).

All event handlers are declared as

```
PGPError myEvents( PGPContextRef pgpContext,
                   PGPEvent *event,
                   PGPUserValue userValue );
```

The pgpContext argument is the reference to the context of the function posting the event. The event argument references a PGPEvent data type as follows:

```
struct PGPEvent_
{
    PGPVersion              version;
    struct PGPEvent_        *nextEvent;
    PGPJobRef               job;
    PGPEventType            type;
    PGPEventData            data;
};
typedef struct PGPEvent_ PGPEvent;
```

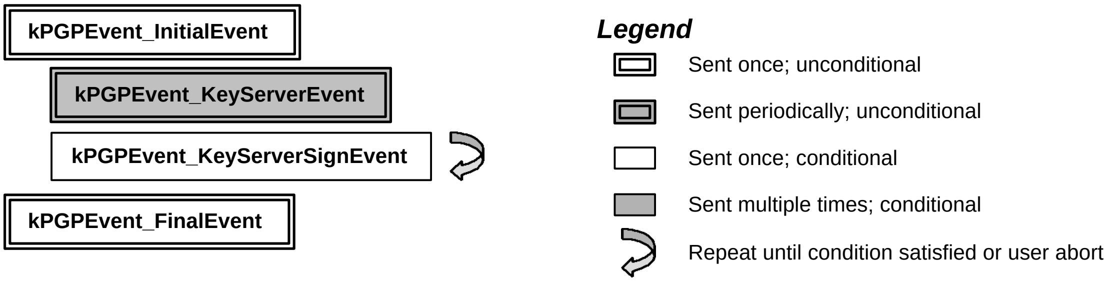
The version and nextEvent members are currently reserved for internal use. The job member is not applicable to key server functions. The type member identifies the event being posted, and recognizes kPGPEvent_… values (see Table 3-1). The data member is a union of the event-specific data structures, which are described with their corresponding event.

The calling application can modify the processing context by invoking PGPAddJobOptions as

```
PGPError  PGPAddJobOptions( PGPJobRef job, ... );
```

The value of the job argument is that of the PGPEvent argument's job member. Additional PGPOptionListRef arguments are specified similarly to the way they are passed to PGPEncode and PGPDecode. However, only certain options can be set after each type of event, and these are listed for each event.

Figure 8-2: Key Server Request Processing Event Sequence.


kPGPEvent_InitialEvent
kPGPEvent_KeyServerEvent
kPGPEvent_KeyServerSignEvent
kPGPEvent_FinalEvent
Legend
Sent once; unconditional
Sent periodically; unconditional
Sent once; conditional
Sent multiple times; conditional
Repeat until condition satisfied or user abort

Key Server Request Events

kPGPEvent_InitialEvent

Sent before all other events. Implies initiation of the key server request.

Data

None

Options

None

kPGPEvent_KeyServerEvent

Similar to `kPGPEvent_NullEvent,` this event reports the progress of the key server request, and allows the PGPsdk developer to determine its completion percentage.

The `state` member indicates the current point in the key server request processing from the caller's point of view, and assumes `kPGPKeyServerState`… values (see Table 8-1).

The `soFar` and `total` members should be treated as relative, unscaled quantities – they are not necessarily byte or number-of-keys quantities. In all cases, the completion percentage is calculated as follows:

```
double      completionPercent;
PGPUInt32   processState;

if  ( event->type == kPGPEvent_KeyServerEvent )
{
     if  ( event->keyServerData.total != 0 )
     {
      completionPercent = ( 100 * event->keyServerData.soFar ) /
                         event->keyServerData.total;
     }
     else
     {
      completionPercent = 100;
     }
     processState = event->keyServerData.state;
}
```

Data

```
typedef struct PGPEventKeyServerData_
{
    PGPUInt32         state;
    PGPUInt32         soFar;
    PGPUInt32         total;
} PGPEventKeyServerData;
```

Options

None

kPGPEvent_KeyServerSignEvent

Sent if a signing key is needed for authentication (posted by`PGPUploadToKeyserver`, `PGPDeleteFromKeyserver`, and `PGPDisableFromKeyserver`) to ensure that the requestor is authorized to effect the operation on the current qualifying key. The event handler should invoke `PGPAddJobOptions` specifying the `PGPOSignWithKey` and the `PGPOClearSign` options, or return `kPGPError_UserAbort`. Note that `PGPOSignWithKey` further requires either the `PGPOPassphrase` or the `PGPOPassphraseBuffer` option:

```
if  ( event->type == kPGPEvent_KeyServerEvent )
    {
            char  passPhraseBuf[ 256 ];

            /*
            ** For operations on multiple keys, the passphrase function should
            ** return a copy of the passphrase from a secure cache.  Otherwise,
            ** the user would be required to enter a passphrase for each key,
            ** instead of just for each operation.
            */
            myGetPassphrase( &passPhraseBuf[ 0 ] );
            err = PGPAddJobOption( event->job,
                                   PGPOSignWithKey(
                                          pgpContext,
                                          sigKey,
                                          PGPOPassphrase( pgpContext,
                                                          &passPhraseBuf[ 0 ]),
                                                          PGPOLastOption( void ) ),
                                    PGPOClearSign( pgpContext,
                                                   TRUE ),
                                    PGPOLastOption( void ) );

          memset( &passPhraseBuf[ 0 ],
                 \0,
                 256 );
     }
```

This event is sent repeatedly until a valid signing key is received, or until the event handler requests abort of the job. This allows the event handler to enforce a limit on the number of passphrase attempts.

The `state` member indicates the current point in the key server request processing from the caller's point of view, and assumes `kPGPKeyServerState`… values (see Table 8-1). It is not particularly useful in this context.

Data

```
typedef struct PGPEventKeyServerSignData_
{
    PGPUInt32         state;
} PGPEventKeyServerSignData;
```

Options

`PGPOSignWithKey`
`PGPOClearSign`

kPGPEvent_FinalEvent

Sent after all other events. Implies completion of the key server request.

Data

None

Options

None

Key Server Functions

PGPKeyServerInit

```
PGPError         PGPKeyServerInit( void );
```

Arguments

Description

Initializes the underlying communcations layer that the PGPsdk requires to access a key server. This function effectively creates a communications session, and must be called prior to calling any other key server function.

Errors

`kPGPError_UnknownError`

PGPNewKeyServerFromURL

```
PGPError         PGPNewKeyServerFromURL(
                      PGPContextRef       pgpContext,
                      char const          *url,
                      PGPKeyServerAccessType
                                          accessType,
                      PGPKeyServerKeySpace
                                          keySpace,
                      PGPKeyServerRef     *keyServer );
```

Arguments

`pgpContext`	the target context
`url`	the destination URL, which is of the form `[[protocol:]//]host.domain[:port]`
`accessType`	recognizes `kPGPKSKeySpace_`… values
`keySpace`	recognizes `kPGPKSKeyAccess_`… values
`keyServer`	the receiving field for the resultant key server communication context

Description

Creates a new HTTP or LDAP communication context, depending on the specified URL.

The `accessType` argument is advisory only, that is, no initial authorization validation occurs. However, it must reflect `kPGPKSAccess_Administrator` if the caller intends to later invoke any of the following functions:

- `PGPLDAPNewServerMonitor`
- `PGPUploadToKeyserver`
- `PGPDeleteFromKeyserver`
- `PGPDisableFromKeyserver`

The `keySpace` argument indicates which keys may be acted upon by the following functions:

- `PGPQueryFromKeyserver`

- `PGPDeleteFromKeyserver`
- `PGPDisableFromKeyserver`

A value of `kPGPKSKeySpaceNormal` restricts the above functions to those keys which meet the target keyserver's policy requirements, while a value of `kPGPKSKeySpacePending` restricts the above functions to those keys which do not.

Errors

`kPGPError_ServerInvalidProtocol`

Notes, Tips, and Warnings

If the `protocol:` portion of the destination URL is omitted, then an HTTP context is assumed.

If the `:port` portion of the destination URL is omitted, then an appropriate HTTP or LDAP port number is assumed.

PGPLDAPNewServerMonitor

```
PGPError          PGPLDAPNewServerMonitor(
                       PGPKeyServerRef    keyServer,
                       PGPEventHandlerProcPtr
                                          callBack,
                       PGPUserValue       callBackArg,
                       PGPKeyServerMonitorRef
                                          *dataAndStats );
```

Arguments

`keyServer`	the target key server
`callBack`	the desired callback function or `( PGPEventHandlerProcPtr )NULL` to indicate no callbacks.
`callBackArg`	the user-defined data, to be passed as an argument to the callback function (meaningful only in conjunction with `callBack`)
`dataAndStats`	the receiving field for the resultant key server data and statistics

Description

Creates a new key server monitor that contains relevant data about and statistics for the specified LDAP key server. The resultant data and statistics are contained in a linked list of `PGPKeyServerMonitor` datatypes, which contain name/value pairs where a pair may have multiple values.

Depending upon the policies established for the target key server, this function may generate a `kPGPEvent_KeyServerSignEvent`. In this case, a valid `callBack` argument is required, or the function will fail with `kPGPError_ServerAuthorizationRequired`.

Errors

`kPGPError_ServerOperationNotAllowed`
`kPGPError_ServerAuthorizationRequired`

Notes, Tips, and Warnings

Calling this function for an HTTP key server will result in the return of `kPGPError_ServerOperationNotAllowed`.

Specify `callBackArg` as `( PGPUserData )0` to indicate a dummy argument.

The caller is responsible for freeing the resultant LDAP server monitor with `PGPLDAPFreeServerMonitor`.

Sample Code

```
PGPKeyServerMonitor         thisMonitor;
PGPKeyServerMonitor         nextMonitor;
PGPKeyServerMonitorRef      dataAndStats;

err = PGPLDAPNewServerMonitor ( keyServer,
                               ( PGPEventHandlerProcPtr )NULL,
                               ( PGPUserValue )0,
                               &dataAndStats );
if  ( ( IsPGPError( err ) ) || ( dataAndStats == ( PGPKeyServerMonitor * )NULL ) )
{
    return;
}

/*
** Display the resultant data and statistics
*/
thisMonitor = dataAndStats;
while (thisMonitor != ( PGPKeyServerMonitor * )NULL )
{
    nextMonitor = thisMonitor->next;
    printf( “%s\n”,
            thisMonitor->monitorTag );
    while ( thisMonitor->monitorValues != NULL )
    {
          printf( “\t%s\n”,
                  thisMonitor->monitorValues++ );
    }
    thisMonitor = next_Monitor;
}

/*
** Free the resultant data and statistics
*/
thisMonitor = dataAndStats;
while (thisMonitor != ( PGPKeyServerMonitor * )NULL )
{
    nextMonitor = thisMonitor->next;
    err = PGPLDAPFreeServerMonitor( keyServer, thisMonitor );
    thisMonitor = next_Monitor;
}

return;
```

```
PGPError   MyEventHandler()
{

}
```

PGPLDAPFreeServerMonitor

```
PGPError          PGPLDAPFreeServerMonitor(
                       PGPKeyServerRef    keyServer,
                       PGPKeyServerMonitorRef
                                          dataAndStats );
```

Arguments

`keyServer`	the target key server
`dataAndStats`	the target key server data and statistics

Description

Decrements the reference count for the specified key server monitor, and frees the key server monitor if the reference count reaches zero.

PGPFreeKeyServer

```
PGPError          PGPFreeKeyServer(
                       PGPKeyServerRef    keyServer );
```

Arguments

`keyServer`	the target key server

Description

Decrements the reference count for the specified key server, and frees the key server if the reference count reaches zero.

PGPKeyServerOpen

```
PGPError          PGPKeyServerOpen(
                       PGPKeyServerRef    keyServer );
```

Arguments

`keyServer`	the target key server

Description

Explicitly opens the specified key server. Key server request processing can be optimized by coding several key server requests within a `PGPKeyServerOpen` / `PGPKeyServerClose` “block”, since this avoids implicit open/close operations for each request.

Errors

```
kPGPError_ServerOpenFailed
kPGPError_ServerSearchFailed
```

Notes, Tips, and Warnings

This function is meaningful for LDAP key servers only. The HTTP protocol does not support the notion of “session”, and so this function is an effective no-op.

A return value of `kPGPError_ServerSearchFailed` indicates that the target key server is not a certificate server, that is, it has no recognizable PGP key space.

The caller is responsible for explicitly closing the specified key server with `PGPKeyServerClose`.

PGPQueryKeyServer

```
PGPError         PGPQueryKeyServer(
                      PGPKeyServerRef     keyServer,
                      PGPFilterRef        filter,
                      PGPEventHandlerProcPtr
                                          callBack,
                      PGPUserValue        callBackArg,
                      PGPKeySetRef        *resultSet,
                      PGPFlags            *resultInfo );
```

Arguments

`keyServer`	the target key server
`filter`	the target key filter
`callBack`	the desired callback function or`( PGPEventHandlerProcPtr )NULL` to indicate no callbacks
`callBackArg`	the user-defined data, to be passed as an argument to the callback function (meaningful only in conjunction with `callBack`)
`resultSet`	the receiving field for the resultant key set
`resultInfo`	the receiving field for the query completion information flags or `( PGPFlags * )NULL` to indicate no information is desired

Description

Applies the specified key filter (constructed as detailed in Chapter 2) to the keys on the specified key server. This yields a resultant key set that contains all of the keys on the key server that meet the key filter criteria.

The `resultInfo` argument is recognized for LDAP key servers only. Currently, only `kPGPKeyServerQuery_PartialResults` flag is ever set, which indicates that the request could not retrieve all of the qualifying keys.

Errors

```
kPGPError_ServerOpenFailed
kPGPError_ServerSearchFailed
kPGPError_UserAbort
```

Notes, Warnings, and Tips

`kPGPError_ServerOpenFailed` and `kPGPError_ServerSearchFailed` are returned for LDAP key servers only, and indicate that no `PGPKeyServerOpen` instance is currently in force.

Specify callBackArg as (PGPUserData)0 to indicate a dummy argument.

The query may legitimately return an empty key set.

The caller is responsible for freeing the resultant key set (empty or not!) with PGPFreeKeySet.

PGPUploadToKeyServer

```
PGPError        PGPUploadToKeyServer(
                     PGPKeyServerRef    keyServer,
                     PGPKeySetRef       keysToUpload,
                     PGPEventHandlerProcPtr
                                        callBack,
                     PGPUserValue       callBackArg,
                     PGPKeySetRef       *keysThatFailed );
```

Arguments

keyServer	the target key server
keysToUpload	the key set containing the keys to be transfered
callBack	the desired callback function or (PGPEventHandlerProcPtr)NULL to indicate no callbacks
callBackArg	the user-defined data, to be passed as an argument to the callback function (meaningful only in conjunction with callBack)
keysThatFailed	the receiving field for the key set containing the keys that could not be successfully transfered

Description

Transfers the specified keys to the specified key server. The key server connection must have been established with an access type of kPGPKSAccess_Administrator.

Errors

kPGPError_ServerAuthorizationRequired
kPGPError_ServerOpenFailed
kPGPError_ServerSearchFailed
kPGPError_ServerKeyFailedPolicy
kPGPError_ServerKeyAlreadyExists
kPGPError_ServerPartialAddFailure
kPGPError_UserAbort

Notes, Warnings, and Tips

kPGPError_ServerOpenFailed and kPGPError_ServerSearchFailed are returned for LDAP key servers only if no PGPKeyServerOpen instance is currently in force.

Depending upon the policies established for the target key server, this function may generate a kPGPEvent_KeyServerSignEvent – potentially one for each key to be uploaded. In this case, a valid callBack argument is required, or the function will fail with kPGPError_ServerAuthorizationRequired.

The returned error code is not always complete – multiple keys may have failed, each for a different reason. The choice of error code obeys the following hierarchy:

- key failed policy – usually indicates that the key was not signed by a recognized user.

- key already exists – the key data presented matches that already on the key server. This implies that the caller already has the most up-to-date version of the key
- key general failure
- other PGPsdk error code

Specify `callBackArg` as `( PGPUserData )0` to indicate a dummy argument.

The caller is responsible for freeing the resultant key set with `PGPFreeKeySet`.

PGPDeleteFromKeyServer (*LDAP key servers only*)

```
PGPError        PGPDeleteFromKeyServer(
                     PGPKeyServerRef    keyServer,
                     PGPKeySetRef       keysToDelete,
                     PGPEventHandlerProcPtr
                                        callBack,
                     PGPUserValue       callBackArg,
                     PGPKeySetRef       *keysThatFailed );
```

Arguments

`keyServer`	the target key server
`keysToDelete`	the key set containing the keys to be deleted
`callBack`	the desired callback function or `( PGPEventHandlerProcPtr )NULL` to indicate no callbacks
`callBackArg`	the user-defined data, to be passed as an argument to the callback function (meaningful only in conjunction with `callBack`)
`keysThatFailed`	the receiving field for the key set containing the keys that could not be successfully deleted

Description

Deletes the specified keys from the specified key server, which must be an LDAP key server. The key server connection must have been established with an access type of `kPGPKSAccess_Administrator`.

Errors

```
kPGPError_ServerOperationNotAllowed
kPGPError_ServerAuthorizationFailed
kPGPError_ServerOpenFailed
kPGPError_ServerSearchFailed
kPGPError_ServerRequestFailed
kPGPError_UserAbort
```

Notes, Warnings, and Tips

This function is *not* valid for HTTP key servers, and results in the return of `kPGPError_ServerOperationNotAllowed`.

`kPGPError_ServerOpenFailed` and `kPGPError_ServerSearchFailed` are returned for LDAP key servers only if no `PGPKeyServerOpen` instance is currently in force.

Depending upon the policies established for the target key server, this function may generate a `kPGPEvent_KeyServerSignEvent` – potentially one for each key to be deleted. In this case, a valid callBack argument is required, or the function will fail with kPGPError_ServerAuthorizationRequired.

Specify callBackArg as (PGPUserData)0 to indicate a dummy argument.

The caller is responsible for freeing the resultant key set with PGPFreeKeySet.

PGPDisableFromKeyServer (*LDAP key servers only*)

```
PGPError         PGPDisableFromKeyServer(
                      PGPKeyServerRef    keyServer,
                      PGPKeySetRef       keysToDisable,
                      PGPEventHandlerProcPtr
                                         callBack,
                      PGPUserValue       callBackArg,
                      PGPKeySetRef       *keysThatFailed );
```

Arguments

keyServer	the target key server
keysToDisable	the key set containing the keys to be disabled
callBack	the desired callback function or (PGPEventHandlerProcPtr)NULL to indicate no callbacks
callBackArg	the user-defined data, to be passed as an argument to the callback function (meaningful only in conjunction with callBack)
keysThatFailed	the receiving field for the key set containing the keys that could not be successfully disabled

Description

Disables the specified keys on the specified key server, which must be an LDAP key server. The key server connection must have been established with an access type of kPGPKSAccess_Administrator.

Errors

kPGPError_ServerOperationNotAllowed
kPGPError_ServerAuthorizationFailed
kPGPError_ServerOpenFailed
kPGPError_ServerSearchFailed
kPGPError_ServerRequestFailed
kPGPError_UserAbort

Notes, Warnings, and Tips

This function is *not* valid for HTTP key servers, and results in the return of kPGPError_ServerOperationNotAllowed.

kPGPError_ServerOpenFailed and kPGPError_ServerSearchFailed are returned for LDAP key servers only if no PGPKeyServerOpen instance is currently in force.

Depending upon the policies established for the target key server, this function may generate a kPGPEvent_KeyServerSignEvent – potentially one for each key to be disabled. In this case, a valid callBack argument is required, or the function will fail with kPGPError_ServerAuthorizationRequired.

Specify `callBackArg` as `( PGPUserData )0` to indicate a dummy argument.

The caller is responsible for freeing the resultant key set with `PGPFreeKeySet`.

PGPGetKeyServerErrorString

```
void            PGPGetKeyServerErrorString(
                     PGPKeyServerRef    keyServer,
                     char               **theString );
```

Arguments

`keyServer`	the target key server
`theString`	the receiving field for a pointer to the associated error text

Description

Places the equivalent error text of the most recent error of the specified key server in the dynamically allocated string buffer.

Notes, Warnings, and Tips

The caller is responsible for deallocating the resultant error text with `PGPFreeData`.

PGPKeyServerClose

```
PGPError        PGPKeyServerClose(
                     PGPKeyServerRef    keyServer );
```

Arguments

`keyServer`	the target key server

Description

Explicitly closes the specified key server (see `PGPKeyServerOpen`).

Notes, Tips, and Warnings

This function is meaningful for LDAP key servers only. The HTTP protocol does not support the notion of “session”, and so this function is an effective no-op.

PGPKeyServerCleanup

```
PGPError        PGPKeyServerCleanup( void );
```

Arguments

Description

Terminates the underlying communcations layer that the PGPsdk requires to access a key server (see `PGPKeyServerInit`). This function effectively destroys a communications session, and so `PGPKeyServerInit` must be called to initiate a new session prior to calling any other key server function.

Appendix A: PGPsdk Error Summary

Table A-1: Generic Errors

Error Constants
kPGPError_NoErr
kPGPError_AssertFailed
kPGPError_BadMemAddress
kPGPError_BadParams
kPGPError_BadPassphrase
kPGPError_BufferTooSmall
kPGPError_CantOpenFile
kPGPError_CorruptData
kPGPError_DiskFull
kPGPError_EndOfIteration
kPGPError_EOF
kPGPError_FeatureNotAvailable
kPGPError_FileLocked
kPGPError_FileNotFound
kPGPError_FileOpFailed
kPGPError_FilePermissions
kPGPError_IllegalFileOp
kPGPError_ImproperInitialization
kPGPError_ItemAlreadyExists
kPGPError_ItemNotFound
kPGPError_LazyProgrammer
kPGPError_OptionNotFound
kPGPError_OutOfMemory
kPGPError_PrefNotFound
kPGPError_ReadFailed
kPGPError_UnknownError
kPGPError_UnknownRequest
kPGPError_UserAbort
kPGPError_WriteFailed

Table A-2: Encode/Decode Errors

Error Constants
kPGPError_CombinedConventionalAndPublicEncryption
kPGPError_CorruptSessionKey
kPGPError_DetachedSignatureFound
kPGPError_DetachedSignatureWithEncryption
kPGPError_DetachedSignatureWithoutSigningKey
kPGPError_IncompatibleAPI
kPGPError_InconsistentEncryptionAlgorithms
kPGPError_InputFile
kPGPError_Interrupted
kPGPError_KeyDisabled
kPGPError_KeyExpired
kPGPError_KeyRevoked
kPGPError_KeyInvalid
kPGPError_KeyUnusableForEncryption
kPGPError_KeyUnusableForSignature
kPGPError_MissingEventHandler
kPGPError_MissingKeySet

`kPGPError_MissingPassphrase`
`kPGPError_MultipleInputOptions`
`kPGPError_MultipleOutputOptions`
`kPGPError_NoDecryptionKeyFound`
`kPGPError_NoInputOptions`
`kPGPError_NoOutputOptions`
`kPGPError_OutputBufferTooSmall`
`kPGPError_RedundantOptions`
`kPGPError_SkipSection`
`kPGPError_TooManyARRKs`

Table A-3: Macintosh MacBinary Format Errors

Error Constants
`kPGPError_NoMacBinaryTranslationAvailable`
`kPGPError_NotMacBinary`

Table A-4: KeySet Filter Errors

Error Constants
`kPGPError_InconsistentFilterClasses`
`kPGPError_InvalidFilterParameter`
`kPGPError_UnknownFilterType`
`kPGPError_UnsupportedHKPFilter`
`kPGPError_UnsupportedLDAPFilter`

Table A-5: Rarely Encountered PGP Errors***

Error Constants
`kPGPError_AsciiParseIncomplete`
`kPGPError_BadCipherNumber`
`kPGPError_BadHashNumber`
`kPGPError_BadKeyLength`
`kPGPError_BadPacket`
`kPGPError_BadSessionKeyAlgorithm`
`kPGPError_BadSessionKeySize`
`kPGPError_BadSignatureSize`
`kPGPError_CantDecrypt`
`kPGPError_CantHash`
`kPGPError_ConfigParseFailure`
`kPGPError_ConfigParseFailureBadFunction`
`kPGPError_ConfigParseFailureBadOptions`
`kPGPError_EnvPriorityTooLow`
`kPGPError_ExtraDateOnSignature`
`kPGPError_FIFOReadError`
`kPGPError_InvalidCommit`
`kPGPError_KeyIsLocked`
`kPGPError_OutOfRings`
`kPGPError_RandomSeedTooSmall`
`kPGPError_AdditionalRecipientRequestKeyNotFound`
`kPGPError_SecretKeyNotFound`
`kPGPError_SignatureBitsWrong`
`kPGPError_SizeAdviseFailure`
`kPGPError_TroubleBadTrust`
`kPGPError_TroubleBareKey`
`kPGPError_TroubleDuplicateKey`
`kPGPError_TroubleDuplicateKeyID`
`kPGPError_TroubleDuplicateName`

kPGPError_TroubleDuplicateSecretKey
kPGPError_TroubleDuplicateSignature
kPGPError_TroubleDuplicateUnknown
kPGPError_TroubleImportingNonexportableSignature
kPGPError_TroubleKeySubKey
kPGPError_TroubleKeyTooBig
kPGPError_TroubleNameTooBig
kPGPError_TroubleNewSecretKey
kPGPError_TroubleOldSecretKey
kPGPError_TroubleSecretKeyTooBig
kPGPError_TroubleSignatureTooBig
kPGPError_TroubleSigSubKey
kPGPError_TroubleUnexpectedName
kPGPError_TroubleUnexpectedSignature
kPGPError_TroubleUnexpectedSubKey
kPGPError_TroubleUnexpectedTrust
kPGPError_TroubleUnexpectedUnknown
kPGPError_TroubleUnknownPacketByte
kPGPError_TroubleUnknownTooBig
kPGPError_TroubleVersionBugCur
kPGPError_TroubleVersionBugPrev
kPGPError_UnbalancedScope
kPGPError_UnknownCharMap
kPGPError_UnknownSignatureType
kPGPError_UnknownString2Key
kPGPError_UnknownVersion
kPGPError_WrongScope

*** These error codes should rarely be encountered, if ever. Most are indicative of internal PGPsdk errors, and not all are propagated to the PGPsdk level.

Table A-6: Key Errors

Error Constants
kPGPError_KeyPacketTruncated
kPGPError_KeyTooLarge
kPGPError_MalformedKeyComponent
kPGPError_MalformedKeyExponent
kPGPError_MalformedKeyModulus
kPGPError_PublicKeyTooLarge
kPGPError_PublicKeyTooSmall
kPGPError_PublicKeyUnimplemented***
kPGPError_RSAPublicExponentIsEven
kPGPError_RSAPublicModulusIsEven
kPGPError_UnknownKeyVersion
kPGPError_UnknownPublicKeyAlgorithm

*** `kPGPError_PublicKeyUnimplemented` indicates that the active key was generated with an algorithm that is not implemented for that version of the PGPsdk. For example, passing an RSA key to any function of a Diffie-Hellman only version will result in this error.

Table A-7: Signature Errors

Error Constants
kPGPError_ExtraSignatureMaterial
kPGPError_MalformedSignatureInteger
kPGPError_TruncatedSignature
kPGPError_UnknownSignatureAlgorithm
kPGPError_UnknownSignatureVersion

Table A-8: KeyDB Errors

Error Constants
kPGPError_CertifyingKeyDead
kPGPError_DuplicateCert
kPGPError_DuplicateUserID
kPGPError_FileCorrupt
kPGPError_InvalidProperty
kPGPError_ItemIsReadOnly
kPGPError_ItemWasDeleted
kPGPError_KeydbMismatch
kPGPError_OutOfEntropy

Table A-9: **Key Server Errors**

Error Constants
kPGPError_ServerAddFailed
kPGPError_ServerAuthorizationFailed
kPGPError_ServerAuthorizationRequired
kPGPError_ServerBadKeysInSearchResults
kPGPError_ServerBindFailed
kPGPError_ServerConnectFailed
kPGPError_ServerCorruptKeyBlock
kPGPError_ServerInvalidProtocol
kPGPError_ServerKeyAlreadyExists
kPGPError_ServerKeyFailedPolicy
kPGPError_ServerOpenFailed
kPGPError_ServerOperationNotAllowed
kPGPError_ServerPartialAddFailure
kPGPError_ServerRequestFailed
kPGPError_ServerSearchFailed
kPGPError_ServerSocketError
kPGPError_ServerTooManyResults
kPGPError_ServerUnknownHost
kPGPError_ServerUnknownResponse

Appendix B: References and Recommended Reading

Introductory Readings

- Bacard, Andre, *Computer Privacy Handbook*, Peachpit Press, 1995
- Garfinkel, Simson, *Pretty Good Privacy,* O'Reilly & Associates, 1995
- Schneier, Bruce, *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*, John Wiley & Sons, 1996
- Schneier, Bruce, *Email Security*, John Wiley & Sons, 1995
- Stallings, William, *Protect Your Privacy*, Prentice Hall, 1994

Other Readings

- Lai, Xuejia, *On the Design and Security of Block Ciphers*, Institute for Signal and Information Processing, ETH-Zentrum, Zurich, Switzerland, 1992
- Lai, Xuejia, James L. Massey, Sean Murphy, “Markov Ciphers and Differential Cryptanalysis,” *Advances in Cryptology - EUROCRYPT'91*
- Rivest, Ronald, *The MD5 Message Digest Algorithm,* MIT Laboratory for Computer Science, 1991
- Wallich, Paul, “Electronic Envelopes,” *Scientific American*, Feb. 1993, page 30.
- Zimmermann, Philip, "A Proposed Standard Format for RSA Cryptosystems,” *Advances in Computer Security, Vol. III*, edited by Rein Turn, Artech House, 1988

Glossary

A5: a trade-secret cryptographic algorithm used in European cellular telephones.

Access control: a method of restricting access to resources, allowing only privileged entities access.

Additional recipient request key: a special key whose presence that indicates that all messages encrypted to its associated base key should also be automatically encrypted to it. Sometimes referred to by its marketing term, *additional decryption key*.

AES (Advanced Encryption Standard): NIST approved standards, usually used for the next 20 - 30 years.

AKEP (Authentication Key Exchange Protocol): key transport based on symmetric encryption allowing two parties to exchange a shared secret key, secure against passive adversaries.

Algorithm (encryption): a set of mathematical rules (logic) used in the processes of encryption and decryption.

Algorithm (hash): a set of mathematical rules (logic) used in the processes of message digest creation and key/signature generation.

Anonymity: of unknown or undeclared origin or authorship, concealing an entity's identification.

ANSI (American National Standards Instituted): develops standards through various Accredited Standards Committees (ASC). The X9 committee focuses on security standards for the financial services industry.

API (Application Programming Interface): provides the means to take advantage of software features, allowing dissimilar software products to interact upon one another.

ASN.1 (Abstract Syntax Notation One): ISO /IEC standard for encoding rules used in ANSI X.509 certificates, two types exist: DER (Distinguished Encoding Rules) and BER (Basic Encoding Rules).

Asymmetric keys: a separate but integrated user key-pair, comprised of one public-key and one private-key. Each key is one way, meaning that a key used to encrypt information can not be used to decrypt the same data.

Authentication: to prove genuine by corroboration of the identity of an entity.

Authorization certificate: an electronic document to prove one's access or privilege rights, also to prove one is who they say they are.

Authorization: to convey official sanction, access or legal power to an entity.

Blind signature: ability to sign documents without knowledge of content, similar to a notary public.

Block cipher: a symmetric cipher operating on blocks of plain text and cipher text, usually 64 bits.

Blowfish: a 64-bit block symmetric cipher consisting of key expansion and data encryption. A fast, simple, and compact algorithm in the public domain written by Bruce Schneier.

CA (Certificate Authority): a trusted third party (TTP) who creates certificates that consist of assertions on various attributes and binds them to an entity and/or to their public key.

CAPI (Crypto API): Microsoft's crypto API for Windows-based operating systems and applications.

Capstone: an NSA-developed cryptographic chip that implements a US government Key Escrow capability.

CAST: a 64-bit block cipher using 64-bit key, six S-boxes with 8-bit input and 32-bit output, developed in Canada by Carlisle Adams and Stafford Tavares.

CBC (Cipher Block Chaining): the process of having plain text XORed with the previous cipher text block before it is encrypted, thus adding a feedback mechanism to a block cipher.

CDK (Crypto Developer Kit): a documented environment, including an API for third parties to write secure applications using a specific vendor's cryptographic library.

CERT (Computer Emergency Response Team): security clearinghouse that promotes security awareness. CERT provides 24-hour technical assistance for computer and network security incidents. CERT is located at the Software Engineering Institute at Carnegie Mellon University in Pittsburgh, PA.

Certificate (digital certificate): an electronic document attached to a public key by a trusted third party, which provides proof that the public key belongs to a legitimate owner and has not been compromised.

CFM (Cipher Feedback Mode): a block cipher that has been implemented as a self-synchronizing stream cipher.

CDSA (Common Data Security Architecture): Intel Architecture Labs (IAL) developed this framework to address the data security problems inherent to Internet and Intranet for use in Intel and others' Internet products.

Certification: endorsement of information by a trusted entity.

CHAP (Challenge Authentication Protocol): a session-based, two-way password authentication scheme.

Cipher text: the result of manipulating either characters or bits via substitution, transposition, or both.

Clear text: characters in a human readable form or bits in a machine readable form (also called *plain text*)

Confidentiality: the act of keeping something private and secret from all but those who are authorized to see it.

Cookie: Persistent Client State HTTP Cookie - a file or token of sorts, that is passed from the web server to the web client (your browser) that is used to identify you and could record personal information such as ID and password, mailing address, credit card number, and other information.

CRAB: a 1024-byte block cipher (similar to MD5), using techniques from a one-way hash function, developed by Burt Kaliski and Matt Robshaw at RSA Laboratories.

Credentials: something that provides a basis for credit or confidence.

CRL (Certificate Revocation List): an online, up-to-date list of previously issued certificates that are no longer valid.

Cross-certification: two or more organizations or Certificate Authorities that share some level of trust.

Cryptanalysis: the art or science of transferring cipher text into plain text without initial knowledge of the key used to encrypt the plain text.

CRYPTOKI: same as PKCS #11.

Cryptography: the art and science of creating messages that have some combination of being private, signed, unmodified with nonrepudiation.

Cryptosystem: a system comprised of cryptographic algorithms, all possible plain text, cipher text, and keys.

Data integrity: a method of ensuring information has not been altered by unauthorized or unknown means.

Decryption: the process of turning cipher text back into plain text.

DES (Data Encryption Standard): a 64-bit block cipher, symmetric algorithm also known as Data Encryption Algorithm (DEA) by ANSI and DEA-1 by ISO. Widely used for over 20 years, adopted in 1976 as FIPS 46.

Dictionary attack: a calculated brute force attack to reveal a password by trying obvious and logical combinations of words.

Diffie-Hellman: the first public key algorithm, invented in 1976, using discrete logarithms in a finite field.

Digital cash: electronic money that stored and transferred through a variety of complex protocols.

Direct trust: an establishment of peer-to-peer confidence.

Discrete logarithm: the underlying mathematical problem used in/by asymmetric algorithms, like Diffie-Hellman and Elliptic Curve. It is the inverse problem of modular exponentiation, which is a one-way function.

DMS (Defense Messaging System): standards designed by the U.S. Department of Defense to provide a secure and reliable enterprise-wide messaging infrastructure for government and military agencies.

DNSSEC (Domain Name System Security Working Group): a proposed *IETF* draft that will specify enhancements to the DNS protocol to protect the DNS against unauthorized modification of data and against masquerading of data origin. It will add data integrity and authentication capabilities to the DNS via digital signatures.

DSA (Digital Signature Algorithm): a public-key digital signature algorithm proposed by NIST for use in DSS.

Digital signature: an electronic identification of a person or thing created by using a public-key algorithm. Intended to verify to a recipient the integrity of data and identity of the sender of the data.

DSS (Digital Signature Standard): a NIST proposed standard (FIPS) for digital signatures using DSA.

ECC (Elliptic Curve Cryptosystem): a unique method for creating public-key algorithms based on mathematical curves over finite fields or with large prime numbers.

EDI (Electronic Data Interchange): the direct, standardized computer-to-computer exchange of business documents (purchase orders, invoices, payments, inventory analyses, and others) between your organization and your suppliers and customers.

EES (Escrowed Encryption Standard): a proposed U.S. government standard for escrowing private keys.

El Gamal scheme: used for both digital signatures and encryption based on discrete logarithms in a finite field, can be used with the DSA function.

Encryption: the process of disguising a message in such a way as to hide its substance.

Entropy: a mathematical measurement of the amount of uncertainty or randomness.

FEAL: a block cipher using 64-bit block and 64-bit key, design by A.Shimizu and S.Miyaguchi at NTT Japan.

Filter: a function, set of functions, or combination of functions that applies some number of transforms to its input set, yielding an output set containing only those members of the input set that satisfy the transform criteria. The selected members may or may not be further transformed in the resultant output set. An example would be a search function that accepts multiple strings having a boolean relationship (`( like` *a* `or like` *b* `) but not containing` *c*) and optionally forces the case of the found strings in the resultant output.

Fingerprint: a unique identifier for a key that is obtained by hashing specific portions of the key data.

FIPS (Federal Information Processing Standard): a U.S. government standard published by NIST.

Firewall: a combination of hardware and software that protects the perimeter of the public/private network against certain attacks to ensure some degree of security.

GAK (Government Access to Keys): a method for the government to escrow individual's private key.

Gost: a 64-bit symmetric block cipher using a 256-bit key, developed in the former Soviet Union.

GSS-API (Generic Security Services API): IETF RFC 1508 is a high-level security API, which isolates session-oriented application code from implementation details.

Hash function: a one-way hash function - a function that produces a message digest that cannot be reversed to produced the original.

Hierarchical trust: a graded series of entities that distribute trust in an organized fashion, commonly used in ANSI X.509 issuing certifying authorities.

HTTP (HyperText Transfer Protocol): a common protocol used to transfer documents between servers or from a server to a client.

IDEA (International Data Encryption Standard): a 64-bit block symmetric cipher using 128-bit keys based on mixing operations from different algebraic groups. Considered one of the strongest algorithms.

IETF (Internet Engineering Task Force): a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet. It is open to any interested individual.

Identity certificate: a signed statement that binds a key to the name of an individual and has the intended meaning of delegating authority from that named individual to the public key.

Initialization vector (IV): a block of arbitrary data that serves as the starting point for a block cipher using a chaining feedback mode (see cipher block chaining).

Integrity: assurance that data is not modified (by unauthorized persons) during storage or transmittal.

IPsec: a TCP/IP layer encryption scheme under consideration within the IETF.

ISA/KMP (Internet Security Association, Key Mgt. Protocol): defines the procedures for authenticating a communicating peer, creation and management of Security Associations, key generation techniques, and threat mitigation (for example, denial of service and replay attacks).

ISO (International Organization for Standardization): responsible for a wide range of standards, like the OSI model and international relationship with ANSI on X.509.

ITU-T (International Telecommunication Union-Telecommunication): formally the CCITT (Consultative Committee for International Telegraph and Telephone), a worldwide telecommunications technology standards organization.

Kerberos: a trusted-third-party authentication protocol developed at MIT.

Key: a means of gaining or preventing access, possession, or control represented by any one of a large number of values.

Key escrow/recovery: a mechanism that allows a third party to retrieve the cryptographic keys used for data confidentiality, with the ultimate goal of recovery of encrypted data.

Key exchange: a scheme for two or more nodes to transfer a secret session key across an unsecured channel.

Key length: the number of bits representing the key size; the longer the key, the stronger it is.

Key management: the process and procedure for safely storing and distributing accurate cryptographic keys, the overall process of generating and distributing cryptographic key to authorized recipients in a secure manner.

Key splitting: a process for dividing portions of a single key between multiple parties, none having the ability to reconstruct the whole key.

LDAP (Lightweight Directory Access Protocol): a simple protocol that supports access and search operations on directories containing information such as names, phone numbers, and addresses across otherwise incompatible systems over the Internet.

Lexical section: a distinct portion of a message that contains a specific class of data, for example, clear-signed data, encrypted data, and key data.

MAA (Message Authenticator Algorithm): an ISO standard that produces a 32-bit hash, designed for IBM mainframes.

MAC (Message Authentication Code): a key-dependent one-way hash function, requiring the use of the identical key to verify the hash.

MD2 (Message Digest 2): 128-bit one-way hash function designed by Ron Rivest, dependent on a random permutation of bytes.

MD4 (Message Digest 4): 128-bit one-way hash function designed by Ron Rivest, using a simple set of bit manipulations on 32-bit operands.

MD5 (Message Digest 5): improved version of MD4, more complex but still a 128-bit one-way hash function.

Message digest: a number that is derived from a message. Change a single character in the message and the message will have a different message digest.

MIC (Message Integrity Check): originally defined in PEM for authentication using MD2 or MD5. Micalg (message integrity calculation) is used in secure MIME implementations.

MIME (Multipurpose Internet Mail Extensions): a freely available set of specifications that offers a way to interchange text in languages with different character sets, and multi-media e-mail among many different computer systems that use Internet mail standards.

MMB (Modular Multiplication-based Block): based on IDEA, Joan Daemen developed this 128-bit key /128-bit block size symmetric algorithm, not used because of its susceptibility to linear cryptanalysis.

MOSS (MIME Object Security Service): defined in RFC 1848, it facilitates encryption and signature services for MIME, including key management based on asymmetric techniques (not widely used).

MSP (Message Security Protocol): the military equivalent of PEM, an X.400-compatible application level protocol for securing e-mail, developed by the NSA in late 1980.

MTI: a one-pass key agreement protocol by Matsumoto, Takashima, and Imai that provides mutual-key authentication without key confirmation or entity authentication.

NAT (Network Address Translator): RFC 1631, a router connecting two networks together; one designated as inside, is addressed with either private or obsolete addresses that need to be converted into legal addresses before packets are forwarded onto the other network (designated as outside).

NIST (National Institute for Standards and Technology): a division of the U.S. Dept. of Commerce that publishes open, interoperability standards called FIPS.

Non-repudiation: preventing the denial of previous commitments or actions.

Oakely: the "Oakley Session Key Exchange" provides a hybrid Diffie-Hellman session key exchange for use within the ISA/KMP framework. Oakley provides the important property of "Perfect Forward Secrecy."

One-time pad: a large nonrepeating set of truly random key letters used for encryption, considered the only perfect encryption scheme, invented by Major J. Mauborgne and G. Vernam in 1917.

One-way hash: a function of a variable string to create a fixed length value representing the original pre-image, also called message digest, fingerprint, message integrity check (MIC).

Orange Book: the National Computer Security Center book entitled *Department of Defense Trusted Computer Systems Evaluation Criteria* that defines security requirements.

PAP (Password Authentication Protocol): an authentication protocol that allows PPP peers to authenticate one another, does not prevent unauthorized access but merely identifies the remote end.

Passphrase: an easy-to-remember phrase used for better security than a single pass word, key crunching converts it into a random key.

Password: a sequence of characters or word that a subject submits to a system for purposes of authentication, validation, or verification.

PCT (Private Communication Technology): a protocol developed by Microsoft and Visa for secure communications on the Internet.

PEM (Privacy Enhanced Mail): a protocol to provide secure internet mail, (RFC 1421-1424) including services for encryption, authentication, message integrity, and key management. PEM uses ANSI X.509 certificates.

Perfect forward secrecy: a cryptosystem in which the cipher text yields no possible information about the plain text, except possibly the length.

Primative filter: a function that applies a single transform to its input set, yielding an output set containing only those members of the input set that satisfy the transform criteria. An example would be a search function that accepts only a single string and outputs a list of line numbers where the string was found.

Pretty Good Privacy (PGP): an application & protocol (RFC 1991) for secure e-mail and file encryption developed by Phil R. Zimmermann. Originally published as Freeware, the source code has always been available for public scrutiny. PGP uses a variety of algorithms, like IDEA, RSA, DSA, MD5, SHA-1 for providing encryption, authentication, message integrity, and key management. PGP is based on the “Web-of-Trust” model and has worldwide deployment.

PGP/MIME: an IETF standard (RFC 2015) that provides privacy and authentication using the Multipurpose Internet Mail Extensions (MIME) security content types described in RFC1847, currently deployed in PGP 5.0 and later versions.

PKCS (Public Key Crypto Standards): set of *de facto* standards for public key cryptography developed in cooperation with an informal consortium (Apple, DEC, Lotus, Microsoft, MIT, RSA, and Sun) that includes algorithm-specific and algorithm-independent implementation standards. Specifications defining message syntax and other protocols controlled by RSA Data Security Inc.

PKI (Public Key Infrastructure): a widely available and accessible certificate system for obtaining an entity’s public-key with some degree of certainty that you have the “right” key and that it has not been revoked.

Plain text (or clear text): the human readable data or message before it is encrypted.

Pseudo-random number: a number that results from applying randomizing algorithms to input derived from the computing environment, for example, mouse coordinates. See *random number*.

Private key: the privately held “secret” component of an integrated asymmetric key pair, often referred to as the decryption key.

Public key: the publicly available component of an integrated asymmetric key pair often referred to as the encryption key.

RADIUS (Remote Authentication Dial-In User Service): an IETF protocol (developed by Livingston, Enterprise), for distributed security that secures remote access to networks and network services against unauthorized access. RADIUS consists of two pieces: authentication server code and client protocols.

Random number: an important aspect to many cryptosystems, and a necessary element in generating a unique key(s) that are unpredictable to an adversary. True random numbers are usually derived from analog sources, and usually involve the use of special hardware.

RC2 (Rivest Cipher 2): variable key size, 64-bit block symmetric cipher, a trade secret held by RSA, SDI.

RC4 (Rivest Cipher 4): variable key size stream cipher, once a proprietary algorithm of RSA Data Security, Inc.

RC5 (Rivest Cipher 5): a block cipher with a variety of arguments, block size, key size, and number of rounds.

RIPE-MD: an algorithm developed for the European Community's RIPE project, designed to resist known cryptanalysis attacks and produce a 128-bit hash value, a variation of MD4.

REDOC: a US-patented block cipher algorithm developed by M. Wood, using a 160-bit key and an 80-bit block.

Revocation: retraction of certification or authorization.

RFC (Request for Comment): an IETF document, either FYI (For Your Information) RFC sub-series that are overviews and introductory or STD RFC sub-series that identify specify Internet standards. Each RFC has an RFC number by which it is indexed and by whichit can be retrieved. *(***www.ietf.org** *)*

ROT-13 (Rotation Cipher): a simple substitution (Caesar) cipher, rotating each 26 letters 13 places.

RSA: short for RSA Data Security, Inc.; or referring to the principals: Ron Rivest, Adi Shamir, and Len Adleman; or to the algorithm they invented. The RSA algorithm is used in public-key cryptography and is based on the fact that it is easy to multiply two large prime numbers together, but hard to factor them out of the product.

SAFER (Secure And Fast Encryption Routine): a non-proprietary block cipher 64-bit key encryption algorithm. It is not patented, is available license free, and was developed by Massey, who also developed IDEA.

Salt: a random string that is concatenated with passwords (or random numbers) before being operated on by a one-way function. This concatenation effectively lengthens and obscures the password, making the cipher text less susceptible to dictionary attacks.

SDSI (Simple Distributed Security Infrastructure): a new *PKI* proposal from Ronald L. Rivest (MIT), and Butler Lampson (Microsoft). It provides a means of defining groups and issuing group-membership, access-control lists, and security policies. SDSI's design emphasizes linked local name spaces rather than a hierarchical global name space.

SEAL (Software-optimized Encryption ALgorithm): a fast stream cipher for 32-bit machines designed by Rogaway and Coppersmith.

Secret key: either the "private-key" in public-key (asymmetric) algorithms or the "session-key" in symmetric algorithms.

Secure channel: a means of conveying information from one entity to another such that an adversary does not have the ability to reorder, delete, insert, or read (SSL, IPsec, whispering in someone's ear).

Self-signed key: a public-key that has been signed by the corresponding private-key for proof of ownership.

SEPP (Secure Electronic Payment Protocol): an open specification for secure bankcard transactions over the Internet. Developed by IBM, Netscape, GTE, Cybercash, and MasterCard.

SESAME (Secure European System for Applications in a Multi-vendor environment): European research and development project that extended Kerbros by adding authorization and access services.

Session key: the secret (symmetric) key used to encrypt each set of data on a transaction basis. A different session key is used for each communication session.

SET (Secure Electronic Transaction): provides for secure exchange of credit card numbers over the Internet.

SHA-1 (Secure Hash Algorithm): the 1994 revision to SHA, developed by NIST, (FIPS 180-1) used with DSS produces a 160-bit hash, similar to MD4, which is very popular and is widely implemented.

Single sign-on: one log-on provides access to all resources of the network.

SKIP (Simple Key for IP): simple key-management for Internet protocols, developed by Sun Microsystems, Inc.

Skipjack: the 80-bit key encryption algorithm contained in NSA's Clipper chip. The algorithm is classified; NSA will not release information on how it works.

SKMP (Secure-Key Management Protocol): an IBM proposed key-recovery architecture that uses a key encapsulation technique to provide the key and message recovery to a trusted third-party escrow agent.

S/MIME (Secure Multipurpose Mail Extension): a proposed standard developed by Deming software and RSA Data Security for encrypting and/or authenticating MIME data. S/MIME defines a format for the MIME data, the algorithms that must be used for interoperability (RSA, RC2, SHA-1), and the additional operational concerns such as ANSI X.509 certificates and transport over the Internet.

SNAPI (Secure Network API): a Netscape driven API for security services that provide ways for resources to be protected against unauthorized users, for communication to be encrypted and authenticated, and for the integrity of information to be verified.

SPKI (Simple Public Key Infrastructure): an IETF proposed draft standard, (by Ellison, Frantz, and Thomas) public key certificate format, associated signature and other formats, and key acquisition protocol. Recently merged with Ron Rivest's SDSI proposal.

SSH (Secure Shell): an IETF proposed protocol for securing the transport layer by providing encryption, cryptographic host authentication, and integrity protection.

SSH (Site Security Handbook): the Working Group (WG) of the Internet Engineering Task Force has been working since 1994 to produce a pair of documents designed to educate the Internet community in the area of security. The first document is a complete reworking of RFC 1244, and is targeted at system and network administrators, as well as decision makers (middle management).

SSL (Secure Socket Layer): developed by Netscape to provide security and privacy over the Internet. Supports server and client authentication and maintains the security and integrity of the transmission channel. Operates at the transport layer and mimics the "sockets library," allowing it to be application independent. Encrypts the entire communication channel and does not support digital signatures at the message level.

SST (Secure Transaction Technology): a secure payment protocol developed by Microsoft and Visa as a companion to the PCT protocol.

Stream cipher: a class of symmetric-key encryption where transformation can be changed for each symbol of plain text being encrypted, useful for equipment with little memory to buffer data.

STU-III (Secure Telephone Unit): NSA designed telephone for secure voice and low-speed data communications for use by the U.S. Dept. of Defense and their contractors.

Substitution cipher: the characters of the plain text are substituted with other characters to form the cipher text.

S/WAN (Secure Wide Area Network): RSA Data Security, Inc. driven specifications for implementing IPSec to ensure interoperability among firewall and TCP/IP products. S/WAN's goal is to use IPSec to allow companies to mix-and-match firewall and TCP/IP stack products to build Internet-based Virtual Private Networks (VPNs).

Symmetric algorithm: a.k.a., conventional, secret-key, single-key algorithms; the encryption and decryption key are either the same or can be calculated from one another. Two sub-categories exist: Block and Stream.

TACACS+ (Terminal Access Controller Access Control System): a protocol that provides remote access authentication, authorization, and related accounting and logging services, used by Cisco Systems.

Timestamping: recording the time of creation or existence of information.

TLS (Transport Layer Security): an IETF draft, version 1 is based on the Secure Sockets Layer (SSL) version 3.0 protocol, and provides communications privacy over the Internet.

TLSP (Transport Layer Security Protocol): ISO 10736, draft international standard.

Transposition cipher: the plain text remains the same but the order of the characters is transposed.

Triple DES: an encryption configuration in which the DES algorithm is used three times with three different keys.

Trust: a firm belief or confidence in the honesty, integrity, justice, and/or reliability of a person, company, or other entity.

TTP (Trust Third-Party): a responsible party in which all participants involved agree upon in advance, to provide a service or function, such as certification, by binding a public-key to an entity, time-stamping, or key-escrow.

UEPS (Universal Electronic Payment System): a smart-card (secure debit-card) -based banking application developed for South Africa where poor telephones make on-line verification impossible.

Validation: a means to provide timeliness of authorization to use or manipulate information or resources.

Verification: to authenticate, confirm, or establish accuracy.

VPN (Virtual Private Network): allows private networks to span from the end-user, across a public network (Internet) directly to the Home Gateway of choice, such as your company's Intranet.

WAKE (Word Auto Key Encryption): produces a stream of 32-bit words, which can be XORed with plain text stream to produce cipher text, invented by David Wheeler.

Web of Trust: a distributed trust model used by PGP to validate the ownership of a public key where the level of trust is cumulative based on the individual's knowledge of the "introducers."

W3C (World Wide Web Consortium): an international industry consortium founded in 1994 to develop common protocols for the evolution of the World Wide Web.

XOR: exclusive-or operation, a mathematical way to represent differences.

X.509v3: an ITU-T digital certificate that is an internationally recognized electronic document used to prove identity and public key ownership over a communication network. It contains the issuer's name, the user's identifying information, and the issuer's digital signature, as well as other possible extensions in version 3.

X9.17: an ANSI specification that details the methodology for generating random and pseudo-random numbers.

Index