A Mechanism for VHDL Source Protection
1 Overview

The intent of this specification is to define the VHDL source protection mechanism. It defines the rules to encrypt the VHDL source. It also defines the format of the encrypted VHDL file. Its primary audiences are implementers of tools that produce encrypted VHDL, or the tools that consume and process the encrypted VHDL.

This specification has been described using the context-free syntax described in the IEEE Standard VHDL Language Reference Manual (IEEE Std. 1076-1993) section 0.2.1.

2 Conventions used in document

This proposal recommends the use of pragmas to demarcate parts of VHDL source that need to be encrypted and to specify various cryptographic directives to be used during the encryption and decryption process. For encryption of the VHDL source, the pragmas are defined in the following format. This subclause specifies the syntactic mechanism that shall be used for specifying pragmas, without standardizing on any particular pragmas.

pragma ::= `protect { pragma_expression } \n
pragma_expression ::= pragma_keyword

| pragma_keyword = pragma_value

| pragma_value

pragma_value ::= constant_expression

| string

pragma_keyword ::= begin

| end

| data_keyowner

| data_keyname

| data_method

| key_keyowner

| key_method

| key_keyname

| data_public_key
In addition, the following convention is used to define the source of various pragma keywords.

ENCRYPTION INPUT refers to anything that user provides to encrypting tool.

ENCRYPTION OUTPUT refers to the output generated by the encryption tool.

DECRIPTION INPUT refers to the input to decryption tool (which is the output of encrypting tool)

3 Protected Envelopes

Protected Envelopes specify a region of text which shall be encrypted prior to analysis by the source language processor. These regions of text are structured to provide the source language processor with the specification of the cryptographic algorithm, key, envelope attributes, and textual design data.

Envelopes may be defined for either of two modes of processing. Encryption envelopes specify the pragma expressions for encrypting source text regions. Decryption envelopes specify the pragma expressions for decrypting encrypted text regions. Decryption envelopes may contain other envelopes within their enclosed data block. The number of nested decryption envelopes that can be processed is implementation-specified.
3.1 Specifying Protected Envelopes

All information which identifies a Protected Envelope is introduced by the `protect` pragma. This pragma is reserved by this standard for the description of Protected Envelopes, and is the prefix for specifying the regions and processing specifications for each protected envelope. Additional information is associated with the pragma by appending pragma expressions.

```
hdl_envelope ::= encrypt_envelope
               | decrypt_envelope

encrypt_envelope ::= protectPragma encrypt_content_params beginPragma source_text endPragma

encrypt_content_params ::= key_block_params [license_params] [encodingPragma] [author_infoPragma] [data_params_set] [commentPragma]

key_block_params ::= \{key_params_set\}

key_params_set ::= key_keyownerPragma key_keynamePragma key_methodPragma

data_params_set ::= data_keyownerPragma data_keynamePragma data_methodPragma

license_params ::= decrypt_licensePragma | runtime_licensePragma

decrypt_envelope ::= begin_protectedPragma decrypt_content_params decrypt_data_block end_protectedPragma

decrypt_content_params ::= \{decrypt_key_block\} [encodingPragma] [author_infoPragma] [commentPragma]

decrypt_key_block ::= key_params_set key_block digest_block

key_block ::= key_blockPragma encoded_text end_key_blockPragma

digest_block ::= digest_blockPragma encoded_text end_digest_blockPragma

decrypt_data_block ::= data_block digest_block

data_block ::= data_blockPragma encoded_text end_data_blockPragma

author_infoPragma ::= \`protect\{author_info_keywords\}=string\n
author_info_keywords ::= author | author_info | encrypt_agent

protectPragma ::= \`protect \n

beginPragma ::= \`protect begin \n

endPragma ::= \`protect end \n

key_keyownerPragma ::= \`protect key_keyowner=string \n

key_keynamePragma ::= \`protect key_keyname=string \n```

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.
key_methodPragma ::=  `protect key_method=method_name\n
data_keyownerPragma ::=  `protect data_keyowner=string\n
data_keynamePragma ::=  `protect data_keyname=string \n
data_methodPragma ::=  `protect data_method=method_name\n
method_name ::= DES | AES | RSA | RC2 | RC4 | RC5

decrypt_licensePragma ::=  `protect decrypt_license=string\n
runtime_licensePragma ::=  `protect runtime_license=string \n
begin_protectedPragma ::=  `protect begin_protected \n
end_protectedPragma ::=  `protect end_protected \n
key_blockPragma ::=  `protect key_block \n
end_key_blockPragma ::=  `protect end_key_block \n
encodingPragma ::=  `protect encoding_descriptor \n
encoding_descriptor ::=  encoding=encoding_type [ line_length=number ] [ bytes=number ] \n
encoding_type ::= raw|uuencode|RFC1113_printable | RFC2045_base64 | RFC2045_quoted-printable

digest_blockPragma ::=  `protect digest_block \n
end_digest_blockPragma ::=  `protect end_digest_block \n
data_blockPragma ::=  `protect data_block \n
end_data_blockPragma ::=  `protect end_data_block \n
commentPragma ::=  `protect comment=string \n

Note:
source_text: The source text encompasses all the text, comments, included pragma directives, user code etc.
encoded_text: is the binary data encoded in printable characters, spanning over multiple lines. This can contains both the encrypted and the message digest data.

The pragma expressions between the protect pragma and the begin pragma in a encryption envelope or between the begin_protect pragma and end_protect pragma are processed to encrypt or decrypt the data in the envelopes.
Examples:

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
package pack_inst is
  `protect
  `protect data_keyowner = owner1
  `protect data_method = RC5
  `protect data_keyname = data_test1.1
  `protect key_keyowner = keyowner1
  `protect key_method = RC4
  `protect key_keyname = key_test1.1
  `protect key_keyowner = keyowner2
  `protect key_method = DES
  `protect key_keyname = key_test1.2
  `protect begin
  signal sigp_protected : std_logic;
  `protect end
end pack_inst;
```

After processing the above input VHDL the encrypting tool should generate data similar to the following:

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
package pack_inst is
  begin_protected
  key_keyowner=keyowner1
  key_keyname=key_test1.1
  key_method=RC4
  encoding=RFC1113_printable line_length=64 bytes=208
  key_block
  T/R0BKmye8voSe1lJJdpeF3ga6182MsHa15sGnOLiVhOYX4unu0XG6W65Nyu
  FY6FWTX+5skQu+qyW+5mVFeMFOTPa6UD8Lfy+5S8MuzTDVCGpg8d9k7nXb92SslcC
  fuE/rUhMCQEOtF0sRvAcLGX5Mh3UqI3bncGe8CC2s1yDmHIdWvjuotUN3xDaZV
  sqRv98aQ6gZT5Dg=
  end_key_block
  encoding=RFC1113_printable line_length=64 bytes=28
  digest_block
  X9PyX59giDALGPebhCyRkc3f7E=
  end_digest_block
  key_keyowner=keyowner2
  key_keyname=key_test1.2
  key_method=DES
  encoding=RFC1113_printable line_length=64 bytes=216
  key_block
  JgRb6WTHB7I/JeMPU/Z6wYs/JE5g6kExOo2XBDmXto/Zy6KCD5vQWEDqj/PWW0
  OlxuAfJfIIIr5WmN/uVHA2F4Hb6BrVVsqNDITZQZBjMGHx/CjDNZvuLevV9jnia6
  pPJTG2pGg6FQMoI1ouTK2X39Z/Tn/Q2uXPSLM91Ftd7y58oc/VjQZ4rEV05DzLw
  8BuR9/CDG3A5CYhXn+Xg==
  end_key_block
  encoding=RFC1113_printable line_length=64 bytes=28
  digest_block
  dkSkDU5xwADj+B7HhomnCn9A8tc=
  end_digest_block
```

Copyright © 2003 Cadence Design Systems, Inc. All rights reserved.
3.2 Processing protected envelopes

Two modes of processing are defined for protected envelopes. Envelope encryption is the process of recognizing encryption envelopes in the source text and transforming them into decryption envelopes. Envelope decryption is the process of recognizing decryption envelopes in the input text and transforming them into the corresponding clear text for the parsing step that follows.

3.2.1 Encryption

VHDL tools that provide encryption services shall transform source text containing encryption envelopes. The tool replaces each encryption envelop with a decryption envelope by encrypting the source text according to the specified pragmas. Source text which is not contained in an encryption envelope shall not be modified by the encrypting language processor.

In the encryption block if the data pragmas (data_keyname, data_keyowner, data_method) are defined, the specified key and algorithm are used along with the session key to encrypt the data. If these pragmas are absent, a random session key is generated and used to encrypt the data. The encrypted data is enclosed in the `data_block` pragmas. This session key data (or the information about the session key) and the information about the encryption algorithm is encrypted by another key and is output between the `key_block` pragmas. This second key and the algorithm are specified by the key pragmas. If the key pragmas (key_method, key_owner, key_name) are absent in the encryption block, the tool’s internal key is used.

3.2.2 Decryption

VHDL tools that support compilation of encrypted data internally decrypt the decryption envelopes according to the specified pragma expressions.

4 Envelope Directives

Protected envelopes are specified as lexical regions delimited by protect pragma declarations. The semantics of a particular protect pragma declaration is specified by its pragma expressions. This standard reserves the keyword names listed in the following table for use as keywords to the protect pragma. These keywords are defined in section 6.1, with a specification of how each participates in the encryption and decryption processing modes. Some keywords are used exclusively in the encryption envelope, some are used exclusively in the decryption envelope, where as some are used in both kind of envelopes.
The following pragma keywords are relevant to encryption envelopes only:

- `<empty>` Opens a new encryption envelope
- `begin` Opens an input data block for encryption
- `end` Closes an encryption envelope

The following are used only in the decryption envelope:

- `begin_protected` Opens a new decryption envelope
- `end_protected` Closes a decryption envelope
- `key_block` Begins an encoded block of key data
- `end_key_block` Closes an encoded block of key data
- `data_block` Begins a block of encrypted data
- `end_data_block` Closes a block of encrypted data
- `digest_block` Begins an encoded block of authentication code data for data integrity
- `end_digest_block` Closes the authentication code
- `decrypt_license` Specifies licensing constraints on decryption
- `runtime_license` Specifies licensing constraints on simulation

The following are used both by the encryption and decryption envelopes.

- `encoding` Specifies the coding scheme for encrypted data
- `data_keyowner` Identifies the owner of the data encryption key
- `data_method` Identifies the data encryption algorithm
- `data_keyname` Specifies the name of the data encryption key
- `key_keyowner` Identifies the owner of the key encryption key
- `key_method` Specifies the key encryption algorithm
- `key_keyname` Specifies the name of the key encryption key
- `data_public_key` Specifies the public key for data encryption
- `data_decrypt_key` Specifies the session key for data decryption
- `viewport` Modifies scope of access into protected envelope
- `decrypt_license` Specifies licensing constraints on encryption
- `runtime_license` Specifies licensing constraints on simulation

The following pragma keywords are just informational in nature.

- `author` Specifies the author of an envelope
- `author_info` Specifies additional information about the author
- `encrypt_agent` Identifies the encryption service
- `comment` Uninterpreted documentation string

The scope of `protect` pragma declarations is completely lexical and not associated with any declarative region or declaration in the HDL text itself.

In the protection envelopes where a specific pragma keyword is absent, the VHDL tool shall use the default value. VHDL tools that perform encryption should explicitly output all relevant pragmas keywords (including the ones for which default values were used) for each envelope in order to avoid unintended interpretations during decryption.

### 4.1 Envelope encoding keywords

#### 4.1.1 begin

**Syntax**

```
begin
```

**Description**

ENCRYPTION INPUT: The `begin` pragma expression is used in the input text to indicate to an encrypting tool the point at which encryption begins. All text, including comments and other protect
pragmas, between the begin pragma expression and the corresponding end pragma expression is encrypted and is stored in the output format using the data_block pragma expression.

Nesting of pragma begin/end blocks is not supported, although there may be begin_protected/end_protected blocks containing previously encrypted content inside such a block. They are simply treated as a byte stream and encrypted as if they were text.

ENCRYPTION OUTPUT: none
DECRIPTION INPUT: none

4.1.2 end

4.1.2.1 Syntax
end

4.1.2.2 Description
ENCRYPTION INPUT: The end pragma expression is used in the input clear text to indicate the end of the region that shall be encrypted

ENCRYPTION OUTPUT: none
DECRIPTION INPUT: none

4.1.3 begin_protected

4.1.3.1 Syntax
begin_protected

4.1.3.2 Description
ENCRYPTION INPUT: If found in an input file during encryption begin_protected/end_protected block and its contents are treated as input clear text. This could result from a situation where a previously encrypted model is being re-encrypted as a portion of a larger model. An additional requirement is that any other protect pragmas inside the begin_protected/end_protected block shall not be interpreted or override pragmas in effect. In this way, nested encryption will not corrupt pragma values in the current encryption in process.

ENCRYPTION OUTPUT: After encrypting a begin/end block during encryption, the encrypting tool produces a corresponding begin_protected/end_protected block in the output file. This block begins with the begin_protected pragma expression. Following begin_protected all pragma expressions required as encryption output shall be generated prior to outputting the end_protected pragma expression. In this way protected blocks are completely self-contained avoiding any undesired interaction when using multiple encrypted models during the decryption process.

Note that this does not begin a block of encrypted data or keys, the data_block and key_block pragma expressions are used for this purpose and they are found within a begin_protected/end_protected block.

DECRIPTION INPUT: The begin_protected pragma expression begins a previously encrypted region. A decrypting tool accumulates all the pragma expressions in the block for use in decryption of the block.
4.1.4 end_protected

4.1.4.1 Syntax
end_protected

4.1.4.2 Description

ENCRYPTION INPUT: This pragma expression indicates the end of a previous begin_protected block. This indicates that the block is complete and new pragma expression values shall be accumulated for the next envelope.

ENCRYPTION OUTPUT: The end_protected pragma expression shall be output to indicate the end of a protected block.

DECRYPTION INPUT: The end_protected pragma expression indicates the end of a set of pragmas that should be sufficient to decrypt the current block. Upon encountering end_protected a tool shall verify that all required information is present.

4.1.5 author

4.1.5.1 Syntax
author=<string>

4.1.5.2 Description

ENCRYPTION INPUT: The author pragma expression is used to indicate the name of the IP author. It should be given outside any begin/end block so that this information is transferred to clear text in the output file.

ENCRYPTION OUTPUT: The author pragma expression should be output in each protected block unchanged from the input.

DECRYPTION INPUT: none.

4.1.6 author_info

4.1.6.1 Syntax
author_info=<string>

4.1.6.2 Description

ENCRYPTION INPUT: The author_info pragma expression is provided to allow arbitrary information to be provided by the IP Author in the form of a string value. Its use is strictly optional and the contents are not required in any way during encryption or decryption.

ENCRYPTION OUTPUT: The author_info pragma expression should be output in each protected block unchanged from the input.

DECRYPTION INPUT: none

4.1.7 encrypt_agent

4.1.7.1 Syntax
encrypt_agent=<string>

4.1.7.2 Description

ENCRYPTION INPUT: none
ENCRYPTION OUTPUT: The `encrypt_agent` pragma expression should be output as clear text in each protected block. It takes a string value indicating the name of the encrypting tool. This is the tool vendor or tool being used to perform the encryption. This key-word is optional in all cases but may be included to document the toolset performing the encryption.

DECRYPTION INPUT: none

4.1.8 Encrypt_agent_info

4.1.8.1 Syntax
encrypt_agent_info=<string>

4.1.8.2 Description

ENCRYPTION INPUT: none
ENCRYPTION OUTPUT: The `encrypt_agent_info` pragma expression is provided to allow arbitrary information to be provided by the encrypting tool in the form of a string value. Its use is strictly optional and the content is not required in any way during encryption or decryption.

DECRYPTION INPUT: none

4.1.9 encoding

4.1.9.1 Syntax
encoding=<encoding_descriptor>

4.1.9.2 Description

ENCRYPTION INPUT: The `encoding` pragma expression specifies how pragma expressions and encrypted text shall be encoded. The encoding is necessary to ensure that this potentially binary data can be re-inserted into a text document without impairing the subsequent editing or transmission of the document. If an `encoding` pragma expression is present in the input stream it specifies how the output should be encoded. A tool may choose to encode the data even if no `encoding` pragma expression was found in the input stream and should output the corresponding `encoding` pragma expression.

The following sub-keywords values are specified for the value of the `<encoding_descriptor>` of the `encoding` pragma expression. Each of them are found in the pragma expression string value given as the `<encoding_descriptor>` and are separated by white space.

encoding=<encoding_type> - specifies the method for calculating the encoding.

<table>
<thead>
<tr>
<th>encoding_type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>raw</td>
<td>Identity transformation</td>
</tr>
<tr>
<td>uuencode</td>
<td>Method specified in IEEE 1003.1-2001 (uuencode Historical Algorithm)</td>
</tr>
<tr>
<td>RFC2045_base64</td>
<td>Base64 encoding method specified in IETF RFC 2045 (also IEEE 1003.1-2001 uuencode -m)</td>
</tr>
<tr>
<td>RFC2045_quoted-printable</td>
<td>Quoted-printable encoding method specified in IETF RFC 2045</td>
</tr>
<tr>
<td>RFC1113_printable</td>
<td>Method specified in RFC 1113</td>
</tr>
</tbody>
</table>

If `raw` then no encoding has been performed and the encoded data may contain non-printable characters. Further encoding methods may be added in future.

All compliant tools are expected to support at least the `RFC2045_base64` encoding mechanism. Default encoding mechanism can be tool specific.
line_length=<number> - this is the number of characters (after any encoding) in a line of the data_block. This allows the insertion of line breaks in the data_block after encryption and encoding to make embedding in ASCII formats simpler. Without the additional line breaks the data_block would typically exceed the line length requirements of commonly used editors (such as vi) and make the containing file not editable.

In the absence of a line_length keyword, a tool may use an implementation specific value for line_length. In such a case, the tool should output the corresponding line_length pragma expression.

ENCRIPTION OUTPUT: The encoding directive should be output in each begin_protected/end_protected block to explicitly specify the encoding used by the encrypt_agent.

The data_block, data_public_key, data_decrypt_key, digest_block, key_block, and key_public_key are all encoded using this encoding. If separate encoding is desired for each of these fields then multiple encoding pragma expressions can be given in the input stream prior to each of the above pragma expressions. In addition to sub-keywords specified for the value of the <encoding_descriptor> in the input text, the encrypting tool is expected to generate the following:

bytes=<number> - this is the number of bytes in the original block of data before any encoding or the addition of line breaks. The bytes value is added by the encrypting tool for each block that it encrypts.

DECRYPTION INPUT: During decryption, the encoding directive is used to find the encoding algorithm used and the size of actual data. The decoded data is then used for further processing.

4.1.10 data_keyowner

4.1.10.1 Syntax
   data_keyowner=<string>

4.1.10.2 Description

ENCRIPTION INPUT: The data_keyowner specifies the company or tool that is providing the keys used for encryption and decryption of the data. The keys might be provided by an IP Author, the encrypting tool, the IP consumer, or possibly even a third party distributor of the IP. It has to be a value which is available in the tool’s key database. If this pragma is absent the encrypting tool shall use its own embedded key. If specified, the tool reads the key from the database and uses this to encrypt the data block.

ENCRIPTION OUTPUT: The data_keyowner is encrypted with the key_method and found in the key_block.

DECRYPTION INPUT: During decryption, the data_keyowner is combined with the data_keyname to determine the appropriate secret/private key to use during decryption of the data_block.

4.1.11 data_method

4.1.11.1 Syntax
   data_method=<method_name>
4.1.11.2 Description

ENCRIPTION INPUT: The `data_method` pragma expression indicates the encryption algorithm that shall be used to encrypt subsequent `begin/end` block. The encryption method is an identifier that is commonly associated with a specific encryption algorithm.

This standard specifies the following values for the `data_method` pragma expression. Additional identifier values are implementation-defined:

- DES Data Encryption Standard
- RSA RSA Public Key
- RC2 RSA RC2
- RC4 RSA RC4
- RC5 RSA RC5
- RC6 RSA RC6

Editor's Note: The above list should be replaced with a normative reference to an existing registry of encryption algorithm identifiers. IETF and W3C are potential registries, and others may exist.

All compliant tools are expected to support at least the RC5 encryption algorithm. Default encryption algorithm can be tool specific.

ENCRIPTION OUTPUT: The `data_method` is encrypted with the `key_method` and found in the `key_block`.

DECRYPTION INPUT: The `data_method` indicates the algorithm that should be used to decrypt the `data_block`.

4.1.12 `data_keyname`

4.1.12.1 Syntax

```
data_keyname=<string>
```

4.1.12.2 Description

ENCRIPTION INPUT: The `data_keyname` pragma expression provides the name of the key or key pair that is used to decrypt the `data_block`. A given `data_keyowner` will typically have multiple keys that they have shared in different ways with different vendors or customers. This pragma expression indicates which of these many keys has been used.

ENCRIPTION OUTPUT: When a `data_keyname` is provided in the input, it indicates the key that is to be used for encrypting the data. The encrypting tool must be able to combine this pragma expression with the `data_keyowner` and determine the key to use. The `data_keyname` is encrypted using `key_method` and encoded in the `key_block`.

DECRYPTION INPUT: In use models where the `data_keyowner` has provided a secret/private key to a Tool Vendor, or a Tool Vendors secret key has been used, then a unique key name must be identified for each key during this exchange. This key name is then used to identify at decryption time which of many possible secret keys for a given key owner should be used for decryption.
4.1.13 data_public_key

4.1.13.1 Syntax
data_public_key=<key>

4.1.13.2 Description
ENCRYPTION INPUT: The `data_public_key` pragma expression indicates that the next line of the file contains the encoded value of the public key, preceded by the single line comment prefix. This is the public key that should be used to encrypt the data. The encoding is specified by the `encoding` pragma expression that is currently in effect. If both `data_public_key` and `data_keyname` are present then they must refer to the same key.

ENCRYPTION OUTPUT: The `data_public_key` pragma expression should be output in each protected block for which it is used, followed by the encoded value. The `data_method` and `data_public_key` can be combined to fully specify the required encryption.

DECRYPTION INPUT: The `data_keyowner` and `data_method` can be combined with the `data_public_key` to determine if the decrypting tool knows the corresponding private key to decrypt a given `data_block`. If the decrypting tool can compute the required key the model can be decrypted (if licensing allows it).

4.1.14 data_decrypt_key

4.1.14.1 Syntax
data_decrypt_key=<key>

4.1.14.2 Description
ENCRYPTION INPUT: The `data_decrypt_key` indicates that the next line contains the encoded value of the key that will decrypt the `data_block`. This pragma expression should only be used when digital signatures are used. An IP author can generate a key and use it to encrypt the clear text. This encrypted text is then stored in the output file as the `data_block`. Then the `data_method` and `data_decrypt_key` are encrypted using the key_method and stored in the output file as the contents of the `key_block`. Note that the `data_block` itself is not re-encrypted, only the information about the data key is.

ENCRYPTION OUTPUT: The `data_decrypt_key` is output as part of the encrypted content of the `key_block`. The value is encoded as specified by the `encoding` pragma expression.

DECRYPTION INPUT: Upon determining that a digital signature was in use for given protected region, the decrypting tool must decrypt the `key_block` to find the `data_decrypt_key` and `data_method` which in turn can be used to decrypt the data block.

4.1.15 data_block

4.1.15.1 Syntax
data_block

4.1.15.2 Description
ENCRYPTION INPUT: A `data_block` should never be found in an input file unless it is contained within a previously generated `begin_protected/end_protected` block in which case it is ignored.

ENCRYPTION OUTPUT: The `data_block` pragma expression indicates that a data block begins on the next line in the file. An encrypting tool takes each `begin/end` block, encrypts the contents as specified by the `data_method` pragma expression, and then encodes the block. The resultant text is generated as the output.
DECRYPTION INPUT: The **data_block** is first read in the encoded form. The encoding is reversed, and then the block should be decrypted in-memory for consumption.

4.1.16  **digest_block**

4.1.16.1 Syntax

    digest_block

4.1.16.2 Description

ENCRYPTION INPUT: none

ENCRYPTION OUTPUT: A Message Authentication Code (MAC) is used to ensure that the IP has not been modified. In Message Authentication Code, the encrypting tool generates the message digest (fixed length, computationally unique identifier corresponding to a set of data). The message digest is generated for both data_block and the key_block.

DECRYPTION INPUT: In order to authenticate the message, the consuming tool shall first decrypt the message, then generate the message digest on the original message, and compare the two message digests. If the two don’t match this means that either the MAC or **data_block** or the key_block has been altered, and the tool can error out.

4.1.17  **key_keyowner**

4.1.17.1 Syntax

    key_keyowner=<string>

4.1.17.2 Description

ENCRYPTION INPUT: The **key_keyowner** specifies the company or tool that is providing the keys used for encryption and decryption of the key information. The value of the **key_keyowner** also has the similar constraint as mentioned in the **data_keyowner** values.

ENCRYPTION OUTPUT: The **key_keyowner** should be unchanged in the output file.

DECRYPTION INPUT: During decryption, the **key_keyowner** can be combined with the **key_keyname** to determine the appropriate secret/private key to use during decryption of the **key_block**.

4.1.18  **key_method**

4.1.18.1 Syntax

    key_method=<method_name>

4.1.18.2 Description

ENCRYPTION INPUT: The **key_method** pragma expression indicates the encryption algorithm that shall be used to encrypt the keys used to encrypt the **data_block**. The same names and formats are used for **data_method** and **key_method**. The values have the same constraint as mentioned for the **data_method** values.

ENCRYPTION OUTPUT: The **key_method** remains unchanged in the output file.
DECRIPTION INPUT: The **key_method** indicates the algorithm that shall be used to decrypt the **key_block**.

### 4.1.19 key_keyname

#### 4.1.19.1 Syntax

```
key_keyname=<string>
```

#### 4.1.19.2 Description

**ENCRYPTION INPUT:** The **key_keyname** pragma expression provides the name of the key or key pair that should be used to decrypt the **key_block**. A given **key_keyowner** will typically have multiple keys that they have shared in different ways with different vendors or customers. This pragma expression indicates which of these many keys has been used.

**ENCRYPTION OUTPUT:** When a **key_keyname** is provided in the input, it indicates the key that shall be used for encryption of the data encryption keys. The encrypting tool must be able to combine this pragma expression with the **key_keyowner** and determine the key to use. The **key_keyname** itself should be output as clear text in the output file.

**DECRYPTION INPUT:** In use models where the **key_keyowner** has provided a secret/private key to a Tool Vendor, or a Tool Vendors secret key has been used, a unique key name must be identified for each key during encryption. This key name is then used to identify at decryption time which of the many possible secret keys for a given key owner shall be used for decryption.

### 4.1.20 key_block

#### 4.1.20.1 Syntax

```
key_block
```

#### 4.1.20.2 Description

**ENCRYPTION INPUT:** A **key_block** shall never be found in an input file unless it is contained within a previously generated **begin_protected/end_protected** block in which case it is ignored.

**ENCRYPTION OUTPUT:** The **key_block** pragma expression indicates that a key block begins on the next line in the file. An encrypting tool takes **data_method**, **data_keyname** and **data_keyowner** to form a text buffer. This buffer is then encrypted with the appropriate **key_method**, **key_keyname** and **key_keyowner**. Then the encrypted region is be encoded. The output of this encoding shall be generated as the contents of the **key_block**.

Where more than one **key_block** pragma expression occurs within a single **begin/end** block, the generated key blocks shall all encode the same data decryption key data. Multiple key blocks are specified for the purpose of providing alternative decryption keys for a single decryption envelope.

**DECRYPTION INPUT:** The **key_block** is first read. The encoding is reversed and then the block internally decrypted. The resulting text can now be parsed to determine the keys required to decrypt the **data_block**. If for a **key_block** the specified key is not available, the tool should try the subsequent **key_blocks** for availability.
4.1.21 Decrypt_license

4.1.21.1 Syntax
decrypt_license=<library_name:entry_point_name:string_parameter[:exit_point_name>]

4.1.21.2 Description

ENCRYPTION INPUT: The decrypt_license pragma expression will typically be found inside a begin/end pair in the original clear text. This is necessary so that it is encrypted in the output IP shipped to the end user.

ENCRYPTION OUTPUT: The decrypt_license is output unchanged in the output description except for encryption and encoding of the pragma exactly as other clear text in the begin/end pair. Note that typically it will be output in the data_block.

DECRYPTION INPUT: After encountering a decrypt_license pragma expression in an encrypted model, prior to processing the decrypted text, the application should load the specified library and call the function indicated by the given entry_point_name, passing it the string_parameter specified. This routine should then return a 0 if the application is licensed to decrypt the model and non-zero if the application is not licensed to decrypt the model. The non-zero value should be printed in any error message about the failure of licensing. If an exit_point_name is specified then it should be called prior to exiting the decrypting application to allow for releasing the license.

Note that this only provides marginal security because the end-user of the model has the shared library and could use readily available debuggers to debug the calling sequence of the licensing mechanism. They could then produce an equivalent library that returns a 0 but avoids the license check.

4.1.22 runtime_license

4.1.22.1 Syntax
runtime_license=<library_name:entry_point_name_name:string_parameter>[:exit_point_name>]

4.1.22.2 Description

ENCRYPTION INPUT: The runtime_license pragma expression will typically be found inside a begin/end pair in the original clear text. This is necessary so that it is encrypted in the output IP shipped to the end user.

ENCRYPTION OUTPUT: The encrypt_license is output unchanged in the output description except for encryption and encoding of the pragma exactly as other clear text in the begin/end pair.

DECRYPTION INPUT: After encountering a runtime_license pragma expression in an encrypted model, prior to executing, the application should load the specified library and call the function indicated by the given entry_point_name, passing it the string_parameter specified. This routine should then return a 0 if the application is licensed to execute the model and non-zero if the application is not licensed to execute the model. The non-zero value should be printed in any error message about the failure of licensing. If an exit_point_name is specified then it should be called prior to exiting the executing application to allow for releasing the license. Note that execution could mean anything from simulation to layout to synthesis.
Note that this only provides marginal security because the end-user of the model has the shared library and could use readily available debuggers to debug the calling sequence of the licensing mechanism. They could then produce an equivalent library that returns a 0 but avoids the license check.

4.1.23 comment

4.1.23.1 Syntax
comment=<value>

4.1.23.2 Description

ENCRYPTION INPUT: The comment pragma expression can be found anywhere in an input file and indicates that even if this is found inside a begin/end block the value should be output as a comment in clear text in the output immediately prior to the data_block. This is provided so that comments that may and up being included in other files inside a begin/end block can protect themselves from being encrypted. This is important so that critical information such as copyright notices can be explicitly excluded from encryption. Since this constitutes known clear text that would be found inside the data_block the pragma itself and the value should not be included in the encrypted text.

ENCRYPTION OUTPUT: The entire comment including the beginning pragma should be output in clear text immediately prior to the data_block corresponding to the begin/end in which the comment was found.

DECRYPTION INPUT: none.

4.1.24 viewport

4.1.24.1 Syntax
viewport=<object_name>:<access>

4.1.24.2 Description

The viewport pragma expression describes objects within the current protected envelope for which access should be permitted by the VHDL tool. The specified object name shall be contained within the current envelope. The access value is an implementation specified relaxation of protection

5 Appendix A

5.1 Encryption/Decryption Flow

This section describes the various scenarios which can be used for IP Protection, and it also shows how to achieve the desired effect of securely protecting, distributing, and decrypting the model.
The data that needs to be protected from access or from unauthorized modification, should be placed in within the protect **begin/end** block. As the tool encrypts all the information in the **begin/end** block, the information is also protected.

### 5.2 Tool Vendor Secret key encryption system

In the secret key encryption system the key is tool vendor proprietary and will be embedded within the tool itself. The same key is used for both encryption and decryption. (In the EDA domain this is the simplest scenario and is roughly equivalent to the existing Verilog-XL ’protect technique). It has the drawback of being completely tool vendor specific. Using this technique, the IP author can encrypt the IP and any IP consumer with appropriate licenses and the same tool vendor can utilize the IP.

If the key pragma are absent in the encryption block, the tool uses its internal key to encrypt the data block. As usual the session is specified by the data pragmas, i.e. data pragmas are specified the mentioned key is used, otherwise a random key is generated to encrypt the data.

### 5.3 Digital Envelopes

Editor’s Note: This is the preferred exchange form in that it permits use of session keys to limit the amount of cipher text exposure for the exchanged encryption keys. The following text is incorrect in the assumption that asymmetric algorithms are the only useful exchange key mechanisms.

In this mechanism, each user will have a public and private key. The public key is made public while the private key remains secret. The sender encrypts the message using a symmetric key encryption algorithm, then encrypts the symmetric key using the recipient’s public key. The recipient then decrypts the symmetric key using the appropriate private key and then decrypts the message with the symmetric key. In this way a fast encryption methods processes large amount of data, yet secret information is never transmitted without encryption. In digital envelopes, using the above encryption technology (secret key encryption system, where the key will be given by the IP author/end user), encryption tool will protect the IP. This symmetric key and algorithm information is them encrypted with a public key, the corresponding private key of which is available to the tool. So only the tool can decrypt the symmetric key internally and decrypt the protected IP.

Instead of using the public key of public/private key pair, a tool specific embedded key can also be used to encrypt the **key_block**. In this case also as only the tool knows its embedded key, only it can internally decrypt the design, hence the same effect can be achieved.

The **data_method** and **data_keyowner/data_keyname** are used to encrypt the **data_block**. The encrypting tool then encrypts the **data_keyowner** and **data_keyname** pragmas with the **key_keymethod/key_keyname** and puts them in the **key_block** along with **data_method**. Alternatively if a dynamic session key is generated, the session key itself is encrypted along with the data method and put in the key block.

In the first approach the **data_keyowner/data_keyname** should also be present with the decrypting tool. No such dependency exists with the second approach as the key is present in the file itself.

For better security in the first approach the encrypting tool can actually read the **data_keyowner/data_keyname** key and put it in the **key_block** as **data_decrypt_key**. Which not only will remove the dependency mentioned above, but will also protect against the hit & trial breaking of the **data_block** with the existing keys at the IP users end.