8. Additional Requirements

8.1 General Discussion

In additional to the functional requirements in Section 7, other requirements are levied on the implementation and use of a DRBG. These requirements are associated with the DRBG boundary, the state of a DRBG, the seed, any key that is used by a given DRBG, and any additional input that is provided during operation. In addition, a discussion on prediction resistance and backtracking resistance in relation to the DRBGs specified in Section 10 is provided.

8.2 DRBG Boundary

A DRBG **shall** be encapsulated within a DRBG boundary. The boundary may be either physical or conceptual. Within the DRBG boundary,

1. The DRBG internal state and the operation of the DRBG functions **shall** only be affected according to the DRBG specification.
2. The DRBG internal state **shall** exist solely within the DRBG boundary.
3. Information about secret parts of the DRBG internal state and intermediate values in computations involving these secret parts **shall not** affect any information that leaves the DRBG boundary, except as specified for the DRBG pseudorandom bit outputs. The internal state **shall** be contained within the DRBG boundary and **shall not** be accessible from outside the boundary.

When a DRBG is implemented within a FIPS 140-2 cryptographic module, the DRBG

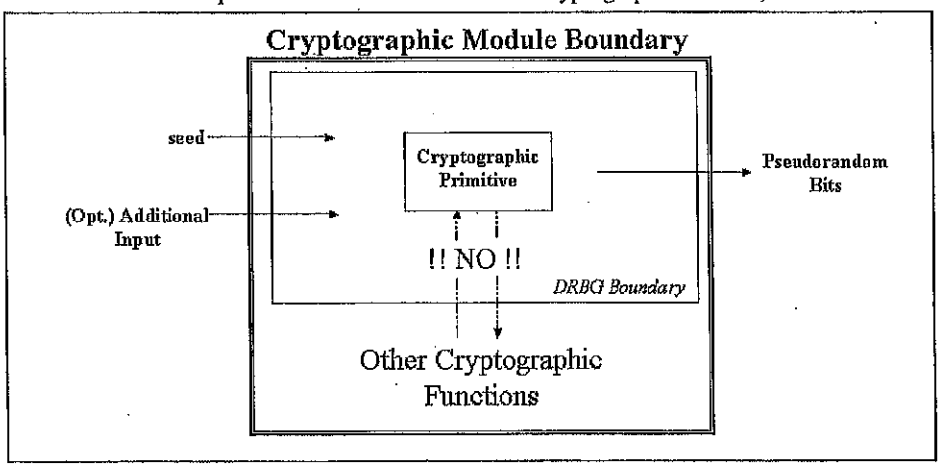

Figure 3: DRBG Boundary Contained within a Cryptographic Module Boundary

boundary **shall** be either fully contained within the cryptographic boundary or **shall** be coincident with the cryptographic boundary of the DRBG.

Figure 3 depicts the DRBG boundary being fully contained within the cryptographic module boundary. The figure shows a generalized DRBG that contains a cryptographic primitive. This design will provide higher assurance of correct operation than a design with a coincident DRBG and cryptographic module boundary. A cryptographic primitive within the DRBG boundary (e.g., a hash function) **shall not** be accessible for other purposes by a function outside the DRBG boundary; observe the !!NO!! across the dotted vertical arrows in the figure. For example, a digital signature function that is within the cryptographic boundary, but not within the DRBG boundary **shall not** use a function (e.g., the hash function) that resides within the DRBG boundary. A separate hash function is required outside the DRBG boundary, but within the cryptographic module boundary for digital signature purposes.

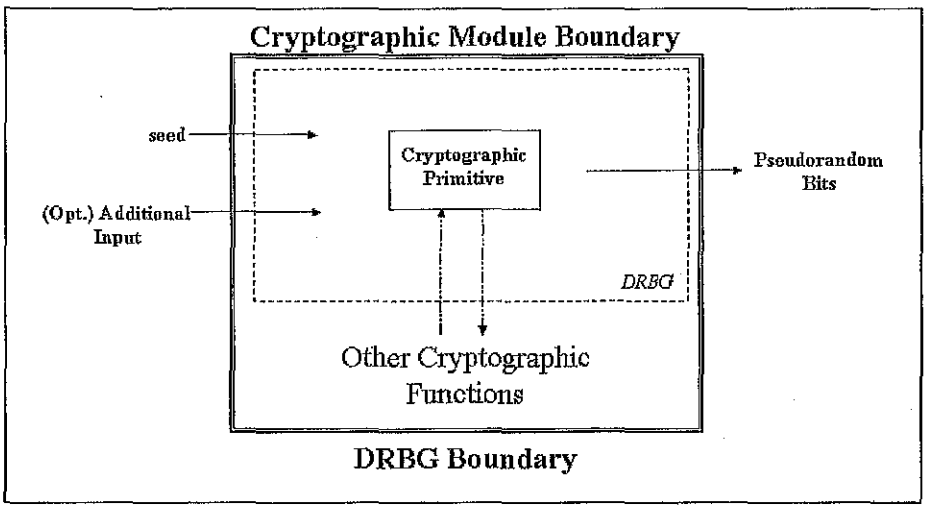

Figure 4: Coincident DRBG and Cryptographic Module Boundaries

Figure 4 depicts coincident DRBG and Cryptographic Module boundaries (i.e., the boundaries are identical). Note that the DRBG itself is contained within the dotted rectangle for convenience only; this is not intended to indicate the actual DRBG boundary. This design provides lesser assurance than the previous design because the internal state is potentially accessible by other non-DRBG functions. In this case, a cryptographic primitive within a DRBG boundary **may** be used by other cryptographic functions within the coincident DRBG and cryptographic module boundaries (e.g., during the generation or validation of digital signatures); observe the vertical dashed arrows to and from the DRBG's hash function. However, the internal state of the DRBG **shall not** be used or affected by other non-DRBG functions within the coincident boundary.

In both figures, the seed is shown as being input from outside the cryptographic module and DRBG boundaries. This is depicted as such for convenience only. The seed may actually be input from either inside or outside the two boundaries. In either case, however, the requirements for protecting and handling the seed are specified in Section 8.4.

8.3 DRBG Instantiation and the Internal State

A DRBG is instantiated for a usage class (i.e., one or more purposes; see Section 9.4) by initializing with a seed; an instantiation may subsequently be reseeded. Each seed defines an instance of the DRBG instantiation; an instantiation consists of one or more instances that begin when a new seed is acquired (see Figure 5). The period of time between seeding and reseeding is considered as the seed life.

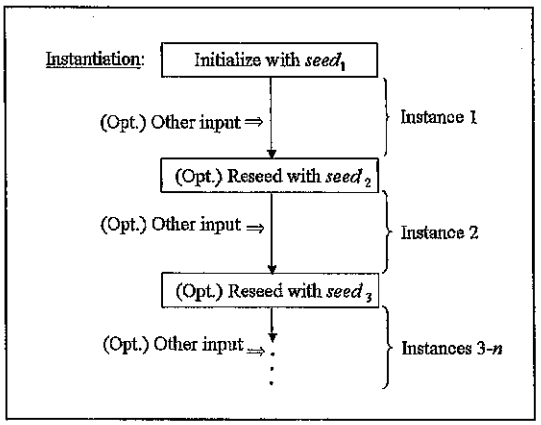

Figure 5: DRBG Instantiation

At any given time after a DRBG has been instantiated, a DRBG exists in a state that is defined by all prior input information.

Different DRBG instances are defined by the seed and any other initial input information that is required by a specific DRBG.

A DRBG **shall** be instantiated prior to the generation of output by the DRBG. During instantiation, an initial internal state (hereafter called just the state) is derived, in part, from a seed. The DRBG instantiation may subsequently be reseeded at any time (see Section 8.4 for a discussion on seeds).

Depending on the DRBG, the state includes:

1. The usage class of the DRBG instantiation,
2. One or more values that are derived from the seed(s); at least one of these derived values is updated during the operation of the DRBG (e.g., at least one component of the state is updated during each call to the DRBG),
3. Other information that is particular to a specific DRBG; this information may remain static or may be updated during the operation of the DRBG,
4. An indication of whether or not prediction resistance is to be provided by the DRBG upon request,
5. The security strength provided by the DRBG, and
6. A transformation of the entropy bits used to create the seed; this information remains static until replaced by new values during reseeding.

The state **shall** be protected at least as well as the intended use of the output bits by the consuming application. Each DRBG instantiation **shall** have its own state. The state for one DRBG instantiation **shall not** be used as the state for a different instantiation.

A DRBG **shall** transition between states on demand (i.e., when the generator is requested to provide new pseudorandom bits). A DRBG may also be implemented to transition in response to internal or external events (e.g., system interrupts) or to transition continuously (e.g., whenever time is available to run the generator). Additional unpredictability is introduced when the generator transitions between states continuously or in response to

external events. However, when the DRBG transitions from one state to another between requests, reseeding may need to be performed more frequently.

8.4 Seeds

When a DRBG is used to generate bits, a seed **shall** be acquired prior to the generation of output bits by the DRBG. The seed is used to instantiate the DRBG and determine the initial *state* that is used when calling the DRBG to obtain the first output bits.

The seed, seed size and the entropy (i.e., randomness) of the seed **shall** be selected to minimize the probability that the sequence of pseudorandom bits produced by one seed significantly matches the sequence produced by another seed, and reduces the probability that the seed can be guessed or exhaustively tested. Since this Standard does not require full entropy for a seed but does require sufficient entropy, the length of the seed may be greater than the entropy requirement (i.e., a seed with n bits of entropy may be longer than n bits in length).

The entry of entropy into a DRBG using an insecure method could result in voiding the intended security assurances. To ensure unpredictability, care **shall** be exercised in obtaining and handling the entropy bits used to create seeds. The seed and its use by a DRBG **shall** be generated and handled as follows:

1. Seed construction: A seed **shall** include entropy bits and **should** include a personalization string (see Figure 6). Note that it is possible, in some cases, that the entropy in the entropy bits may not be distributed across the sequence of bits.
   A personalization string need not be secret. Whether or not the personalization string is present, the resulting seed **shall** be unique. That is, when a personalization string is used, the combination of the entropy bits and the personalization string **shall** determine a unique seed; when a personalization string is not used, the entropy bits **shall** be statistically unique. Examples of data that may be included in a personalization string include a product and device number, user identification, date and timestamp, IP address, or any other information that provides assurance that two DRBG seeds are not inadvertently the same.
   The combination of the entropy bits and the optional personalization string is called the *seed material*. A derivation function **shall** be used to distribute the entropy in the entropy bits across the entire seed (e.g., the seed is not constructed with all the entropy on one end of the seed) whenever:

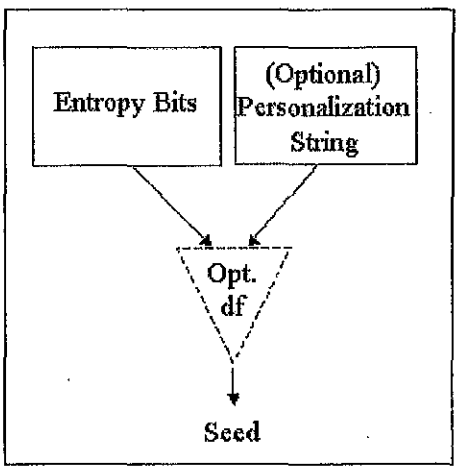

Figure 6: Seed Construction

   - A personalization string is used, or

- When a personalization string is not used, and entropy bits are not independent and uniformaly distributed throughout the entropy bit string.

2. Seed entropy: The entropy bits in the seed **shall** contain entropy that is appropriate for the desired level of security, and the entropy **shall** be distributed across the seed (e.g., by an appropriate derivation function).
   A consuming application may or may not be concerned about collision resistance between seeds. In order to accommodate possible collision concerns, a seed **shall** have entropy that is equal to or greater than 128 bits or the required security strength for the consuming application, whichever is greater (i.e., *entropy* ≥ **max** (128, *security_strength*)).
   Table 1 identifies the five security strengths to be provided by Approved DRBGs, along with the associated entropy requirements. If a selected DRBG and seed are not able to provide the required strength required by the consuming application, then a different DRBG and seed **shall** be used.

Table 1: Minimum Entropy and Seed Size

Bits of Security Strength	80	112	128	192	256
Minimum entropy	128	128	128	192	256

3. Seed size: The minimum size of the seed depends on the DRBG and the security strength required by the consuming application. See Section 10.
4. Entropy input for a seed: The entropy input for a seed **shall** provide the required amount of entropy for that seed in order to support the appropriate security level for a consuming application. The source of the entropy input **may** be an Approved NRBG, an Approved DRBG (or chain of Approved DRBGs) that is seeded by an Approved NRBG, or another source whose entropy characteristics are known. Further discussion about the entropy source is provided in Section 7.2.3.
   When entropy input is not readily obtainable for multiple requests for entropy input (e.g., multiple seeds are required), but sufficient entropy is available for a single DRBG (e.g., DRBG A), this DRBG **may** be used to provide entropy input for other DRBGs. In this case, the entropy provided to the first DRBG (i.e., DRBG A) **shall** be equal to or greater than the entropy requirement of any lower level DRBG. For example, DRBG A could provide entropy input for DRBGs B, C and D. The highest level DRBG (i.e., DRBG A) **may**, in fact, be used to provide entropy input for a chain of DRBGs. For example, DRBG A could provide entropy input for DRBG B, which in turn could be used to provide entropy input for DRBG E.

   An entropy input source need not be co-located with the DRBG. For example, for smart card applications, the entropy input could be generated from an NRBG that is not resident on the smart card and loaded onto the card to be combined with any personalization string to produce the seed.
5. Seed privacy: Seeds **shall** be handled in a manner that is consistent with the security required for the data protected by the consuming application. For example, if the only secrets in a cryptographic system are the keys, then the seeds used to generate keys **shall** be treated as if they are keys.

6. Reseeding: Reseeding (i.e., replacement of one seed with a new seed) is a means of recovering the secrecy of the output of the DRBG if a seed or the internal state becomes known. Periodic reseeding is a good countermeasure to the potential threat that the seeds and DRBG output become compromised. In some implementations (e.g., smartcards), an adequate reseeding process may not be possible. In these cases, the best policy might be to replace the DRBG, obtaining a new seed in the process (e.g., obtain a new smart card).
   Generating too many outputs from a seed (and other input information) may provide sufficient information for successfully predicting future outputs unless prediction resistance is provided (see Section 8.7). Periodic reseeding will reduce security risks, reducing the likelihood of a compromise of the data that is protected by cryptographic mechanisms that use the DRBG.

   Seeds **shall** have a specified finite seedlife. The seed **shall** be replaced periodically, or prediction resistance **shall** be provided or the DRBG **shall** be rendered inoperable at the end of the seedlife. If seeds become known (i.e., the seeds are compromised), unauthorized entities may be able to determine the DRBG output. Reseeding of the DRBG (i.e., creating a new DRBG instance) **shall** be performed in accordance with the specification for the given DRBG[1]. The DRBG reseed specifications within this standard are designed to produce a new seed that is determined by both the old seed and newly-obtained entropy bits that will support the desired security level. The newly-obtained entropy bits **shall** be checked to assure that they are not the same as the entropy bits obtained to create the previous DRBG instance. More than one set of entropy bits **shall not** be saved by the DRBG. The entropy bits **shall not** be saved in their original form, but **shall** be transformed by a one-way process (see the specifications in Section 10). When new entropy bits are generated and compared to the "old" entropy bits (i.e., the new entropy bits are transformed and compared with the transformed old entropy bits), the transformed new entropy bits **shall** replace the old transformed entropy bits in memory. If the new entropy bits are determined to be identical to the old entropy bits, then the DRBG **shall** fail.
   It should be noted that an alternative to reseeding is to create an entirely new instantiation. This may be appropriate, for example, in environments with restricted capabilities, where the seed is obtained from a source that is not co-located with the DRBG (e.g., in a smart card applicaton).

   Comment [ebb1]: Page: 1
   Should prediction resistance be mentioned here ?

7. Seed use: DRBGs **may** be used to generate both secret and public information. In either case, the seed **shall** be kept secret. A single instantiation of a DRBG **should not** be used to generate both secret and public values. However, cost and risk factors must be taken into account when determining whether different instantiations for secret and public values can be accommodated.

---

[1] For some applications (e.g., smart cards), reseeding essentially consists of replacing the entire DRBG module. Therefore, the restrictions within this paragraph **may not** apply.[NEED TO ANALYZE THIS STATEMENT].

A seed that is used to initialize one instantiation of a DRBG **shall not** be intentially used to reseed the same instantiation or used as a seed for another DRBG instantiation.
A DRBG **shall not** provide output until a seed is available, and the state has been initialized.

8. Seed separation: Seeds used by DRBGs **shall not** be used for other purposes (e.g., domain parameter or prime number generation).
It is recommended that when resources permit (e.g., storage capacity), different (i.e., statistically unique) seeds **should** be used for the generation of different types of random data (i.e., the instantiations of the DRBGs **should** be different). For example, the seed used to generate public values **should** be different than the seed used to generate secret values. The seed used by a DRBG technique to generate asymmetric key pairs **should** be different than a seed used by the same (or a different) DRBG technique to seed other DRBGs, which **should**, in turn, be different than a seed used by the same (or a different) DRBG technique to generate symmetric keys. The seed used by a DRBG technique to generate random challenges **should** be different than the seed used by the same (or a different) DRBG technique to generate PINS or passwords. However, the amount of seed separation is a cost/benefit decision.

8.5 Keys

Some DRBGs require the use of one or more keys. Such DRBGs are designed to generate keys from seeds (see Section 8.4, item 1 for a discussion on seed construction). A key and its use in a DRBG **shall** conform to the following:

1. Key entropy: The seed for the key **shall** have entropy that is equal to or greater than 128 bits or the required security strength of the consuming application, whichever is greater (i.e., $entropy \geq \max(128, security\_strength)$).
2. Key size: Key sizes **shall** be selected to support the desired security strength of the consuming application (see SP 800-57). If the DRBG primitive using the key (e.g., the block cipher algorithm) cannot support the required security strength, then a different primitive or a different DRBG **shall** be used.
3. Entropy source for a key: The entropy source for the key is the seed for the DRBG instance(see Section 8.4, item 4).
4. Key secrecy: Keys **shall** remain secret and **shall** be handled in a manner that is consistent with the security required for the data protected by the consuming application using the DRBG pseudorandom bits. Keys **shall** be protected in accordance with [SP 800-57].
5. Rekeying: Rekeying (i.e., replacement of one key with a new key) is a means of recovering the secrecy of the output of the DRBG if a key becomes known. Periodic rekeying is a good countermeasure to the potential threat that the keys and DRBG output become compromised. However, the result from rekeying is only as good as the entropy source used to provide the new key. In some implementations (e.g., smartcards), an adequate rekeying process may not be possible, and rekeying may actually reduce security. In these cases, the best policy might be to replace the DRBG, obtaining a new key in the process (e.g., obtain a new smart card).

Generating too many outputs using a given key may provide sufficient information for successfully predicting future outputs when prediction resistance is not provided. Periodic rekeying will reduce security risks, reducing the likelihood of a compromise of the data that is protected by consuming applications that use the DRBG.

Keys **shall** have a specified finite keylife (i.e. a cryptoperiod). Keys **shall** be replaced periodically. Expired keys or keys that have been replaced **shall** be destroyed (see SP 800-57). If keys become known (e.g., the keys or seeds are compromised), unauthorized entities may be able to determine the DRBG output.

6. Key use: Keys **shall** be used as specified in a specific DRBG. A DRBG requiring a key(s) **shall not** provide output until the key(s) is available.
7. Key separation: A key used by a DRBG **shall not** be used for any purpose other than random bit generation. Different instantiations of a DRBG **shall** use different keys. Different instances of of the same instantiation of a DRBG **should** use different keys.

8.6 Additional Input

During each request for bits from a DRBG, the insertion of additional input is allowed. This input is optional and may be either secret or publicly known; its length and value are arbitrary (i.e., there are no restrictions on its length or content). The additional input allows less reliance on the seed. If the additional input is kept secret and has sufficient entropy, the input may be used to provide additional entropy for random bit generation and provide an ability to recover from the compromise of the seed or one or more states of the DRBG. Depending on the method for acquiring the input, the exact value of the input may or may not be known to the user or application. For example, the input could be derived directly from values entered by the user or application, or the input could be derived from information introduced by the user or application (e.g., from timing statistics based on key strokes), or the input could be the output of another DRBG or an NRBG.
Additional unpredictability for the DRBG may also be provided by reseeding the DRBG (see Section 8.4).

8.7 Prediction Resistance and Backtracking Resistance

Each of the DRBGs specified in Section 10 has been designed to provide prediction resistance and backtracking resistance when observed from outside the DRBG boundary, given that the observer does not know the seed, or any key or state values.
Figure 7 depicts the sequence of DRBG states that result from a given seed. Some subset of bits from each state are used to generate pseudorandom bits upon request by a user. The following discussions will use the figure to explain backtracking and prediction resistance. Suppose that the user wants assurance that an adversary cannot determine the pseudorandom bits produced from $State_x$.

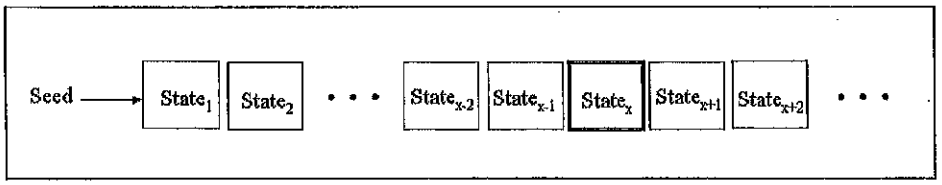

Figure 7: Sequence of DRBG States

Backtracking Resistance: When a DRBG provides backtracking resistance, an adversary is unable to determine the bits in $State_x$ if that adversary is able to determine the bits in any state occurring subsequent to $State_x$. That is, if $State_{x+1}$ (or any state after $State_{x+1}$) is compromised, the adversary is unable to "back up" the process to determine the bits in $State_x$. When observed from within the DRBG boundary (i.e., the DRBG is observed as a glass box, and the advsersary can get the current state (e.g., $State_{x+1}$)), the previous states cannot be determined. Each of the DRBGs in this Standard provide backtracking resistance, with the exception of the Hash_DRBG specified in Section 10.1.1.

Prediction Resistance: When prediction resistance is provided, an adversary is unable to determine the bits in $State_x$ if that adversary is able to determine the bits in any state prior to $State_x$. That is, if $State_{x-1}$ (or any state prior to $State_{x-1}$) is compromised, the adversary is unable to generate the next bits in the process and so (ultimately) to determine the bits in $State_x$. Note that an adversary will normally be able to determine the next bits if prediction resistance is not provided because of the deterministic nature of the DRBG. When observed from within the DRBG boundary (that is, as a glass box where the current state (e.g., $State_{x-1}$) is known), prediction resistance may be provided for a DRBG by the insertion of sufficient additional entropy prior to generating pseudorandom bits; for example, by doing an explicit reseed. Sufficient entropy is defined as being at least equal to the amount of entropy required for the seed used to instantiate the DRBG at the desired security strength (i.e., *min-entropy* = **max** (128, *strength*); see Section 8.4, item 2). Providing the additional entropy prior to generating new pseudorandom bits (i.e., generating a new state) isolates the newly generated bits from prior bits generated by the DRBG (i.e., from prior states); knowledge of previously generated bits (e.g., obtained via a compromise) does not allow the prediction of the new bits.

Note that prediction resistance is not provided if the entropy is obtained in amounts that are less than required to support the desired security level. Inserting insufficient additional entropy is better than not inserting additional entropy at all, but the DRBG cannot provide prediction resistance in this case.

9 General Discussion of the Specified DRBGs

9.1 Model of DRBG Interaction

Figure 8 depicts the use of a DRBG by an application. Prior to requesting pseudorandom bits, the application **shall** instantiate the DRBG using a seed. Depending on the application, the DRBG instantiation may need to be periodically reseeded using a new seed. These seeds are used to determine the initial state of the DRBG instantiation and the new state resulting from the reseed process. When the generation of pseudorandom bits is requested, the state is updated.

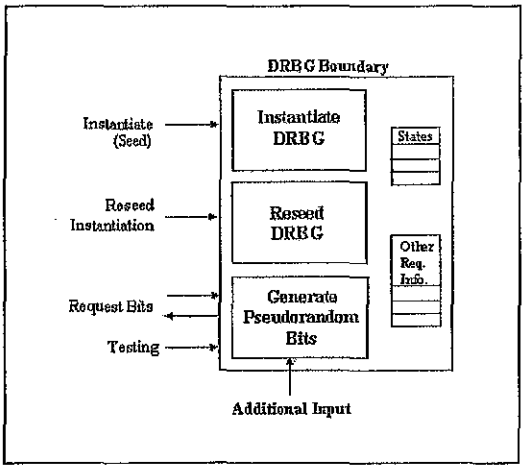

Figure 8: Using a DRBG

The discussions in the following subsections assume that a single type of DRBG will be available to an application. The availability and use of multiple DRBG types (e.g., both the **Hash_DRBG (...)** and the **Dual_EC_DRBG (...)**) is allowed, and the discussions may be extended to this case.

9.2 Security Strength Supported by a DRBG Instantiation

The DRBGs specified in this Standard support one or more of five security strengths (i.e., security levels): 80, 112, 128, 192 or 256 bits. The security strengths that may be supported by a particular DRBG are specified for each. However, the security strength actually supported by a particular instantiation may be less than the maximum security strength possible for that DRBG, depending upon the amount of entropy that is obtained in the seed.

The maximum strength provided by an instantiation is determined when the DRBG is instantiated. The instantiated security strength **shall** be less than or equal to the maximum security strength that can be supported by the DRBG

9.3 Effective Security Strength, Entropy and Seed Size of an Instantiation

The instantiation of a DRBG requires the acquisition of a seed with sufficient entropy to support the requested security strength; reseeding the instantiation requires the acquisition of another seed with the same properties. As discussed in Section 8.4, reseeding requires the acquisition of the appropriate amount of new entropy to support the desired security level and combining the newly-obtained entropy with the entropy from the previous instance.

The minimum new entropy (*min_entropy*) to be acquired when seeding or reseeding **shall** be equal to either 128 or the requested strength, whichever is greater (i.e., *min_entropy* = max (128, *requested_strength*).

Note that the use of more entropy than the minimum value will offer a security "cushion". The minimum size of the *seed* depends on the DRBG. Many DRBGs allow a range of seed sizes. A variation in the allowable seed size permits the use of an entropy source that provides either full entropy (i.e., one bit of entropy for each bit of the *seed*) or less than full entropy (i.e., multiple bits of the *seed* may be required to provide each bit of entropy).

9.4 DRBG Purposes and Usage Classes

A DRBG may be used to obtain pseudorandom bits for different purposes (e.g., DSA private keys and AES keys). This Standard recommends that different instantiations be used to generate bits for different purposes. However, if an application needs to generate bits for different purposes, it may not always be practical to use multiple instantiations. Each instantiation is associated with a *usage_class* for the purpose(s)supported by the instantiation. For example, a *usage_class* may be associated with the generation of only 1024-bit DSA keys, and a separate *usage_class* may be associated with the generation of 128-bit AES keys. Both *usage_classes* may use the same type of DRBG, but use different instantiations, or they may use different DRBG types (e.g., the generation of DSA keys may use the **Hash_DRBG (...)**, while AES keys may be generated using the **Dual_EC_DRBG (...)**). As another example, if an application cannot support multiple instantiations (e.g., because of memory restrictions), then the same *usage_class* could be associated with generating both 1024-bit DSA keys and 128-bit AES keys (i.e., the *usage_class* supports two purposes).

9.5 Security Strengths of an Instantiation for a Usage Class

For each DRBG instantiation, a security strength (i.e., security level) needs to be requested and obtained during the instantiation process. When a DRBG instantiation needs to provide pseudorandom bits for only one purpose (i.e., the *usage_class* is associated with that single purpose), then the security level needs to support that purpose. Examples:

1. 256-bit AES keys can provide a maximum of 256-bits of security. An instantiation must support at least 256 bits of security if the full 256 bits of security are to be provided by the AES keys. Note that the minimum entropy requirement would be 256 bits to support 256 bits of security.

2. 1024-bit DSA private keys can only provide 80 bits of security. In this case, an instantiation used only for the generation of 1024-bit DSA keys must be supported by at least 128 bits of entropy (see Section 9.3).

When an instantiation is used for multiple purposes (i.e., the *usage_class* is associated with more than one purpose), the minimum entropy requirement for each purpose must be considered. The DRBG needs to be instantiated for the highest entropy requirement (see Section 9.3). For example, if one purpose requires 80 bits of security (i.e., *min_entropy* = 128 bits), and another purpose requires 256 bits of security (i.e., *min_entropy* = 256 bits), then the DRBG **shall** be instantiated to support at least 256 bits of security (i.e., *min_entropy* = 256 bits).

9.6 Instantiating a DRBG

9.6.1 The Instantiation Function Call

Prior to the first request for pseudorandom bits, a DRBG **shall** be instantiated using a form of the following function call:

status = **Instantiate_DRBG** (*usage_class, requested_strength , prediction_resistance_flag, personalization_string, DRBG_specific_parameters*)

where:

1. *status* is the indication returned from the instantiation process. A *status* of Success indicates that the instantiation has been successful, and pseudorandom bits may be requested. Failure messages that could be returned from this process are specified for each DRBG. The *status* **shall** be checked to determine that the DRBG has been correctly instantiated.
2. **Instantiate_ DRBG (...)** is specified for each DRBG. Note that the name of the generalized function call of this section (i.e., **Instantiate_DRBG (...)**) is different than the specific name used for each DRBG (e.g., **Instantiate_Dual_EC_DRBG (...)**).
3. *usage_class* indicates the *usage_class* of the DRBG instantiation (e.g., to create DSA private keys). An indication of the *usage_class* is optional when the DRBG will never be used to support multiple usage classes; if the *usage_class* indicator is present, then the DRBG may be associated with multiple *usage_classes.*
4. *requested_strength* is used to request the minimum security strength for the instantiation. Note that DRBG implementations that support only one security strength do not require this parameter; however, any application using the DRBG must be aware of this limit.
5. The *prediction_resistance_flag* indicates whether or not prediction resistance may be required by the consuming application during one or more requests for pseudorandom bits. Note that DRBGs that are implemented to always or never support prediction resistance do not require this parameter. However, the user of a consuming application must determine whether or not prediction resistance may be required by the application.
6. The *personalization_string* is an optional input that is used to personalize a seed (see Section 8.4, item 1). If an implementation never intends to use a personalization string, then the parameter may be omitted.
7. The *DRBG_specific_parameters*, if any, are provided in Section 10 for each DRBG.

9.6.2 Request for Entropy

The DRBG specifications in this Standard **may** request bits from an entropy source during the instantiation and reseeding processes and in order to provide prediction resistance. This is specified in each specification as:

(*status*, *entropy_bits*) = **Get_entropy** (*min_entropy*, *min_length*, *max_length*), where

1. *status* is the status returned from the entropy source. In the DRBG specifications, either an indication of Success or Failure is expected as the returned *status*. The *status* **shall** be checked to determine that the requested entropy has been provided.
2. *entropy_bits* is the string of bits returned from the entropy source when the returned *status* = Success. For example, *x* might be used as the *seed* or used to derive the *seed*, depending on the DRBG. If the returned *status* = Failure, a Null string **shall** be returned.
3. *min_entropy* is the minimum amount of entropy to be returned in the bit string *x*. If an implementation always requires the same minimum entropy, this parameter may be omitted.
4. *min_length* is the minimum length of the bit string to be returned as *x*. Note that *min_length* is determined either by the value of *min_entropy* or by the DRBG design requirements. If an implementation always requires the same minimum length, this parameter may be omitted.
5. *max_length* is the maximum length of the bit string to be returned as *x*. Some of the DRBGs have a maximum length requirement in their design. Other DRBGs have no such restriction; in this case, the maximum length is an implementation issue and is denoted as *implementation_choice* in the specification. If an implementation always requires the same maximum length, this parameter may be omitted.

For implementations where the *min_length* is always the same as the *max_length*, the two parameters may be expressed as a single parameter (e.g., the call would be (*status*, *x*) = **Get_entropy** (*min_entropy*, *length*)).

The specific details of the **Get_entropy (...)** process are left to the implementer, with the above restrictions and any other entropy source requirements in this Standard (see Sections 7.2.1, 8.4 and 8.5).

9.6.4 Derivation Functions

9.6.4.1 Introduction

Derivation functions are used during DRBG instantiation and reseeding to either derive state values or to distribute entropy throughout a bit string. Two methods are provided. One method is based on hash functions and is used when the DRBG is based on hash functions (e.g., **Hash_DRBG (...)**); the other method is based on the block cipher algorithm used by a given DRBG (e.g., [redacted]_DRBG (...) using AES).

9.6.4.2 Derivation Function Using a Hash Function

The hash-based derivation function hashes an input string and returns the requested number of bits. Let **Hash (...)** be the hash function used by the DRBG, and let *outlen* be its output length. Note that the *requested_bits* string **shall not** be greater than (255 × *outlen*) bits in length (i.e., *no_of_bits_to_return* ≤ (255 × *outlen*)). The following or an equivalent process **shall** be used to derive the requested number of bits.

Hash_df (...):

Input: bitstring *input_string*, integer *no_of_bits_to_return*.
Output: bitstring *requested_bits*.
Process:

1. *temp* = the Null string.
2. $len = \left\lceil \frac{no\_of\_bits\_to\_return}{outlen} \right\rceil$.
3. *counter* = an 8 bit binary value represented in hexadecimal as x'01'.
4. For *i* = 1 to *len* do
   4.1 *temp* = *temp* || **Hash** (*counter* || *no_of_bits_to_return*, *input_string*).
   4.2 *counter* = *counter* + 1.
5. *requested_bits* = Leftmost (*no_of_bits_to_return*) of *temp*.
6. **Return** (*requested_bits*).

Comment [ebb2]: Page: 1
We need to specify an integer to string conversion process.

9.6.4.3 Derivation Function Using a Block Cipher Algorithm

9.6.4.3.1 The TDEA_df (...)Derivation Function

The **TDEA_df (...)** function derives bits from an input string using the TDEA block cipher algorithm, a derivation key and an Approved key wrapping algorithm (**TDEA_Wrap (...)**). **TDEA_Wrap(...)** is define in ANSI X9.102. Note that two key and three key TDEA are specified. Two keys are presented in a 112-bit string; three keys are presented in a 168-bit string.

The following or an equivalent process **shall** be used to derive the requested number of bits.

TDEA_df (...):

Input: integer *keylen*, bitstring (*derivation_key*, *M*), integer *no_of_bits_to_return*.
Output: string *status*, bitstring (*requested_bits*).
Process:

Comment: Parse the *derivation_key* into three TDEA keys (see below).

1. (*status*, *key*1, *key*2, *key*3) = **Parse_TDEA_Key** (*derivation_key*).
2. If (*status* = "Failure"), then **Return** ("Invalid *Key* size", Null).

   Comment: Wrap *M* using the three TDEA keys; the ciphertext string is returned as *C*.

3. If (*no_of_bits_to_return* > **len** (*M*)), then **Return** ("Too many bits requested from **TDEA_df**", Null).
4. *C* = **TDEA_Wrap** (*key*1, *key*2, *key*3, (*no_of_bits_to_return* || *M*)).

Comment [ebb3]: Page: 1
We need to specify an integer to string conversion process.

5. *requested_bits* = Leftmost (*no_of_bits_to_return*) of *C*.
6. **Return** ("Success", *requested_bits*).

Parse_TDEA_Key (...):

Input: bitstring *Key*.

Output: string *status*, bitstring (key_1, key_2, key_3).

1. *keylen* = **len** (*Key*).
2. If (*keylen* = 112), then do:
   - 2.1 key_1 = Leftmost 56 bits of *Key*.
   - 2.2 key_2 = Rightmost 56 bits of *Key*.
   - 2.3 key_3 = key_1.
   - 2.4 **Return** ("Success", key_1, key_2, key_3).
3. If (*keylen* = 168), then do:
   - 3.1 key_1 = Leftmost 56 bits of *Key*.
   - 2.2 key_2 = Bits 57-112 of *Key*.
   - 2.3 key_3 = Rightmost 56 bits of *Key*.
   - 2.4 **Return** ("Success", key_1, key_2, key_3).
4. **Return** ("Failure").

9.6.4.3.2 The AES_df (...)Derivation Function

The **AES_df (...)** function derives bits from an input string using the AES block cipher algorithm, a derivation key and an Approved key wrapping algorithm (**AES_Wrap (...)**). **AES_Wrap (...)** is define in ANSI X9.102. Note that AES keys may consist of 128, 192 or 256 bits.

The following or an equivalent process **shall** be used to derive the requested number of bits.

AES_df (...):

Input: integer *keylen*, bitstring (*derivation_key*, *M*), integer *no_of_bits_to_return*.

Output: string *status*, bitstring *requested_bits*.

Process:

1. If (*no_of_bits_to_return* > **len** (*M*)), then **Return** ("Too many bits requested from **AES_df**", Null).

   Comment: Get the ciphertext string *C*.

2. *C* = **AES_Wrap** (*derivation_key*, *keylen*, (*no_of_bits_to_return* || *M*)).
3. *requested_bits* = Leftmost (*no_of_bits_to_return*) of *C*.
4. **Return** ("Success", *requested_bits*).

Comment [ebb4]: Page: 1
Need to specify an integer to string conversion process.

9.7 Reseeding a DRBG Instantiation

9.7.1 Introduction

The reseeding of an instantiation is not required, but is recommended whenever an application and implementation are able to perform this process. Alternatively, a new DRBG instantiation may be created (see Section 9.6). Reseeding may also be performed in order to provide prediction resistance (see Section 8.7).

Reseeding will insert additional entropy into the generation process. Reseeding **may** be performed at the explicit request of the consuming application, perhaps on a periodic basis

determined by time. Reseeding **may** also be initiated by the DRBG when the maximum number of states have been generated or when prediction resistance is required. In either case, the DRBG **shall not** continue to produce bits until the DRBG is successfully reseeded.

9.7.2 The Function Call

When a DRBG instantiation is reseeded, the DRBG **shall** be reseeded using a form of the following function call:

status = **Reseed_ DRBG_Instantiation** (*usage_class*)

where:

1. *status* is the indication returned from the reseeding process. A *status* of Success indicates that the reseeding process has been successful, and pseudorandom bits may be requested. Failure messages that could be returned from this process are specified for each DRBG. The *status* **shall** be checked to determine that the DRBG has been correctly reseeded.
2. **Reseed__DRBG_Instantiation (...)** is specified for each DRBG. Note that the name of the generalized function call of this section (i.e., **Reseed_DRBG_Instantiation (...)**) is different than the specific name used for each DRBG (e.g., **Reseed_Hash_DRBG_Instantiation (...)**).
3. *usage_class* indicates the purpose of the DRBG instance (e.g., to create DSA private keys). An indication of the *usage_class* is optional when the DRBG will not be used to support multiple usage classes; if the *usage_class* indicator is present, then the DRBG may be associated with multiple *usage_classes*.

9.8 Generating Pseudorandom Bits Using a DRBG

9.8.1 Introduction

Each request for pseudorandom bits **shall** generate bits for only one value. For example, a single request **shall not** be used to generate bits for multiple AES keys, or bits for both an AES key and a DSA key). Instead, separate calls to the generation function **shall** be used.

Multiple requests **may** be used to construct a single value. For example, a 1024 bit pseudorandom string may be generated using eight calls for 128 bits each, and concatenating the eight 128-bit strings.

9.8.2 The Function Call

An application may request the generation of pseudorandom bits by a DRBG using a form of the following call:

(*status, pseudorandom_bits*) = **DRBG** (*usage_class, requested_no_of_bits, requested_strength, additional_input, prediction_resistance_flag*)

where:

1. *status* is the indication returned from **DRBG (...)**. A *status* of Success indicates that *pseudorandom_bits* have been successfully generated. Failure messages that could be returned from this process are specified for each DRBG. If an indication of failure is returned, a Null string is returned in place of the *pseudorandom_bits*. The status returned by the DRBG **shall** be checked by the consuming application to determine that the request has been successful prior to using any bit string returned.
2. *pseudorandom_bits* are returned when the *status* indicates Success. These are the bits requested by the application. If the status indicates a failure, a Null string **shall** be returned.
3. **DRBG (...)** is specified for each DRBG. Note that the name of the generalized function call of this section (i.e., **DRBG (...)**) is different than the specific name used for each DRBG (e.g., **Hash_DRBG (...)**).
4. *usage_class* indicates the purpose of the DRBG instance. An indication of the *usage_class* is optional when the DRBG will not be used to support multiple usage classes; if the *usage_class* indicator is present, then *usage_class* is used to select the appropriate instantiation to be reseeded.
5. *requested_no_of_bits* indicates the number of bits to be returned by the DRBG. If an application always requires the same number of pseudorandom bits to be returned, this parameter may be omitted.
6. *requested_strength* is used to request the minimum security strength for the pseudorandom bits to be generated. Note that this parameter is not required for implementations that provide only a single security strength. Note that the *requested_strength* parameter in the DRBG call is a failsafe mechanism. The implementation **shall** check that the value requested is not more than that provided by the instantiation, as determined by the call to the instantiation process (see Section 9.6.1). A call for greater strength **shall** result in an error condition .
7. Optional *additional_input* may to be provided. This parameter is not required for implementations that will never use additional_input.
8. *prediction_resistance_flag* indicates whether or not prediction resistance is to be provided for the pseudorandom bits to be generated (see Section 8.7). This parameter is not required if an implementation will always or never require prediction resistance.

9.9 Inserting Additional Entropy Between Requests

An implementation **may** insert additional entropy between requests for pseudorandom bits. This **may** be initiated by internal or external events. When additional entropy is inserted between requests for pseusorandom bits, it is recommended that such entropy not be inserted unless sufficient entropy is available to support the security strength of the instantiation (i.e., *entropy* ≥ **max** (128, *strength*)). The insertion of additonal entropy will result in an update of the state. If insufficient entropy is available, the implementation **may** choose to update the state or to exit the process without changing the state.

Comment [ebb5]: Page: 57
This needs to be incorporated, if we really want it.

Additional entropy **may** be inserted into the state of the **DRBG (...)** between requests for pseudorandom bits as follows:

(*status, entropy_bits*) = **Add_Entropy_to_ DRBG** (*usage_class, request_sufficient_entropy_flag, always_update_flag*)

where:

1. *status* is the indication returned from **Add_Entropy_to_ DRBG (...).** A *status* of Success indicates that additional entropy has been inserted. Failure messages that could be returned from this process are specified for each DRBG. The returned status **shall** be checked to determine whether or not *new_bits* have been returned.
2. *entropy_bits* contains the additional entropy that is acquired.
3. **Add_Entropy_to_DRBG (...)** is specified for each DRBG. Note that the name of the generalized function call of this section (i.e., **Add_Entropy_to_DRBG (...)**) is different than the specific name used for each DRBG (e.g., **Add_Entropy_to_Hash_DRBG_ (...)**).
4. *usage_class* indicates the purpose of the DRBG instance. An indication of the *usage_class* is optional when the DRBG will not be used to support multiple usage classes; if the *usage_class* indicator is present, then *usage_class* is used to select the appropriate DRBG instantiation for the add entropy process.
5. *request_sufficient_entropy_flag* indicates whether or not the new bits containing entropy are to be used if the entropy is insufficient to support the security strength of the instantiation.
6. *always_update_flag* indicates whether or not the state is to be updated when additional entropy is not available. This parameter is not required if the implementation is designed to always behave in the same way, i.e., always update the state whether or not additional entropy is available, or never update the state unless additional entropy is available.

9.10 Error Handling

[This section will contain guidance about the handling of error conditions by the consuming application]

9.11 DRBG Selection

Several DRBGs are specified in this Standard . The selection of a DRBG depends on several factors, including the security strength to be supported and what basic building blocks are available. An analysis of the consuming application's requirements for random numbers **shall** be conducted in order to select an appropriate DRBG.

10 DRBG Specifications

10.1 Deterministic RBGs Based on Hash Functions

10.1.1 Discussion

A hash DRBG is based on a hash function that is non-invertible or one way. The following are provided as DRBGs based on hash functions:

1. The **Hash_DRBG (...)** specified in Section 10.1.2 has been designed to use any Approved hash function and may be used by applications requiring various levels of security, providing that the appropriate hash function is used and sufficient entropy is obtained for the seed.
2. The **Keyed_Hash_DRBG (...)** specification in Section 10.1.3 is based on the **Hash_DRBG (...)** in Section 10.1.2, with the addition of a key.

10.1.2 Hash Function DRBG Using Any Approved Hash Function (Hash_DRBG)

10.1.2.1 Discussion

Figures 9 and 10 present a DRBG that uses any Approved hash function.

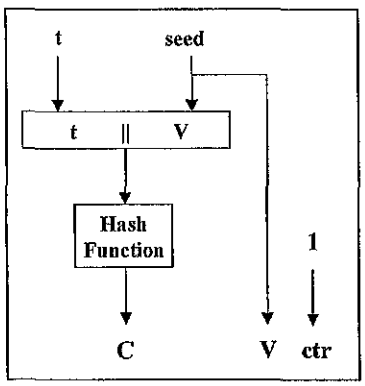

Figure 9: Hash_DRBG (...) Instantiation

Hash_DRBG (...) employs an Approved hash function that produces a block of pseudorandom bits using a seed (*seed*) and an application specific constant (*t*). Optional additional input (*additional_input*) may be provided during each access of **Hash_DRBG (...)** to obtain bits; the size of the *additional_input* is arbitrary.

The **Hash_DRBG (...)** requires the use of a hash function at three points in the process, including the instantiation and reseeding processes (see Figures 9 and 10). The same hash function **shall** be used at all three points. The hash function to be used **shall** meet or exceed the desired security strength of the consuming application.

Hash_DRBG (...) has been designed to meet different security levels, depending on the hash function used. The security strengths that can be accommodated by each hash function, the associated entropy requirement and the seed lengths are specified in Table 1. For each security *strength*, the required minimum entropy (*min_entropy*) **shall** be the maximum of 128 and the security *strength* (i.e., *min_entropy* = **max** (128, *strength*)). The minimum length of the *seed* (*seedlen*) **shall** be the maximum of the hash output block size (*outlen*) and the security *strength* + 64; the maximum length of the *seed* **shall** be the size of the hash input block (*inlen*); i.e., **max** (*outlen, strength* + 64) ≤ *seedlen* ≤ *inlen*. Further requirements for the *seed* are provided in Section 8.4.

Table 1: Security Strength, Entropy Requirement and Seed Length for Each Hash Function

Hash Function	Security Strength	Required Minimum Entropy	Seed Length
SHA-1	80	128	160-512
	112	128	160-512
	128	128	192-512
SHA-224	80	128	224-512
	112	128	224-512
	128	128	224-512
	192	192	256-512
SHA-256	80	128	256-512
	112	128	256-512
	128	128	256-512
	192	192	256-512
	256	256	384-512
SHA-384	80	128	384-1024
	112	128	384-1024
	128	128	384-1024
	192	192	384-1024
	256	256	384-1024
SHA-512	80	128	512-1024
	112	128	512-1024
	128	128	512-1024
	192	192	512-1024
	256	256	512-1024

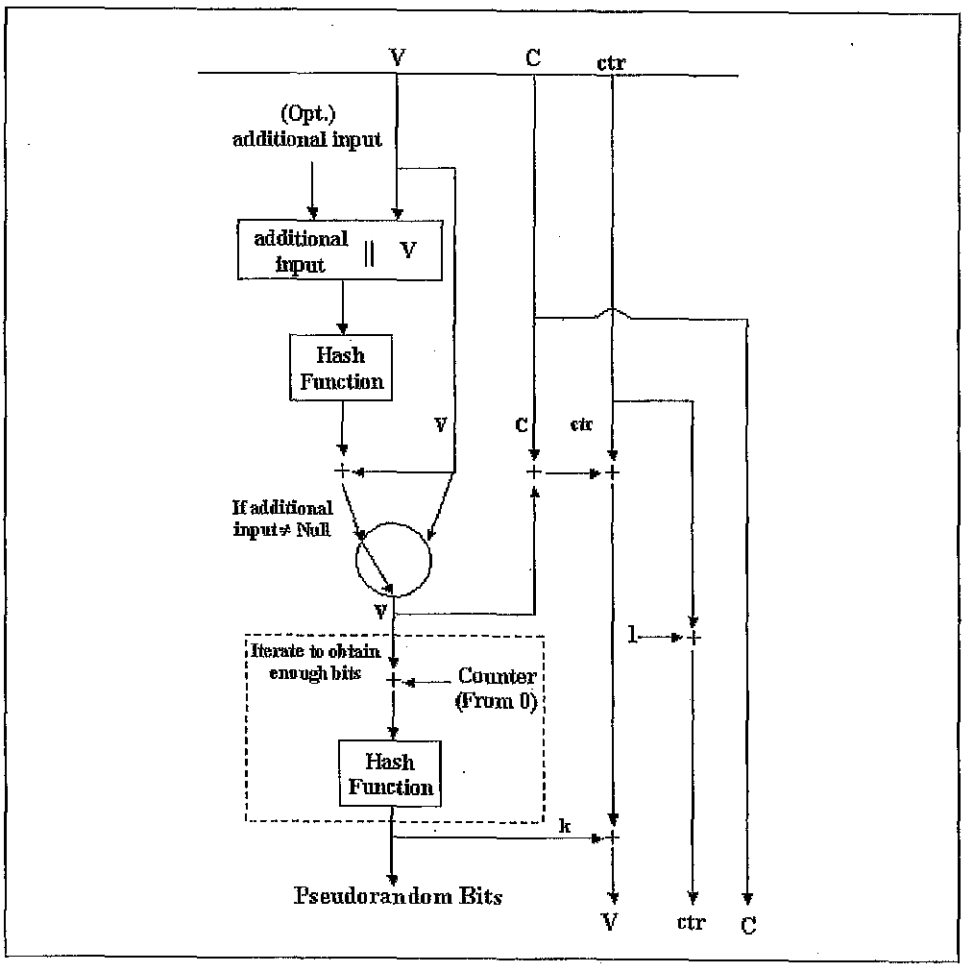

Figure 10: Hash_DRBG (...)

The application-specific constant (*t*) **shall** be *outlen* bits in length. See Annex E.22 for some values for *t* for different purposes.

Figures 11 and 12 depict the insertion of test input for the *seed*, the application-specific constant (*t*) and the additional input values (*additional_input*). The tests **shall** be run on the output of the generator.

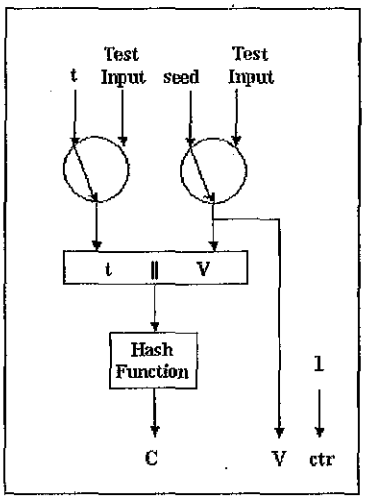

Figure 11: Hash_DRBG (...) Instantiation (with Tests)

Validation and operational testing are discussed in Section 11. Detected errors **shall** result in a transition to the error state.

10.1.2.2 Interaction with Hash_DRBG (...)

10.1.2.2.1 Instantiating Hash_DRBG (...)

Prior to the first request for pseudorandom bits, **Hash_DRBG (...) shall** be instantiated using the following call:

status = **Instantiate_Hash_DRBG** (*usage_class, requested_strength, prediction_resistance_flag, personalization_string*),

as described in Section 9.6.1.

10.1.2.2.2 Reseeding a Hash_DRBG (...) Instantiation

When a DRBG instantiation requires reseeding (see Section 9.7), the DRBG **shall** be reseeded using the following call:

status = **Reseed_ Hash_DRBG_Instantiation** (*usage_class*)

as described in Section 9.7.2.

10.1.2.2.3 Generating Pseudorandom Bits Using Hash_DRBG (...)

An application **shall** request the generation of pseudorandom bits by **Hash_DRBG (...)** using the following call:

(*status, pseudorandom_bits*) = **Hash_DRBG** (*usage_class, requested_no_of_bits, requested_strength, additional_input, prediction_resistance_flag*)

as described in Section 9.8.2.

10.1.2.2.4 Self Testing of the Hash_DRBG (...) Process

A Hash_DRBG(...) implementation is tested at power up and on demand using the following call:

(*status*) = **Self_Test_Hash_DRBG** ( )

as described in Section 9.9.

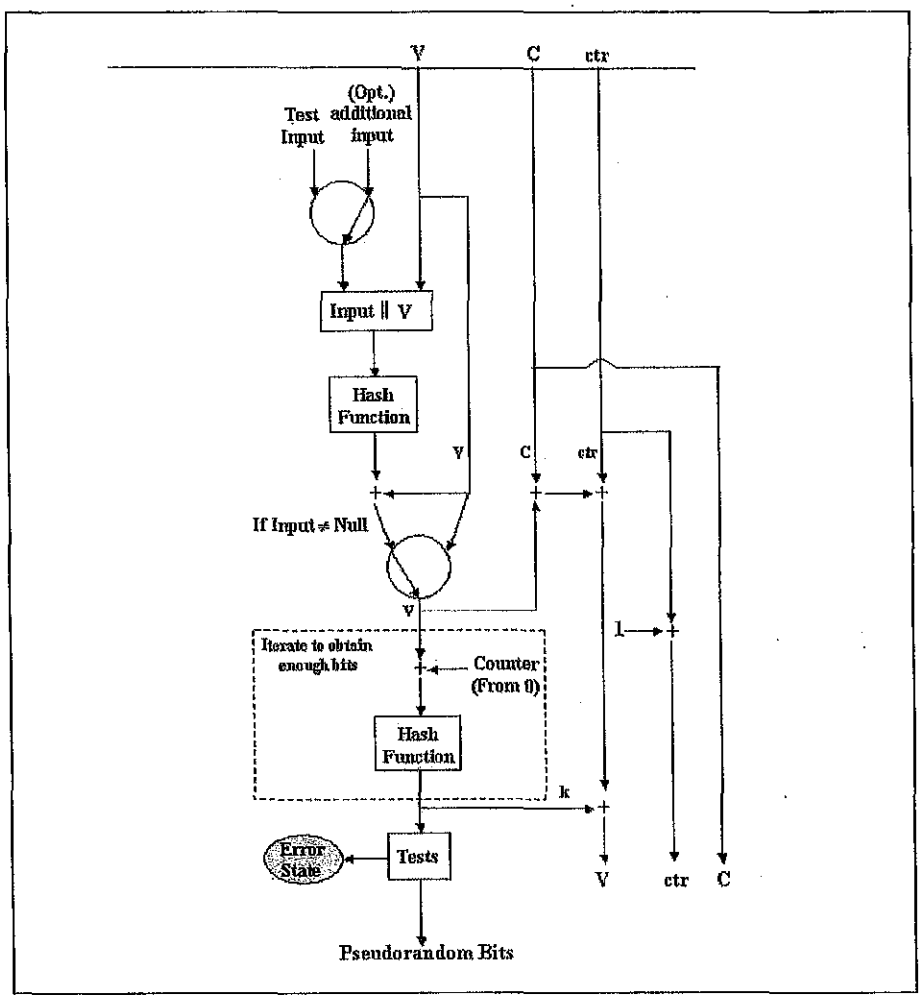

Figure 12: Hash_DRBG (...) with Tests

10.1.2.3 Specifications

10.1.2.3.1 General

The instantiation and reseeding of **Hash_DRBG (...)** consists of obtaining a *seed* with at least the requested amount of entropy. The *seed* is used to derive elements of the initial *state*, which consists of:

1. (Optional) The *usage_class* for the DRBG instantiation; if the DRBG is used for multiple *usage_classes*, requiring multiple instantiations, then the *usage_class*

parameter **shall** be present, and the implementation **shall** accommodate multiple *states* simultaneously; if the DRBG will be used for only one *usage_class*, then the *usage_class* parameter **may** be omitted).

2. A value (*V*) that is updated during each call to the DRBG.
3. A constant *C* that depends on the application-specific constant (*t*) and the *seed*.
4. A counter (*ctr*) that indicates the number of ipdates of *V* since the *seed* was acquired.
5. The application specific constant (*t*) (see Annex E).
6. The security *strength* of the DRBG instance.
7. The length of the *seed* (*seedlen*).
8. A *prediction_resistance_flag* that indicates whether or not prediction resistance is required by the DRBG, and
9. (Optional) A transformation of the *seed* using a one-way function for later comparison with a new *seed* when the DRBG is reseeded; this value **shall** be present if the DRBG will potentially be reseeded; it **may** be omitted if the DRBG will not be reseeded.

The variables used in the description of **Hash_DRBG (...)** are:

additional entropy	A string of bits containing entropy.
additional_input	Optional additional input.
C	An *outlen*-bit constant that is calculated during the instantiation and reseeding processes.
ctr	A counter that is used to update the *state* of **Hash_DRBG (...)** and records the number of times that *V* has been updated since the instantiation was seeded or reseeded.
data	The *data* to be hashed.
Get_additional_input ( )	Returns a value for *additional_input*. The specification of this function is left to the implementer. See Section 9.6.3.
Get_entropy (*min_entropy, min_length, max_length*)	A function that acquires a string of bits from an entropy source. *min_entropy* indicates the minimum amount of entropy to be provided in the returned bits; *min_length* indicates the minimum number of bits to be returned; *max_length* indicates the maximum number of bits to be returned. See Section 9.6.2
Hash (*a*)	A hashing operation on data *a* using an appropriate Approved hash function for the required security strength (see Table 1).
Hash_df (*seed_material, seedlen*)	The derivation function specified in Section 9.6.2. The same hash function that is used to generate bits for the **Hash_DRBG (..,) shall** be used by the derivation function. *seed_material* is the data that will be used to create the seed; *seedlen* is the requested seed length.
i	A temporary value used as a loop counter.
inlen	The length of the input block of a hash function.

m	The number of iterations of the hash function needed to obtain the requested number of pseudorandom bits.
max_updates	The maximum number of updates of *V* for the DRBG.
min_entropy	The minimum amount of *entropy* to be provided in the *seed*.
min_length	The minimum length of the *seed*.
old_seedlen	The *seedlen* from the previous seeding of the instantiation.
old_transformed_seed	The *transformed_seed* from the previous seeding of the instantiation
outlen	The length of the hash function output block.
prediction_resistance_flag	For instantiation, this flag indicates whether or not prediction_resistance may need to be provided upon request. 1 = requests may indicate a need for prediction resistance; 0 = prediction_resistance should never be provided.
prediction_resistance_requested	For pseudorandom bit generation, this flag indicates whether or not prediction resistance is required; 1 = yes, 0 = no.
pseudorandom_bits	The pseudorandom bits produced by the DRBG.
requested_no_of_bits	The number of bits to be generated.
requested_strength	The security strength to be associated with the pseudorandom bits obtained from the DRBG.
seed	The string of bits containing entropy that is used to determine the initial state of the DRBG during instantiation or reseeding.
seedlen	The length of the *seed* containing the required entropy.
seed_material	The data that will be used to create the *seed*
state	The *state* of **Hash_DRBG (...)** that is carried between calls to the DRBG. In the following specifications, the entire *state* is defined as {*usage_class, V, C, ctr, t, strength, seedlen, prediction-resistance_flag, transformed_seed*}. A particular element of the *state* is specified as *state.element*, e.g., *state.V*.
status	The status returned from a function call, where *status* = "Success", "No update performed" (informative message only) or an indication of failure. Failure messages are: 1. Invalid *requested_strength*. 2. No value of *t* is available for the *usage_class*. 3. Failure indication returned by the entropy source. 4. State not available for the indicated *usage_class*. 5. Failure from request for *additional_input*. 6. Prediction resistance cannot be supported. 7. Prediction resistance capability not instantiated
strength	The security strength provided by the instance of the DRBG.
t	The application-specific constant associated with the *usage_class* (see Annex E.3).
transformed_seed	A one-way transformation of the *seed* for the **Hash_DRBG (...)** instance.
usage_class	The usage class of a DRBG instance.

Comment [ebb6]: Page: 1
Need a value for this.

Comment [ebb7]: Page: 1
This may not make sense.

V	A value that is initially derived from the *seed*, but assumes new values based on optional additional input (*additional_input*), the pseudorandom bits produced by the generator (*pseudorandom_bits*), the constant (*C*) and the iteration count (*ctr*).
w, W	Intermediate values.

10.1.2.3.2 Instantiation of Hash_DRBG (...)

The following process or its equivalent **shall** be used to instantiate the **Hash_DRBG (...)** process. Let **Hash (...)** be the Approved hash function to be used; let *outlen* be the output length of that hash function, and let *inlen* be the input length.

Instantiate_Hash_DRBG (...):

Input : integer (*usage_class, requested_strength, prediction_resistance_flag, personalization_string*).

Output : string *status.*

Process :

1. If *requested_strength* > the maximum security *strength* that can be provided for the hash function (see Table 1), then **Return** ("Invalid *requested_strength*").
2. If (*prediction_resistance_flag* = 1) and prediction resistance cannot be supported, then **Return** ("Prediction resistance cannot be supported").
3. Set the strength to one of the five security strengths.
   If (*requested_strength* ≤ 80), then *strength* = 80
   Else if (*requested_strength* ≤ 112), then *strength* = 112
   Else (*requested_strength* ≤ 128), then *strength* = 128
   Else (*requested_strength* ≤ 192), then *strength* = 192
   Else *strength* = 256.
4. Set up *t* in accordance with the indicated *usage_class*. If no value of *t* is available for the *usage_class*, then **Return** ("No value of *t* is available for the *usage_class*").
5. *min_entropy* = **max** (128, *strength*).
6. *min_length* = **max** (*outlen*, *strength*).

   Comment Get the *seed.*

7. (*status, entropy_bits*) = **Get_entropy** (*min_entropy, min_length, inlen*).
8. If (*status* = "Failure"), then **Return** ("Failure indication returned by the entropy source").
9. *seed_material* = *entropy_bits* || *personalization_string.*
10. *seedlen* = **max** (*strength* + 64, *outlen*).
11. If (*seedlen* > *inlen*), then *seedlen* = *inlen.*

    Comment: Ensure that the entropy is distributed throughout the seed.

12. *seed* = **Hash_df** (*seed_material, seedlen*).

    Comment : Perform a one-way function on the seed formlater comparison during reseeding.

13. *transformed_seed* = **Hash** (*entropy_bits*).

14. *ctr* = 1.

15. *V* = *seed*.
16. *C* = **Hash** (*t* || *V*).
17. *state* = {*usage_class*, *V*, *C*, *ctr*, *t*, *strength*, *seedlen*, *prediction_resistance_flag*, *transformed_seed*}.
18. **Return** ("Success").

Note that multiple *state* storage is required if the DRBG is used for multiple *usage_classes*.
If an implementation does not need the *usage_class* as a calling parameter (i.e., the implementation does not handle multiple usage classes), then the *usage_class* parameter can be omitted, step 4 must set *t* to the value to be used, and the *usage_class* indication in the *state* (see step 17) must be omitted.
If an implementation does not handle all five security strengths, then step 3 must be modified accordingly.
If no *personalization_string* will ever be provided, then the *personalization_string* parameter in the input may be omitted, and step 9 becomes *seed_material* = *entropy*.
If an implementation will never be reseeded using the process specified in Section 10.1.2.3.3, then step 13 may be omitted, as well as the *transformed_seed* in the *state* (see step 17).
If an implementation does not need the *prediction_resistance_flag* as a calling parameter (i.e., the **Hash_DRBG (....)** routine in Section 10.1.2.3.4 either always or never acquires new entropy in step 5), then the *prediction_resistance_flag* in the calling parameters and in the *state* (see step 17) may be omitted.

10.1.2.3.3 Reseeding a Hash_DRBG (...) Instantiation

The following process or its equivalent **shall** be used to reseed the **Hash_DRBG (...)** process. Let **Hash (...)** be the Approved hash function to be used; let *outlen* be the output length of that hash function, and let *inlen* be the input length.

Reseed_Hash_DRBG_Instantiation (...):

Input: integer (*usage_class*).
Output: string *status*.
Process:

1. If a *state* is not available for the indicated *usage_class*, then **Return** ("State not available for the indicated *usage_class*").
2. Get the appropriate *state* values for the indicated *usage_class*, e.g., *V* = *state.V*, *t* = *state.t*, *strength* = *state.strength*, *old_seedlen* = *state.seedlen*, *old_transformed_seed* = *state.transformed_seed*.
3. *min_entropy* = **max** (128, *strength*).
4. *min_length* = **max** (*outlen*, *strength*).
5. (*status*, *entropy_bits*) = **Get_entropy** (*min_entropy*, *min_length*, *inlen*).
6. If (*status* = "Failure"), then **Return** ("Failure indication returned by entropy source").

Comment: Determine the larger of the key sizes so that entropy is not lost.

7. *seedlen* = **max** (*strength* + 64, *outlen*).
   Comment: Combine the new *entropy_bits* with the entropy present in *V*, and distribute throughout the *seed*.
8. *seed_material* = *entropy_bits* || *V*.
9. *seed* = **Hash_df** (*seed_material*, *seedlen*).
   Comment: Perform a one-way function on the seed and compare with the old transformed seed.
10. *transformed_seed* = **Hash** (*entropy_bits*).
11. If (*transformed_seed* = *old_transformed_seed*), then **Return** ("Entropy source failure").
12. V = seed.
13. *ctr* = 1.
14. *C* = **Hash** (*t* || *V*).
15. Update the appropriate *state* values for the *usage_class*.
    15.1 *state.V* = *V*.
    15.2 *state.C* = *C*.
    153 *state.ctr* = *ctr*.
    15.4 *state.seedlen* = *seedlen*.
    15.5 *state.transformed_seed* = *transformed.seed*.
16. **Return** ("Success").

If an implementation does not need the *usage_class* as a calling parameter (i.e., the implementation does not handle multiple usage classes), then the *usage_class* parameter and step 1 can be omitted, and steps 2 and 15 will use the only *state* available.

10.1.2.3.4 Generating Pseudorandom Bits Using Hash_DRBG (...)

The following process or its equivalent **shall** be used to generate pseudorandom bits. Let **Hash (...)** be the Approved hash function to be used; let *outlen* be the output length of that hash function, and let *inlen* be the input length.

Hash_DRBG (...):

Input: integer (*usage_class*, *requested_no_of bits*, *requested_strength*, *additional_input*, *prediction_resistance_requested*).

Output: string *status*, bitstring *pseudorandom_bits*.

Process:

1. If a *state* for the indicated *usage_class* is not available, then **Return** ("State not available for the indicated *usage_class*", Null).
2. Set up the *state* in accordance with the indicated *usage_class*, e.g., *V* = *state.V*, *C* = *state.C*, *ctr* = *state.ctr*, *strength* = *state.strength*, *seedlen* = *state.seedlen*, *prediction_resistance_flag* = *state.prediction_resistance_flag*.
3. If (*requested_strength* > *strength*), then **Return** ("Invalid *requested_strength*").
4. If ((*prediction_resistance_requested* = 1) and (*prediction_resistance_flag* = 0)), then Return ("Prediction resistance capability not instantiated").
5. If (*prediction_resistance_requested* = 1), then
   5.1 *status* = **Reseed_ Hash_DRBG_Instantiation** (*usage_class*).

5.2 If (*status* ≠ "Success"), then **Return** (*status*, Null).

6. If (*additional_input* ≠ Null), then do
   6.1 w = **Hash** (*additional_input* || V).
   6.2 $V = (V + w) \bmod 2^{seedlen}$.
7. *pseudorandom_bits* = **Hashgen** (*requested_no_of_bits*, V).
8. $V = (V + pseudorandom\_bits + C + ctr) \bmod 2^{seedlen}$.
9. $ctr = ctr + 1$.
10. If ($ctr \geq max\_updates$), then
    10.1 *status* = **Reseed_ Hash_DRBG_Instantiation** (*usage_class*).
    10.2 If (*status* ≠ "Success"), then **Return** (*status*, Null).
    Else Update the changed values in the *state*.
    10.3 *state*.V = V.
    10.4 *state.ctr* = *ctr*.
11. **Return** ("Success", *pseudorandom_bits*).

Hashgen (...):

Input: integer *requested_no_of_bits*, bitstring V.
Output: bitstring *pseudorandom_bits*.
Process:

1. $m = \left\lceil \frac{requested\_no\_of\_bits}{outlen} \right\rceil$.
2. $data = V$.
3. W = the Null string.
4. For i = 1 to m
   4.1 w_i = **Hash** (*data*).
   4.2 $W = W \| w_i$.
   4.3 $data = data + 1$.

   [Note that in Figures 5 and 7, this step is shown a bit differently; a suggestion for reconciliation is welcome.]
5. *pseudorandom_bits* = Leftmost (*requested_no_of_bits*) bits of W.
6. **Return** (*pseudorandom_bits*).

If an implementation does not need the *usage_class* as a calling parameter (i.e., the implementation does not handle multiple usage classes), then the *usage_class* input parameter and step 1 can be omitted, and step 2 uses the only *state* available.
If an implementation does not need the *prediction_resistance_flag*, then the *prediction_resistance_flag* and steps 4 may be omitted. If prediction resistance is never used, then step 5 may be omitted.