IETF
Internet-Draft
Intended status: Standards Track
Expires: December 31, 2018

A. Freytag
ASMUS, Inc.
J. Klensin

A. Sullivan
Oracle Corp.
June 29, 2018

Those Troublesome Characters: A Registry of Unicode Code Points Needing Special Consideration When Used in Network Identifiers
draft-freytag-troublesome-characters-02

Abstract

Unicode's design goal is to be the universal character set for all applications. The goal entails the inclusion of very large numbers of characters. It is also focused on written language in general; special provisions have always been needed for identifiers. The sheer size of the repertoire increases the possibility of accidental or intentional use of characters that can cause confusion among users, particularly where linguistic context is ambiguous, unavailable, or impossible to determine. A registry of code points that can be sometimes especially problematic may be useful to guide system administrators in setting parameters for allowable code points or combinations in an identifier system, and to aid applications in creating security aids for users.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on December 31, 2018.

Table of Contents

1. Unicode code points and identifiers

Unicode [Unicode] is a coded character set that aims to support every writing system. Writing systems evolve over time and are sometimes influenced by one another. As a result, Unicode encodes many characters that, to a reader, appear to be the same thing; but that are encoded differently from one another. This sort of difference is usually not important in written texts, because competent readers and writers of a language are able to compensate for the selection of the "wrong" character when reading or writing. Finally, the goal of supporting every writing system also implies that Unicode is designed to properly represent written language; special provisions are needed for identifiers.

Identifiers that are used in a network or, especially, an Internet context present several special problems because of the above feature of Unicode:

[[CREF1: AF: This whole business of language context seems unconnected from the data we have in the registry: that data is about code points and sequences that look the same, and many examples are in the same language. For example the duplicated shapes for digit / letter pairs. In very few cases would knowing the language context make a difference. In some cases, if you knew the script (not for the label, but the code point) you might be able to distinguish two labels, but that is it. I think we should further rewrite this summary so it matches better with the what the proposed registry contains.]]

1. In many (perhaps most) uses of identifiers, they are neither constrained to words in a particular language, nor would it be possible to ascertain reliably the language context in which the identifier is being or will be used. In the case of an internationalized domain name, for instance, each label could in principle represent a new locus of control, because there could be a delegation there. A new locus of control means that the administrator of the resulting zone could speak, read, or intend a different language context than the one from the parent. Moreover, at least some domains (such as the root) have an Internet-wide context and therefore do not really have a language context as such. In any case, the language context is simply not available as part of a DNS lookup, so there is no way to make the DNS sensitive to this sort of issue. Even in the case of email local-parts, where a sender is likely to know at least one of the languages of the receiver, the language context that was in use at the time the identifier was created is often unknown.

2. Identifiers on the network are in general exact-match systems, because an ambiguous identifier is problematic. Sometimes, but not always, there are facilities for aliasing such that multiple identifiers can be put together as a single identity; the DNS, for example, does not have such an aliasing capability, because in the DNS all aliases are one-way pointers. Aliasing techniques are in any case just an extension of the exact-match approach, and do not work the way a competent human reader does when interpolating the "right" character upon seeing the "wrong" one.

3. Because there are many characters that may appear to be the same (or even, that are defined in such a way that they are all but guaranteed to be rendered by the same glyphs), it is fairly easy to create an identifier either by accident or on purpose that is likely to be confused with some other identifier even by competent readers and writers of a language. In some cases knowing the language context would be of no help to recognition, for example, in cases where a language uses the same shape for a letter as for one of the digits.

4. For some scripts their repertoire of shapes overlaps with one or more other scripts, so that there are cases where two strings look identical to each other, even though all the code points in the first string are of one script, and all the code points in the second string are of another script. In these cases, the strings cannot be distinguished by a reader, and the whole strings are confusable.

5. For some scripts, both users and rendering systems do not expect to encounter code points in arbitrary sequence. Most code points normally occur only in specific locations within a syllable. If random labels were permitted, some would not display as expected (including having some features misplaced or not displayed) while others would present recognition problems to users experienced with the script. Some devices may also not support arbitrary input.

Beyond these issues, human perception is easily tricked, so that entirely unrelated character sequences can become confusable -- for example "rn" being confused with "m". Humans read strings, not characters, and they will mostly see what they expect to see. Some additional discussion of the background can be found in Appendix A.

The remainder of this document discusses techniques that can be used to design the label generation rules for a particular zone so they ameliorate or avoid entirely some of the issues caused by the interaction between the Unicode Standard and identifiers. The

registry is intended to highlight code points that require such techniques.

2. Background and Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

A reader needs to be familiar with Unicode [Unicode], IDNA2008 [RFC5890] [RFC5891] [RFC5892] [RFC5893] [RFC5894], PRECIS (at least the framework, [RFC7564]), and conventions for discussion of internationalization in the IETF (see [RFC6365]).

3. Techniques already in place

In the IDNA mechanism for including Unicode code points [RFC5892], a code point is only included when it meets the needs of internationalizing domain names as explained in the IDNA framework [RFC5894]. For identifiers other than those specified by IDNA, the PRECIS framework [RFC7564] generalizes the same basic technique. In both cases, the overall approach is to assume that all characters are excluded, and then to include characters according to properties derived from the Unicode character properties. This general strategy cuts the enormous size of the Unicode database somewhat, avoiding including some characters that are necessarily unsuited for use as identifiers.

The mechanism of inclusion by derived property, while helpful, is insufficient to guarantee every included character is safe for use in identifiers. Some characters' properties lead them to be included even though they are not obviously good candidates. In other cases, individual characters are good for inclusion, but are problematic in combination. Finally, there are cases where characters (or sequences of characters) are not problematic by themselves, or if used in a mutually exclusive manner in the same identifier, but become problematic when their choice represents the only difference between otherwise identical identifiers. For some examples, see Appendix B.

Operators of systems that create identifiers (whether through a registry or through a peer-to-peer identifier negotiation system) need to make policies for characters they will permit. Operators of registries, for instance, can help by adopting good registration policies: "Users will benefit if registries only permit characters from scripts that are well-understood by the registry or its advisers."[RFC5894]

The difficulty for many operators, however, is that they do not have the writing system expertise to claim any character is "well-understood", and they do not really have the time to develop that expertise. Such operators should in fact not use or register such characters. Unfortunately, in many cases the operators are stewards of systems where the user population demands identifiers useful to them in their local languages. In other cases, operators may proceed without a proper understanding owing to financial or market share incentives. The risk for Internet identifiers in such cases is obviously that ill-understood and potentially exploitable gaps in registration policies will open.

To help mitigate such issues, this document proposes a registry of Unicode code points that are known to present special issues for network identifiers with the aim to guide protocol and operating decisions about whether to permit a given code point or sequence of code points. By necessity, any list or guidance can only reflect issues that are known and understood at the time of writing. By limiting itself largely to characters that are widely used to write languages in contemporary use, the registry will address the more critical needs, while simultanesously focusing on characters that are well understood and for which there may already be some implementation experience in IDNs.

By itself, such a registry will not completely protect against poor registration or use, but it may provide operational guidance necessary for people who are responsible for creating policies. It also obviates the need for everyone to repeat basic investigation into the behavior of Unicode characters. Instead, scarce expertise can be focused on ways to mitigate issues, perhaps caused by user requirements for a specific character.

Note that the registry defined herein does not address any of the issues created by whole-string confusables where each of the identifiers is of a different script. A common workaround, limiting a registry to identifiers of only a single script, would mitigate this issue. [[CREF2: AF: we should evaluate that; cross-script variants that are homoglyphs have now been collected across modern scripts as part of the root zone LGR and are easily captured in a registry.]]

For some of the code points (or code point sequences) listed as presenting issues for identifiers, it may be most expeditious to simply not include them, even though they are valid according to the protocol. Sometimes, one of a pair of identical code points (or code point sequences) may be deemed preferable over the other for practical reasons.

However, simply leaving out any code point listed in this registry would render a registry of doubtful value for many scripts. It is not always necessary or desirable to exclude characters. Sometimes, it is merely necessary to ensure that for two otherwise identical identifiers, only one of a set of mutually exclusive code points (or sequences of code points) is used, while preventing the later registration of the label containing the other one in order to avoid ambiguity. This way the operator does not need to impose a choice.

In cases where two or more variants of such an identifier mean the same thing to the native reader, an operator may decide to allow all of the variant labels to be registered simultaneously, but only to the same entity (and with proper safeguards that limit the multiplicity of such allocatable variant labels).

The implementation of this strategy would be via the variant mechanism described in [RFC7940] and [RFC8228] which allows mechanical processing of mutual exclusion and /or bundling of identifiers respectively.

This specification defines a registry of code points and sequences that have been identified as requiring special attention when they are to be used in identifiers. An administrator who does not have the time or inclination to develop the requisite policies might contemplate simply not to permit these code points at all.

However, for some scripts the remaining subset might not be usable in a meaningful way. Identifiers in these scripts cannot be safely implemented without understanding the issues involved. Further note that many code points listed here are problematic only in their relationship to other code points and that as long as these issues are adequately addressed, for example using the variant mechanism, they do not need to be excluded. [[CREF3: AF: the above needs more editing, it’s a bit repetitive.]]

4. A registry of code points requiring special attention

4.1. Description

The registry contains four fields. [[CREF4: AF If we are limited to the "texttable" format, we are limited to three columns, there’s no way we can fit more than that into the RFC plain text format and remain legible. If we want more columns, then we need to use some other data format, including PDF (which would allow us to show the images for the code points).]]

1. The first field, called "Code Point(s)", is a code point or sequence of code points. Sequences in this and other fields are

listed as space separated code point values. For completeness, full code point sequences are listed, even if some of their constituents are "Not recommended". A code point value is a series of 4-6 uppercase hexadecimal digits, as defined in [Unicode].

2. The second field, "Related CP", contains zero or more cross references to related code points or sequences. Cross references consist of single code points or sequences. Multiple cross references are separated by a comma.

3. The third field, called "References", contains one or more references to documents describing the code point and the reason why it presents an issue. References are cited by numeric values, each in square brackets; multiple references are separated by space.

4. The last field, "Comment", is a free form text field that briefly describes the issue; it also The comment field starts with a category, separated by a colon, to allow quick identification of similar cases

The following are the defined category values:

Not Recommended While the code point (or sequence) is not DISALLOWED, there is emerging consensus in the community that it is not recommended for identifiers, or it is considered as such in the Unicode Standard. This includes, but is not limited to code points that are formally deprecated in the Unicode standard, as well as code points or sequences listed in the standard as "Do not use" or not preferred or similar. Code points not in active use, obsolete code points, or those intended for specialist use may also be listed under this category. Details are given in the explanation and references.

Identical The code point (or sequence) is normally identical in appearance to another code point (or sequence); or may be identical in some contexts. If the related CP is listed as "PREFERRED", it is recommended that this code point (or sequence) be excluded; in the case of a sequence, it may be appropriate to exclude, the constituent combining marks (after first consulting the details given in the listing for the marks). Otherwise, it is recommended to make the two identical code points or sequences mutually exclusive by treating them as variants. Details are given in the explanation and references.

Restricted Context The code point is problematic in relation to some other code points in the same label. For example, it should be

used only after some code points or not adjacent to certain other code points. Further details are given in the explanation and references. This is a common case for certain combining marks or other code points in so-called "complex" scripts. These scripts generally require a coordinated set of context rules; in those cases the registry would not list any specific context rules, but to point to documentation of existing Label Generation Rulesets implementing a coherent set of rules as examples. Code points with IDNA2008 property of CONTEXTJ or CONTEXTO are not listed, as long as the given context rules mitigate any concerns.

Preferred The code point is preferred to some other code point given in the cross reference (with the other code point normally "IDENTICAL" or "NOT RECOMMENDED"). In some cases this represents a preference for a code point (or sequence) that is a basic constituent in some alphabet over a code point (or sequence) that is rare or has specialized use. In some cases the preference may be formally specified or otherwise represent established community consensus. Details are given in the explanation and references.

Other All cases that do not fit one of the other categories. Details are given in the explanation and references.

If a character appears in the registry, that does not automatically mean that it is a bad candidate for use in identifiers generally. Absent a well-defined and verifiable policy, however, such a code point or sequence might well be treated with suspicion by users and by tools.

For code points tagged as being "identical" to or "indistinguishable" from other code points, it may be that one is preferred over the other, but it may also be that implementing a scheme for mutual exclusion of any resulting identical labels is the best solution, such as assigning them "blocked" variants according to [RFC7940] and [RFC8228].

Where characters are confusable with a combining sequence, only the combining sequence is listed; suggested mitigation may consist of disallowing either the specific combining sequence or disallowing the combining marks involved. It is usually inappropriate to exclude any of the basic letters involved, as they are generally members of the standard alphabet for one or more languages.

The registry and this document are to be understood as guidance for the purpose of developing operational policies that are used for protocols under normal administrative scope. For instance, zone operators that support IDNA are expected to create policies governing the code points that they will permit (see [RFC5894] and

[I-D.rfc5891bis]). The registry herein defined is intended to highlight particularly troublesome code points or code point sequences for the benefit of administrators creating such policies. It is also intended to highlight characters that may create identifier ambiguities and thereby create security vulnerabilities. However, by itself it is no substitute for such policies.

The registry is by necessity limited to code points for which adequate information is available; by and large this means code points used in connection with modern languages or writing systems, except that specialized extensions to modern scripts may be indicated, if their use would fall into any of the categories defined. Historic scripts, and any modern scripts not represented in the registry can be assumed to not be well-understood; operators are cautioned to locate other sources of information and to develop the necessary policies before deploying such scripts.

4.2. Maintenance

The registry is updated by Expert Review using an open process. From time to time, additional code points may be added to the Unicode standard, or further information may be discovered related to code points, to existing code points or those already listed here. The Unicode Standard may recommend against using a code point for all or some purposes. Or a script community may have gained more experience in deploying IDNs for that script and may create or update recommendations as to best policy.

4.3. Scope

Code points that are DISALLOWED in IDNA 2008 are not eligible to be listed. Code points that are CONTEXTJ or CONTEXTO are not included here unless there are documented concerns that are not mitigated by the existing IDNA context rules. The focus is on scripts that are significant for identifiers; code points from scripts that are historic or otherwise of limited use have generally not been considered - however exceptions may exist where authoritative information is readily available. Code points and code point sequences included are those that need special policies (including, but not limited to policies of exclusion).

New code points of sequences are listed whenever information becomes available that identifies a specific issue that requires attention in crafting a policy for the use of that code point or sequence in network identifiers. Likewise cross references, categories, explanations and references cited may be updated.

The contents of the registry generally does not represent original research but a collection of issues documented elsewhere, with appropriate references cited. An exception might be cases that are in clear analogy to existing entries, but not explicitly covered by existing references, for example, because the code point in question was recently added to Unicode.

If a particular language or script community reaches an apparent consensus that some code point is problematic, or that of two identical code points or sequences one should be preferred over the other, such recommendations, if known, should be documented in this registry.

In addition, if the Unicode Standard designates a code point as formally "deprecated" or less formally as "do not use", or identifies code points that are "intentionally identical", this is also something that should be reflected in the registry. Another source of potential information might be existing registry policies or recommended policies, particularly where it is apparent that they represent a careful analysis of the issue or a wider consensus, or both.

Proposed additions to the registry are to be shared on a mailing list to allow for broader comment and vetting.

If there is a disagreement about the existence of an issue or its severity, it is preferable to document both the issue and the different evaluations of it. In all cases, the information and documentation presented must allow a user to fully evaluate the status of any entry in the registry.

There is no requirement for the registry to form a stable body of data to which any future document would have to be backward compatible in any way. If new information emerges, additional code points may be considered problematic, or they may need to be reclassified. In case of significant changes, the explanation should note the nature of the change and cite a reference to document the basis for it.

5. Registry initial contents

5.1. Overview

IDNA 2008 uses an inclusion process based on Unicode properties to define which code points are PVALID, but also recognizes that some code points require a context rule (CONTEXTJ, CONTEXTO).

A number of code points which are PVALID in [RFC5892] may require additional attention in the design of label generations rules. In some cases, the issue is not necessarily with an individual code point, but with a code point sequence. In the following, "code point" and "code point sequence" are used synonymously unless explicitly called out. The fact that a code point require such attention does not affect its status under IDNA 2008.

The following describes a number of conditions that pose problems for network identifiers and common strategies for mitigating them.

5.2. Interchangeable Code Points

At times two code points or code point sequences are considered by all users (or a significant fraction) as equivalent to a degree that they accept one of them as substitute for another. This has obvious implications for the unambiguous recognition of identifiers. This document lists the code points and sequences affected (except for certain generic classes too numerous to list here). Note that one of the two may be preferred over the other, in which case the non-preferred one may be excluded or folded away. But in many cases either one is equally preferred. Mitigation techniques for such cases are discussed below.

Homoglyphs Homoglyphs are code points that have identical appearance, or are so close in appearance that they are indistinguishable if not presented side-by-side. Whenever two labels differ only by code points that are homoglyphs of each other and occur in the same position, users cannot distinguish the labels from each other or tell which label is intended, even though the underlying code points are different. Users will substitute one label for another.

Code points that are merely similar in appearance, including strongly similar code points, or code points that are difficult to distinguish (such as certain diacritical marks) are not considered here; handling such similarities often requires case by case judgment.

Instead, this document considers these types of code points that can be fully substituted for one another:

1. code points that, by design or derivation, are identical to each other;

2. code points that assume the same shape in some context, e.g. at the end of a label;

3. code points of a striking similarity based on derivation or common origin;

4. and code points that are otherwise indistinguishable from one another unless placed side by side.

Cross-script Homoglyphs A number of code points are homoglyphs of code points in another script (cross-script homoglyphs). Cross-script homoglyphs are a concern for any zone that supports labels from more than one script, even if each label is required to be in a single script. Note that some writing systems ordinarily use a combination of scripts (such as the use of Han, Hiragana and Katakana for Japanese). For many writing systems, an admixture of Latin letters is not uncommon, for example in brand or product names. If not handled carefully, this can prove problematic for identifiers.

Homophones As discussed in [202], the Amharic language treats many code points from the Ethiopic script as sound-alikes (homophones). In writing, these are freely substituted, users do not recognize some spelling as more correct. A conservative approach would treat these as mutually exclusive; the alternative, to make all variants available to the same applicant is appears not feasible due to the high number of such variants per label.

Semantic Variants The Chinese writing system, shared among several geographically distributed user communities, has many instances of code points that represent the same semantic. Even though they are visually distinct, they can be substituted for one another; typically these correspond to the simplified and traditional forms of Chinese characters. See [RFC4713] for details.

5.3. Excludable Code Points

Code points that are not substitutable but troublesome for other reasons are candidates for exclusion from a zone’s repertoire. For each such code point, the comment field briefly describes why it should be excluded or considered troublesome. There is no identified mitigation strategy that can be recommended for general usage: unless careful study indicates that a code point with this status is exceptionally acceptable for a particular zone, after all, it should normally be excluded from the repertoire. These reasons are varied.

Deprecated Code Points Deprecated code points are those that [Unicode] recommends not to use for any purpose. They should be excluded from identifiers; there is no mitigation. In addition, Unicode recommends against the use of some sequences and code

points for any purpose, but without formal deprecation. These should likewise be excluded from identifiers.

Non-preferred or other Troublesome Code Point This category includes all code points that are troublesome for other reasons; they include code points that represent non-preferred variations; or code points that not meant to be used in a combining sequence for letter; or code points that may be indistinguishable from a punctuation mark or other DISALLOWED code point. For each such code point, the comment field briefly describes why it should be excluded or considered troublesome.

Obsolete or not in Active Use Many code points across scripts that are otherwise in modern use represent additions for use in obsolete orthographies and writing systems, that is for writing languages that are extinct or not longer written in that script. Some have been researched and no evidence of active use could be found. These code points are not recommended for use in identifiers and should be excluded. Except for specialists, users are unlikely to recognize them, or find them of use in constructing mnemonic strings for identifiers. In addition, they often have not been sufficiently analyzed as to whether they represent other issues for identifiers. That makes their use risky. Obsolete, rare and code points otherwise not in active are generally not listed here. The reader can find a list of code points with high probability of being in active use in [MSR].

5.4. Combining Marks

Non Normalizable Sequences Certain combining marks are part of non-normalizable sequences. Normally, when a combining sequence is an alternate encoding to a composite code point, normalization can be used to select a preferred representation. For IDNA 2008, which uses NFC to normalize, this means the composite code point. However, some combining marks are not considered identical to the same mark when graphically part of a composite character. Sequences with these marks may look more or less like some composite code point, but they are considered different, and therefore not normalized. For identifiers, the best recommendation is to exclude those combining marks.

Combining marks that are also part of precomposed letters Many combining marks are part of canonical decompositions. For identifiers that are normalized to the composed forms using NFC (as required by IDNA 2008), these combining marks usually are not needed on their own, that is as separate element of a combining sequence after normalization. (The vast majority of letters using these marks have been encoded as precomposed characters). It is

strongly recommended to exclude these combining marks on their own, but, as needed for a specific language, to enumerate the needed sequences. (One notable example is Vietnamese which, after normalization to NFC uses a mixture of precomposed code points and combining marks). [TBD]The most common generic combining marks affected have been entered in the registry as excluded.

Non-spacing combining marks These marks are typically accents, diacritics and the like. They pose an additional problem: if they are allowed to occur twice in a row, some rendering systems will "overprint" them, in effect making them indistinguishable from single marks. This problem can be avoided by allowing only enumerated sequences, or alternatively by a context rule.

Ambiguous Rendering There are other ways in which certain code points and sequences representing particular combinations of code points may suffer from unreliable rendering, because rendering engines normally do not expect to encounter them. While Unicode allows the use of combining marks, in principle, in combination with any base character, in practice this can lead to unrecognizable labels, or labels that are not reliably distinct. This situation mostly affects the so-called complex scripts.

Combining marks in complex scripts In some scripts, there are no precomposed sequences. Usually, these scripts are "complex" scripts, that require context rules for many classes of code points. For these scripts, context rules (see [RFC7940]) should be used to limit non-spacing marks to acceptable contexts. For an example of such rules see [204], [206].

Soft Dotted and Dotless Letters Unicode code points with the Soft_Dotted property encode letter that lose their dot if followed by a diacritical mark above. (See [UCD]) If the following mark is a COMBINING DOT ABOVE, the combination is indistinguishable from the letter by itself. This can be mitigated by limiting or excluding the code point for DOT ABOVE. A soft dotted code point followed by any other diacritical mark above will look identical to the corresponding dotless letter with diacritical mark above. All combinations of dotless letters followed by diacritical marks should be excluded. (This can be done with a context rule, see [RFC7940]).

5.5. Mitigation

Thiere are several techniques that can be used to help to mitigate confusion. The focus in the following is on issues addressable by protocol or registry policy. However, user agents might implement

additional mitigation approaches, such as always using a font designed to distinguish among different characters.

5.5.1. Mitigation Strategies

Exclusion The primary mitigation technique is to reduce the problem space: operators should only ever use the smallest repertoire of code points possible for their environment. So, for example, if there is a code point that is sometimes used but is perhaps a little obscure, it is better to leave it out. Users are unlikely to be familiar with many code points added to Unicode for the representation of historical forms of writing a script, or for highly specialized purposes. That unfamiliarity may present challenges to correct identification or keyboard entry, making the code point less usable. In addition, their use may present other problems not appreciated by anyone not familiar with them.

For these reasons, code points used only in a language with which the administrator is not familiar should probably be excluded. The same applies to code points used in specialized contexts, such as those only found in historic or sacred documents, or only used for phonetic transcription or poetry.

By reducing the repertoire to a well-understood essential subset it is often possible to eliminate some possible instances of confusion. For example, in the Arabic script, combining marks are generally used for optional or specialized aspects of the writing system. At the same time, many combining sequences are confusable with basic letters of the script. Because of this, excluding all Arabic combining mark would greatly reduce confusability without significantly affecting usability of the script for identifiers.

Preferred code points Sometimes, each of these code points will be used by a different user community; or one of the code points is not in wide use, for example because it is intended for special purposes like phonetic annotation or transliteration. In such cases, the one not needed for a given zone could be excluded.

In other cases, zones may be shared by a wider community, making it unattractive or impossible to institute a preference. A common method of mitigating issues from such homoglyphs is to make two labels that differ only by using a different homoglyph mutually exclusive. This can be done by making the homoglyphs code point variants, usually of type "blocked". See [RFC8228].

In some cases, while two code points may be homoglyphs, one of them can be identified as the preferred alternative to encode the intended character. In these cases, one of the code points has

been identified as "preferred", while the other has been identified as "troublesome"; or "excluded". In all other cases, no such preference exists in the general usage; a conservative mitigation might be to define the alternatives as blocked variants. However, the users of a given zone might have a specific preference, in which case one of the alternatives could be excluded instead.

For convenience in presentation, this document presents pairs or sets of homoglyphs as mutually exclusive variants of type "homoglyph". Other ways of handling these code points are possible. While one might implement such a variant relation in many cases as one label blocking another, in some cases allowing both to be registered to the same applicant may be appropriate. Finally, in some case eliminating one or both code points from the repertoire may be a feasible alternative to establishing a variant relation.

Script limitation For homoglyphs, a large number of cases (but not all of them) turn out to be in different scripts. As a result, it is usually a good idea to adopt the operational convention that identifiers for a protocol should always be in a single script.

This mitigation strategy has limits. First, even if any given identifier is only in a single script, it may co-exist with identifiers from other scripts. Sometimes the repertoire used in operation allows multiple scripts that create whole string confusables -- strings made up entirely of homoglyphs of another string in a different script (such as can be found between Cyrillic and Latin, for example). In such cases, mitigation must turn to other means of preventing the registration of mutually confusable string, for example by In that case, a robust mechanism for mutual exclusion of confusable identifiers must exist, ensuring that the registration of one of them (whichever comes first) blocks the later registration of the other.

Second, some writing systems use a combination of scripts and for commercial names in many scripts, admixture of Latin letters is common. Allowing limited script mixing may be an essential requirement in some cases.

Lastly, identifiers are not always under the operational control of a single authority (such as in the case of DNS, where the system is under distributed control so that different parts of the hierarchy can have different operational rules).

In the case of IDNA, some client programs restrict display of U-labels to top-level domains known to have policies about single-script labels.

Exact homoglyphs No policy or convention, other than ensuring mutual exclusion, will do anything to help mititgate confusion for strict homoglyphs of each other in the same script (see Appendix B for some example cases.)

Beyond the issue of mutual confusability, some combining sequences in particular can give rise to other difficulties in recognition - usually because client systems will not reliably and correctly display them. One particular case concerns sequences of more than one instance of the same non-spacing combining mark such as the repetition of an accent or diacritic. These are often rendered indistinguishably from single instances of the same mark. Operators should prohibit such repetition, particularly, as there are no known cases where they would be required in ordinary writing. Note that this prohibition would also apply to a non-spacing mark following a pre-composed code point containing the same diacritic. A more general mitigation technique would be to limit nonspacing marks to known combinations which can be enumerated. Where that is not possible for some scripts, some other context restrictions can usually be applied.

There are some writing systems where characters do not normally occur in arbitrary locations in the context of each syllable. Neither users nor rendering systems for such scripts are adept at handling arbitrary sequences of such characters. While some latitude beyond strict spelling rules may be accommodated, policies that enforce a minimal set of structural rules are required to ensure that users can identify the identifier and systems can render them predictably.

5.5.2. Limits of Mitigation

As noted in Section 1, it is not possible to solve all the problems with identifier systems, particularly when human factors are taken into account. In addition, each of the mitigation approaches has its own limits of the type of problems that can be addressed, whether it is by exclusion of specific code points; requiring or prohibiting contexts for certain code points; restriction to a single script per label; or mutual exclusion of labels differing only by code points identical or otherwise confusably equivalent to other code points. Additional policies may be needed to prevent registration of labels that are problematic or confusable for other reasons.

There are a number of issues in implementing and presenting identifiers to the user which are not specific to individually identifiable code points (or sequences). For example, fonts can vary widely in whether they make or do not make a distinction in appearance of characters; relying on the native reader to get the intended meaning from context. It is up to user agents to make sure to select fonts that render each code point as distinct as possible.

When new code points are assigned in Unicode, systems, keyboards, fonts and rendering engines may all be updated unevenly, with considerable delays. During a possibly lengthy transition period, this will lead to inconsistent user experience or inability to distinguish certain labels. Even if unsupported labels are presented as A-labels, users may not reliably identify them, because they appear as essentially random sequences of letters and digits.

5.6. Notes

In the explanation the character names have been abbreviated. The following list shows sample entries for the proposed registry. It is non-normative, and only included for illustrative purposes. Also see the examples below (Appendix B).

6. Table of Code Points

```
-----------------------------------------------------------------
  Code Point: 01C0
  Related CP:
  References: [120] [155]
  Comment:    Not Recommended: Indistinguishable from a
              punctuation character that is not PVALID
-----------------------------------------------------------------
  Code Point: 01C1
  Related CP:
  References: [120] [155]
  Comment:    Not Recommended: Indistinguishable from a
              punctuation character that is not PVALID
-----------------------------------------------------------------
  Code Point: 01C2
  Related CP:
  References: [120]
  Comment:    Not Recommended: Indistinguishable from a
              punctuation character that is not PVALID
-----------------------------------------------------------------
  Code Point: 01C3
  Related CP:
  References: [120] [150]
  Comment:    Not Recommended: Indistinguishable from a
```

```
              punctuation character that is not PVALID
   ---------------------------------------------------------------
     Code Point: 01DD
     Related CP: 0259
     References: [150]
     Comment:    Identical: Identical in appearance to U+0259
   ---------------------------------------------------------------
     Code Point: 0259
     Related CP: 01DD
     References: [150]
     Comment:    Identical: Identical in appearance to U+01DD
   ---------------------------------------------------------------
     Code Point: 0131
     Related CP:
     References: [100]
     Comment:    Restricted Context: If followed by any combining
                 mark above, renders the same way as U+0069 in any
                 good font. Should be restricted to where it is not
                 followed by a combining mark above
   ---------------------------------------------------------------
     Code Point: 0237
     Related CP:
     References: [115]
     Comment:    Not Recommended: If followed by any combining mark
                 above, renders the same way as U+006A in any good
                 font. As its use is limited, it is best excluded.
   ---------------------------------------------------------------
     Code Point: 025F
     Related CP:
     References: [115]
     Comment:    Not Recommended: If followed by any combining mark
                 above, renders the same way as U+0249 in any good
                 font. As its use is limited, it is best excluded.
   ---------------------------------------------------------------
     Code Point: 02A3
     Related CP: 0064 007A
     References: [115]
     Comment:    Not Recommended: Looks like small LETTER D plus
                 LETTER Z, except for slight kerning; in limited
                 use.
   ---------------------------------------------------------------
     Code Point: 02A6
     Related CP: 0074 0073
     References: [115]
     Comment:    Not Recommended: Looks like small LETTER T plus
                 LETTER S, except for slight kerning; in limited
                 use.
   ---------------------------------------------------------------
```

```
  Code Point: 02A7
  Related CP: 0074 0283
  References: [115]
  Comment:    Not Recommended: Looks like small LETTER T plus
              LETTER ESH, except for slight kerning; in limited
              use.
---------------------------------------------------------------
  Code Point: 02AA
  Related CP: 006C 0073
  References: [115]
  Comment:    Not Recommended: Looks like small LETTER L plus
              LETTER S, except for slight kerning; in limited
              use.
---------------------------------------------------------------
  Code Point: 02AB
  Related CP: 006C 007A
  References: [115]
  Comment:    Not Recommended: Looks like small LETTER L plus
              LETTER Z, except for slight kerning; in limited
              use.
---------------------------------------------------------------
  Code Point: 02B9
  Related CP:
  References: [120]
  Comment:    Not Recommended: Indistinguishable from a
              punctuation character that is not PVALID
---------------------------------------------------------------
  Code Point: 02BA
  Related CP:
  References: [120]
  Comment:    Not Recommended: Indistinguishable from a
              punctuation character that is not PVALID
---------------------------------------------------------------
  Code Point: 02BB
  Related CP:
  References: [120]
  Comment:    Not Recommended: Indistinguishable from a
              punctuation character that is not PVALID
---------------------------------------------------------------
  Code Point: 02BC
  Related CP:
  References: [6912]
  Comment:    Not Recommended: Indistinguishable from a
              punctuation character (U+2019), which is not
              PVALID
---------------------------------------------------------------
  Code Point: 02BD
  Related CP:
```

```
  References: [120]
  Comment:    Not Recommended: Indistinguishable from
              punctuation character that is not PVALID
---------------------------------------------------------------
  Code Point: 02BE
  Related CP:
  References: [120]
  Comment:    Not Recommended: Indistinguishable from
              punctuation character that is not PVALID
---------------------------------------------------------------
  Code Point: 02BF
  Related CP:
  References: [120]
  Comment:    Not Recommended: Indistinguishable from
              punctuation character that is not PVALID
---------------------------------------------------------------
  Code Point: 02C0
  Related CP:
  References: [120]
  Comment:    Not Recommended: Indistinguishable from
              punctuation character that is not PVALID
---------------------------------------------------------------
  Code Point: 02C1
  Related CP:
  References: [120]
  Comment:    Not Recommended: Indistinguishable from
              punctuation character that is not PVALID
---------------------------------------------------------------
  Code Point: 02C6
  Related CP:
  References: [120]
  Comment:    Not Recommended: Indistinguishable from
              punctuation character that is not PVALID
---------------------------------------------------------------
  Code Point: 02C7
  Related CP:
  References: [120]
  Comment:    Not Recommended: Indistinguishable from
              punctuation character that is not PVALID
---------------------------------------------------------------
  Code Point: 02C8
  Related CP:
  References: [120]
  Comment:    Not Recommended: Indistinguishable from
              punctuation character that is not PVALID
---------------------------------------------------------------
  Code Point: 02C9
  Related CP:
```

```
  References: [120]
  Comment:    Not Recommended: Indistinguishable from
              punctuation character that is not PVALID
---------------------------------------------------------------
  Code Point: 02CA
  Related CP:
  References: [120]
  Comment:    Not Recommended: Indistinguishable from
              punctuation character that is not PVALID
---------------------------------------------------------------
  Code Point: 02CB
  Related CP:
  References: [120]
  Comment:    Not Recommended: Indistinguishable from
              punctuation character that is not PVALID
---------------------------------------------------------------
  Code Point: 0300
  Related CP:
  References: [100]
  Comment:    Not Recommended: Not recommended other than as
              part of enumerated sequences
---------------------------------------------------------------
  Code Point: 0301
  Related CP:
  References: [100]
  Comment:    Not Recommended: Not recommended other than as
              part of enumerated sequences
---------------------------------------------------------------
  Code Point: 0302
  Related CP:
  References: [100]
  Comment:    Not Recommended: Not recommended other than as
              part of enumerated sequences
---------------------------------------------------------------
  Code Point: 0303
  Related CP:
  References: [100]
  Comment:    Not Recommended: Not recommended other than as
              part of enumerated sequences
---------------------------------------------------------------
  Code Point: 0304
  Related CP:
  References: [100]
  Comment:    Not Recommended: Not recommended other than as
              part of enumerated sequences
---------------------------------------------------------------
  Code Point: 0306
  Related CP:
```

```
  References: [100]
  Comment:    Not Recommended: Not recommended other than as
              part of enumerated sequences
---------------------------------------------------------------
  Code Point: 0307
  Related CP:
  References: [115]
  Comment:    Restricted Context: By definition, LATIN SMALL
              LETTER I plus combining DOT ABOVE renders exactly
              the same as LATIN SMALL LETTER I by itself and
              does so in practice for any good font. The same is
              true for all Unicode characters with the
              soft_dotted property; they lose their dot if
              followed by a combining mark. DOT ABOVE should be
              excluded, or restricted to  contexts where it does
              not follow a soft_dotted letter.
---------------------------------------------------------------
  Code Point: 0308
  Related CP:
  References: [100]
  Comment:    Not Recommended: Not recommended other than as
              part of enumerated sequences
---------------------------------------------------------------
  Code Point: 0624
  Related CP: 0648
  References: [201]
  Comment:    Identical: Identical in appearance in some
              positional form and/or not reliably distinguished
              because of small size of distinguishing features
---------------------------------------------------------------
  Code Point: 0625
  Related CP: 0622, 0623, 0627, 0672
  References: [201]
  Comment:    Identical: Identical in appearance in some
              positional form and/or not reliably distinguished
              because of small size of distinguishing features
---------------------------------------------------------------
  Code Point: 0626
  Related CP: 0649, 064A, 067B, 06CC, 06CD, 06D0, 06D2
  References: [201]
  Comment:    Identical: Identical in appearance in some
              positional form and/or not reliably distinguished
              because of small size of distinguishing features
---------------------------------------------------------------
  Code Point: 0627
  Related CP: 0622, 0623, 0625, 0672
  References: [201]
  Comment:    Identical: Identical in appearance in some
```

positional form and/or not reliably distinguished because of small size of distinguishing features

---

Code Point: 064B
Related CP:
References: [5564]
Comment: Not Recommended: Not to be used in zone files for the Arabic language, per RFC 5564

Code Point: 064C
Related CP:
References: [5564]
Comment: Not Recommended: Not to be used in zone files for the Arabic language, per RFC 5564

Code Point: 065C
Related CP:
References: [300]
Comment: Not Recommended: Part of homoglyph sequence(s) not covered by normalization.

Code Point: 0660
Related CP: 06F0
References: [110]
Comment: Identical: Identical in appearance and meaning to EXTENDED ARABIC-INDIC DIGIT ZERO

Code Point: 0661
Related CP: 06F1
References: [110]
Comment: Identical: Identical in appearance and meaning to EXTENDED ARABIC-INDIC DIGIT ONE

Code Point: 077F
Related CP:
References: [115]
Comment: Not Recommended: Obsolote (archaic)

Code Point: 08AA
Related CP:
References: [201]
Comment: Not Recommended: No evidence of active use found; not recommended

---

Code Point: 0A72 0A3F
Related CP: 0A07
References: [401]
Comment: Not Recommended: Do not use for U+0A07

```
---------------------------------------------------------------
  Code Point: 0A72 0A40
  Related CP: 0A08
  References: [401]
  Comment:    Not Recommended: Do not use for U+0A08
---------------------------------------------------------------
  Code Point: 0E3A
  Related CP:
  References: [206]
  Comment:    Other issue: Renders unreliably, or not at all, if
              adjacent to any Thai vowel below. This may be
              prevented by a context rule
---------------------------------------------------------------
  Code Point: 0E41
  Related CP:
  References: [206]
  Comment:    Restricted Context: Digraph of U+0E40 SARA E
              U+0E40 SARA E. Normally handled by disallowing the
              sequence via a context rule
---------------------------------------------------------------
  Code Point: 0E45
  Related CP:
  References: [206]
  Comment:    Restricted Context: Only occurs after two special
              Thai vowels,U+0E24 RU and U+0E26 LU. Is also
              potentially confused with U+0E32 SARA I. Both
              issues can be addressed by defining a context
              rule. Alternatively the context may be spelled out
              by enumerating the two sequences and excluding
              U+0E45 if occurring by itself.
---------------------------------------------------------------
  Code Point: 0E4E
  Related CP:
  References: [206]
  Comment:    Not Recommended: Rarely used in modern Thai; it is
              more commonly replaced with U+0E3A (PHINTHU).
              Excluding it avoids issues with confusing it with
              another diacritic U+0E4C (THANTHAKHAT). Both are
              rendered atop a syllable and hard to distinguish
              at small sizes.
---------------------------------------------------------------
  Code Point: 12A5
  Related CP: 12D5
  References: [100] [202]
  Comment:    Interchangeable: U+12A5 and U+12D5 are used
              interchangeably in Amharic
---------------------------------------------------------------
  Code Point: 12A6
```

```
  Related CP: 12D6
  References: [100] [202]
  Comment:    Interchangeable: U+12A6 and U+12D6 are used
              interchangeably in Amharic
---------------------------------------------------------------
  Code Point: 17D2 178A
  Related CP: 17D2 178F
  References: [204]
  Comment:    Identical: When preceded by U+17D2, U+178A and
              U+178F are indistinguishable
---------------------------------------------------------------
  Code Point: 17D2 178F
  Related CP: 17D2 178A
  References: [204]
  Comment:    Identical: When preceded by U+17D2, U+178A and
              U+178F are indistinguishable
---------------------------------------------------------------
```

6.1. References for Registry

[99] The Unicode Consortium, "The Unicode Standard", (latest version) http:www.unicode.org/versions/latest (Multiple, or latest version)

[100] Integration Panel, "Maximal Starting Repertoire (MSR-2)", April 2015, https://www.icann.org/en/system/files/files/msr-2-overview-14apr15-en.pdf (Code points included in MSR-2 as potentially appropriate for the root zone)

[115] Integration Panel, "Maximal Starting Repertoire (MSR-2)", April 2015, https://www.icann.org/en/system/files/files/msr-2-overview-14apr15-en.pdf (Code points excluded from MSR-2 as inappropriate for the root zone)

[120] Integration Panel, "Maximal Starting Repertoire (MSR-2)", April 2015, https://www.icann.org/en/system/files/files/msr-2-overview-14apr15-en.pdf (Code points considered problematic by MSR-2)

[150] The Unicode Consortium, "Intentional.txt", Version 10.0.0, http://www.unicode.org/Public/security/10.0.0/intentional.txt (Code points considered identical by intention)

[155] "Proposal to Update Identical.txt", L2 17/301 (and revisions) http://www.unicode.org/L2/L2017/17301-update-intentional.pdf (Code points considered identical by intention)

[201] TF-AIDN, "Proposal for Arabic Script Root Zone LGR", 18 November 2015 https://www.icann.org/en/system/files/files/arabic-lgr-proposal-18nov15-en.pdf (In-script variants and code points excluded)

[202] Ethiopic Generation Panel, "Proposal for Ethiopic Script Root Zone LGR", May 17, 2017, https://www.icann.org/en/system/files/files/proposal-ethiopic-lgr-17may17-en.pdf ()

[204] Khmer Generation Panel, "Proposal for Khmer Script Root Zone Label Generation Rules (LGR)", August 15, 2016, https://www.icann.org/en/system/files/files/proposal-khmer-lgr-15aug16-en.pdf ()

[206] Thai Generation Panel, "Proposal for the Thai Script Root Zone LGR", May 25, 2017 https://www.icann.org/en/system/files/files/proposal-thai-lgr-25may17-en.pdf ()

[300] Internationalized Domain Names Variant Issues Project: Arabic Case Study Team Issues Report, ICANN, October 7, 2011 https://archive.icann.org/en/topics/new-gtlds/arabic-vip-issues-report-07oct11-en.pdf (In-script variants and code points excluded)

[401] Table 12-14 in Chapter 12 "South and Central Asia-I", ,"The Unicode Standard", Version 10.0, https://www.unicode.org/versions/Unicode10.0.0/ch12.pdf (Vowel sequences not to be used in Gurmukhi)

[5564] RFC 5564 (Code points to be excluded from repertoires for the Arabic language)

[6912] RFC 6912 (Code points considered problematic)

7. IANA Considerations

The IANA Services Operator is hereby requested to create the Registry of Unicode Code Points for Special Consideration in Network Identifiers, and to populate it with the values in section Section 5. The registry is to be updated by Expert Review.

This registry has no formal protocol status with respect to IDNA or PRECIS. It is a registry intended to be used by those creating registration or lookup policies, in order to inform the development of such policies.

8. Security Considerations

The registry established by this document is intended to help operators of identifier systems in deciding what to permit in identifiers. It may also be useful for user agents that attempt to provide warnings to users about suspicious or inadvisable identifiers. Operators that fail to make policies addressing the contents of the registry may permit the creation of identifiers that are misleading or that may be used in attacks on the network or users.

The registry is not a magic solution to all identifier ambiguity, and even refusing to permit registration of, or lookup of, every code point in the registry cannot ensure that misleading or confusing identifiers will never be created.

9. References

9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4713] Lee, X., Mao, W., Chen, E., Hsu, N., and J. Klensin, "Registration and Administration Recommendations for Chinese Domain Names", RFC 4713, DOI 10.17487/RFC4713, October 2006, <https://www.rfc-editor.org/info/rfc4713>.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <https://www.rfc-editor.org/info/rfc5890>.

[RFC5891] Klensin, J., "Internationalized Domain Names in Applications (IDNA): Protocol", RFC 5891, DOI 10.17487/RFC5891, August 2010, <https://www.rfc-editor.org/info/rfc5891>.

[RFC5892] Faltstrom, P., Ed., "The Unicode Code Points and Internationalized Domain Names for Applications (IDNA)", RFC 5892, DOI 10.17487/RFC5892, August 2010, <https://www.rfc-editor.org/info/rfc5892>.

[RFC5893] Alvestrand, H., Ed. and C. Karp, "Right-to-Left Scripts for Internationalized Domain Names for Applications (IDNA)", RFC 5893, DOI 10.17487/RFC5893, August 2010, <https://www.rfc-editor.org/info/rfc5893>.

[RFC5894] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Background, Explanation, and Rationale", RFC 5894, DOI 10.17487/RFC5894, August 2010, <https://www.rfc-editor.org/info/rfc5894>.

[RFC7564] Saint-Andre, P. and M. Blanchet, "PRECIS Framework: Preparation, Enforcement, and Comparison of Internationalized Strings in Application Protocols", RFC 7564, DOI 10.17487/RFC7564, May 2015, <https://www.rfc-editor.org/info/rfc7564>.

[RFC7940] Davies, K. and A. Freytag, "Representing Label Generation Rulesets Using XML", RFC 7940, DOI 10.17487/RFC7940, August 2016, <https://www.rfc-editor.org/info/rfc7940>.

[UAX44] The Unicode Consortium, "Unicode Standard Annex #44, Unicode Character Database", <http://www.unicode.org/reports/tr44/>.

This references the most currently published version of the description of the Unicode Character Database.

[UCD] The Unicode Consortium, "Unicode Character Database", <http://www.unicode.org/Public/UCD/latest/ucd/>.

This references the most currently published version of the data files for the Unicode Character Database

[Unicode] The Unicode Consortium, "The Unicode Standard, Latest Version", <http://www.unicode.org/versions/latest/>.

This references the most currently published version

9.2. Informative References

[I-D.klensin-idna-5892upd-unicode70]
Klensin, J. and P. Faltstrom, "IDNA Update for Unicode 7.0 and Later Versions", draft-klensin-idna-5892upd-unicode70-05 (work in progress), October 2017.

[I-D.rfc5891bis]
Klensin, J., "Internationalized Domain Names in Applications (IDNA): Registry Restrictions and Recommendations", March 2017, <https://datatracker.ietf.org/doc/draft-klensin-idna-rfc5891bis/>.

[MSR] Integration Panel, ""Maximal Starting Repertoire (MSR-3)"", March 2018, <https://www.icann.org/en/system/files/files/msr-3-overview-28mar18-en.pdf>.

[RFC5564] El-Sherbiny, A., Farah, M., Oueichek, I., and A. Al-Zoman, "Linguistic Guidelines for the Use of the Arabic Language in Internet Domains", RFC 5564, DOI 10.17487/RFC5564, February 2010, <https://www.rfc-editor.org/info/rfc5564>.

[RFC6365] Hoffman, P. and J. Klensin, "Terminology Used in Internationalization in the IETF", BCP 166, RFC 6365, DOI 10.17487/RFC6365, September 2011, <https://www.rfc-editor.org/info/rfc6365>.

[RFC8228] Freytag, A., "Guidance on Designing Label Generation Rulesets (LGRs) Supporting Variant Labels", RFC 8228, DOI 10.17487/RFC8228, August 2017, <https://www.rfc-editor.org/info/rfc8228>.

[RZ-LGR] Integration Panel, ""Root Zone Label Generation Rules (LGR-2) - Overview and Summary"", July 2017, <https://www.icann.org/sites/default/files/lgr/lgr-2-overview-26jul17-en.pdf>.

Appendix A. Additional Background

A.1. The Theory of Inclusion

The mechanism that the IETF has come to prefer for internationalization of identifiers may be called "inclusion-based identifier internationalization", or "inclusion" for short. Under inclusion, the characters that are permissible in identifiers for a protocol are selected from the set of all Unicode characters. One starts with an empty set of characters, and then gradually adds characters to the set, usually based on Unicode properties (see below, and also Section 3).

Inclusion depends in part on assumptions the IETF made when the strategy was adopted and developed; some of those assumptions were about the relationships between different characters and the

likelihood that similar such relationships would get added to future versions of Unicode. Those assumptions turn out not to have been true in every case. Code points at issue are among those to be listed in the registry defined here. (See Section 5.)

The intent of Unicode is to encode all known writing systems into a single coded character set. One consequence of that goal is that Unicode encodes an enormous number of characters. Another is that the work of Unicode does not end until every writing system is encoded; even after that, it needs to continue to track any changes in those writing systems.

Unicode encodes abstract characters, not glyphs. Because of the way Unicode was built up over time, there are sometimes multiple ways to encode the same abstract character. For example, an e with an acute accent may be written by combining U+0065 LATIN SMALL LETTER E and U+0031 COMBINING ACUTE ACCENT, or it may be written U+00E9 LATIN SMALL LETTER E WITH ACUTE. If Unicode encodes an abstract character in more than one way, then for most purposes the different encodings should all be treated as though they’re the same character. This "canonical equivalence" between encodings of the same abstract characters is explicitly called out by Unicode. A lack of a defined canonical equivalence is tantamount to an assertion by Unicode that the two encodings do not represent the same abstract character, even if both happen to result in the same appearance.

Every encoded character in Unicode (more precisely, every code point) is associated with a set of properties. The properties define what script a code point is in, whether it is a letter or a number or punctuation and so forth, its direction when written, to what other code point or code point sequence it is canonically equivalent, and many other properties. These properties are important to the inclusion mechanism. They are defined in the Unicode Character Database [UCD] [UAX44].

Inclusion depends on the assumption that such strings as will be used in identifiers will not have any ambiguous matching to other strings. In practice, this means that input strings to the protocol are expected to be in Normalization Form C. This way, any alternative sequences of code points for the same characters will be normalized to a single form. If all the characters in the string are also included for the protocol’s candidate identifiers, then the string is eligible to be an identifier under the protocol.

A.2. The Difference Between Theory and Practice

In principle, under inclusion identifiers should be unambiguous. It has always been recognized, however, that for humans some ambiguity is inevitable, because of the vagaries of writing systems and of human perception.

Normalization Form C ("NFC") removes the ambiguities based on dual or multiple encoding for the same abstract character. However, characters are not the same as their glyphs. This means that it is possible for certain abstract characters to share a glyph. We can call such abstract characters "homoglyphs". While this looks at first like something that should be handled (or should have been handled) by normalization (NFC or something else), there are important differences; the situation is in some sense an extreme case of a spectrum of ambiguity.

A.2.1. Confusability

While Unicode deals in abstract characters and inclusion works on Unicode code points, users interact with strings as actually rendered: sequences of glyphs. There are characters that, depending on font, sometimes look quite similar to one another (such as "l" and "1"); any character that is like this is often called "visually similar". More difficult are characters that, in any normal rendering, always look the same as one another. The shared history of Cyrillic, Greek, and Latin scripts, for example, means that there are characters in each script that function similarly and that are usually indistinguishable from one another, though they are not the same abstract character. These are examples of "homoglyphs." Any character that can be confused for another one can be called confusable, and confusability can be thought of as a spectrum with "visually similar" at one end, and "homoglyphs" at the other. (We use the term "homoglyph" strictly: code points that normally use the same glyph when rendered.)

Note that homoglyphs are not restricted to cross-script scenarios - there are a number of homoglyphs where both code points or sequences are part of the same script.

A further issue is introduced by the fact that Unicode caters not only to living and dead languages alike, but also to scholarly and scientific notation, as well as specialized modes of written text, such as for poetry, religious works, or texts to be sung or chanted. Where these notations use symbols, they are excluded under inclusion, but where they use varieties of letter forms or marks used with letters, they are included by default. Some of these letters or marks, have been incorporated over time into orthographies for living

languages, which is one reason they were not rigorously excluded from the start. However, in some cases, they may (alone or in combination with ordinary letters appear the same (or very similar to) existing letters. This makes some of these characters, and especially the marks in question "troublesome".

Finally, IDNA 2008 has a limited appreciation for the fact that characters in complex scripts, unlike ASCII letters, cannot simply occur in random sequences. Neither software (for display or data entering) nor readers are prepared to process some of these code points "out of order". For such scripts, without a policy that describes permissible contexts, labels could be registered that cannot be rendered or typed reliably and which most users would not know how to read or recognize. In some cases, combining sequences typed in the "wrong" order may display identically to to those typed in the "correct" ordering; again something that needs to be sorted out by defining permissible contexts, for example by using the context rule mechanism in [RFC7940].

Appendix B. Examples

There are a number of cases that illustrate the combining sequence or digraph issue:

U+08A1 vs \u'0628'\u'0654' This case is ARABIC LETTER BEH WITH HAMZA ABOVE, which is the one that was detected during expert review that caused the IETF to first notice the issue, even though the issue existed before this. For detailed discussion of this case and some of the following ones, see [I-D.klensin-idna-5892upd-unicode70].

U+0681 vs \u'062D'\u'0654' This case is ARABIC LETTER HAH WITH HAMZA ABOVE, which (like U+08A1) does not have a canonical equivalent. In both cases, the places where hamza above and similar Arabic combining marks are used are specialized enough that the combining marks are generally excluded. See [RFC5564] and [RZ-LGR]. Unicode has a policy of encoding as composite any letter needed in an Arabic orthography, even if it appears superficially that the same shape could be achieved by a combining sequence. (In actual typography there's often a small but noticeable difference in placement of the mark between a composite character and a combining sequence.)

U+0623 vs \u'0627'\u'0654' This case is ARABIC LETTER ALEF WITH HAMZA ABOVE. Unlike the previous two cases, it does have a canonical equivalence with the combining sequence. Therefore, only the composite is used in IDNs.

U+09E1 vs u\’098C’u\’09E2’ This case is BENGALI LETTER VOCALIC LL. This is an example in the Bengali script of a case without a canonical equivalence to the combining sequence. Per Unicode, the single code point should be used to represent vowel signs in text, and the sequence of code points should not be used. There are similar cases in many Indic scripts. It is not a simple matter of disallowing the combining vowel mark in cases like this, because it is commonly used as vowel sign. The recommendation would be to add a context rule, restricting the vowel signs from appearing directly after an independent vowel like U+098C..

U+019A vs \u’006C’\u’0335’ This case is LATIN SMALL LETTER L WITH BAR. In at least some fonts, there is a detectable difference between the composite code point and the combining sequence, but only if one compares them side-by-side. Unlike a separable diacritic, there are no fast rules for placement of overlays. A bar may cross at different heights for different glyph shape or may cross different parts of the glyph. For this reason, there is no canonical equivalence defined between the sequence and the composite. Unicode has a principle of encoding barred letters of specific shape as single code point composites when needed for any writing system. The code point U+0335 COMBINING SHORT STROKE OVERLAY and similar overlay diacritics are therefore never needed as part of any orthography and are recommended to be excluded from identifiers.

U+00F8 vs \u’006F’\u’0337’ This is LATIN SMALL LETTER O WITH STROKE. The effect is similar to the previous case. Unicode has a principle of encoding stroked letters as composites when needed for any writing system.

U+02A6 vs \u’0074’\u’0073’ This is LATIN SMALL LETTER TS DIGRAPH, which is not canonically equivalent to the letters t and s. The intent appears to be that the digraph shows the two shapes as kerned, but the difference may be slight if viewed out of context. The use of the digraph is for specialized purposes; it can be excluded from identifiers.

U+01C9 vs \u’006C’\u’006A’ Unlike the TS digraph, the LJ digraph has a relevant compatibility decomposition, so it fails the relevant stability rules under inclusion and is therefore DISALLOWED in IDNA2008. This illustrates the way that consistencies that might be natural to some users of a script are not necessarily found in it, possibly because of uses by another writing system.

U+06C8 vs u\’0648’u\’0670’ ARABIC LETTER YU is an example where the normally-rendered character looks just like a combining sequence, but are named differently. This an example that shows that the

Unicode name is not a reliable indicator of the intended appearance. Like other cases in Arabig, the recommendation is to exclude the combining mark (and therefore the sequence) in favor of the composite.

U+069 vs \u'0069'\u'0307' LATIN SMALL LETTER I followed by COMBINING DOT ABOVE by definition, renders exactly the same as LATIN SMALL LETTER I by itself and does so in practice for any good font. The same would be true if "i" was replaced with any of the other Soft_Dotted characters defined in Unicode. The character sequence \u'0069'\u'0307' (followed by no other combining mark) is reportedly rather common on the Internet. Because base character and stand-alone code point are the same in this case, and the code points affected have the Soft_Dotted property already, this could be mitigated separately via a context rule affecting U+0307.

Other cases that demonstrate that the issue does not lie exclusively or primarily with combining sequences:

U+0B95 vs U+0BE7 The TAMIL LETTER KA and TAMIL DIGIT ONE are always indistinguishable, but needed to be encoded separately because one is a letter and the other is a digit.

Arabic-Indic Digits vs. Extended Arabic-Indic Digits Seven digits of these two sequences have entirely identical shapes. This case is an example of something dealt with in inclusion that nevertheless can lead to confusions that are not fully mitigated. IDNA, for example, contains context rules restricting the digits to one set or another; but such rules apply only to a single label, not to an entire name. Moreover, it provides no way of distinguishing between two labels that both conform to the context rule, but where each contains a different member one of the seven identical shape pairs.

U+53E3 vs U+56D7 These are two Han characters (roughly rectangular) that are different when laid side by side; but they may be difficult to distinguish out of context or in very small print.

U+01DD vs U+0259 The two Latin script code points share the have the identical appearance of a lower-case upside down "e". They are encoded differently due to different uppercase forms. The fact that they uppercase differently is taken as evidence that they are not the same abstract character, despite the superficial evidence of their shared shape. The more common cases, where the uppercase forms are identical may be of less concern, given that IDNA 2008 is limited to lower case.

Cross script homoglyphs usually do not involve combining sequences, but can be mitigated by rules requiring strings to be in a single script. For zones that support multiple scripts, it may be necessary to have policies to prevent whole-script homographs: labels entirely in one script that look the same as another label in the other script. One method would be to define "blocked" variants (See [RFC7940] and [RFC8228]).

LATIN SMALL LETTER OPEN E is one of a handful of examples of characters borrowed from another script, in this case GREEK SMALL LETTER EPSILON.

LATIN SMALL LETTER E and CYRILLIC SMALL LETTER IE are historically related, both derive from uppercase forms of the GREEK CAPTIAL LETTER EPSILON. There are a number of such pairs -- enough to make many whole strings that look the same in both scripts (but usually spell nonsense in one of them). An example would be "pax".

Appendix C. Discussion Venue

Note to RFC Editor: this section should be removed prior to publication as an RFC.

This Internet-Draft may be discussed on the IAB Internationalization public list: i18n-discuss@iab.org.

Appendix D. Change History

Note to RFC Editor: this section should be removed prior to publication as an RFC.

00:

* Initial version

01:

* Add background and examples from the LUCID Problem Statement
* Add a paragraph about motivation to explain the difference between this registry and administrative policy more generally
* Expand and clarify a number of earlier points of discussion
* Attempt to make clear that this registry does not update any protocols

* Move some formerly-appendix material to the body

* Expand the initial registry.

02:

* Expanded the discussion of possible mitigation approaches and made its own section.

* Added more detail to the categories of troublesome characters

* Minor updates to "Existing techniques" section.

* Some extension to the description of the contents of the registry and discussion of how to handle additional information.

Authors' Addresses

Asmus Freytag
ASMUS, Inc.

Email: asmus@unicode.org

John C Klensin
1770 Massachusetts Ave, Ste 322
Cambridge, MA 02140
U.S.A.

Email: john-ietf@jck.com

Andrew Sullivan
Oracle Corp.
100 Milverton Drive
Missisauga, ON L5R 4H1
Canada

Email: andrew.s.sullivan@oracle.com

Network Working Group
Internet-Draft
Updates: 5892, 5894 (if approved)
Intended status: Standards Track
Expires: April 11, 2018

J. Klensin

P. Faltstrom
Netnod

October 8, 2017

IDNA Update for Unicode 7.0 and Later Versions
draft-klensin-idna-5892upd-unicode70-05

Abstract

The current version of the IDNA specifications anticipated that each new version of Unicode would be reviewed to verify that no changes had been introduced that required adjustments to the set of rules and, in particular, whether new exceptions or backward compatibility adjustments were needed. The review for Unicode 7.0.0 first identified a potentially problematic new code point and then a much more general and difficult issue with Unicode normalization. This specification discusses those issues and proposes updates to IDNA and, potentially, the way the IETF handles comparison of identifiers more generally, especially when there is no associated language or language identification. It also applies an editorial clarification to RFC 5892 that was the subject of an earlier erratum and updates RFC 5894 to point to the issues involved.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 11, 2018.

Table of Contents

1. Introduction

Note in/about -04 and -05 Drafts: These two versions of the document contains a very large amount of new material as compared to the -03 version. The new material reflects an evolution of community understanding in the first quarter of 2015 and further evolution between then and mid-2017 from an assumption that the problem involved only a few code points and one combining character in a single script (Hamza Above and Arabic) to an understanding that the problem we have come to call "non-decomposing code points" and several closely related ones are quite pervasive and may represent fundamental misunderstandings or omissions from IDNA2008 (and, by extension, the basics of PRECIS [RFC8264]) that must be corrected if those protocols are going to be used in a way that supports internationalized identifiers on the Internet predictably (as seen by the end user) and securely.

This version is still necessarily incomplete: not only is our understanding probably still not comprehensive, but there are a number of placeholders for text and references. Nonetheless, the document in its current form should be useful as both the beginning of a comprehensive overview is the issues and a source of references to other relevant materials.

This draft could almost certainly be better organized to improve its readability: specific suggestions would be welcome.

1.1. Origins and Discovery of the Issue

The current version of the IDNA specifications, known as "IDNA2008" [RFC5890], anticipated that each new version of Unicode would be reviewed to verify that no changes had been introduced that required adjustments to IDNA's rules and, in particular, whether new exceptions or backward compatibility adjustments were needed. When that review was carefully conducted for Unicode 7.0.0 [Unicode7], comparing it to prior versions including the text in Unicode 6.2 [Unicode62], it identified a problematic new code point (U+08A1, ARABIC LETTER BEH WITH HAMZA ABOVE). The code point was added for Arabic Script use with the Fula (also known as Fulfulde, Pulaar, amd Pular'Fulaare) language. That language is apparently most often written in Latin characters today [Omniglot-Fula] [Dalby] [Daniels].

The specific problem is discussed in detail in Section 3. In very broad terms, IDNA (and other IETF work) assume that, if one can represent "the same character" either as a combining sequence or as a single code point, strings that are identical except for those alternate forms will compare equal after normalization. Part of the difficulty that has characterized this discussion is that "the same" differs depending on the criteria that are chosen. It may be further complicated in practice by differences in preferred type styles or rendering, but Unicode code point choices are not supposed to depend on type style (font) variations and, again, IDNA has no mechanism for specifying language choices that might affect rendering.

The behavior of the newly-added code point, while non-optimal for IDNA, follows that of a few code points that predate Unicode 7.x and even the IDNA 2008 specifications and Unicode 6.0. Those existing code points, which may not be easy to accurately characterize as a group, make the question of what, if anything, to do about this new exceedingly problematic one and, perhaps separately, what to do about existing sets of code points with the same behavior, because different reasonable criteria yield different decisions, specifically:

- To disallow it (and future, but not existing, characters with similar characteristics) as an IDNA exception case creates inconsistencies with how those earlier code points were handled.

- To disallow it and the similar code points as well would necessitate invalidating some potential labels that would have been valid under IDNA2008 until this time. Depending on how the

collection of similar code points is characterized, a few of them are almost certainly used in reasonable labels.

- To permit the new code point to be treated as PVALID creates a situation in which it is possible, within the same script, to compose the same character symbol (glyph or grapheme) in two different ways that do not compare equal even after normalization. That condition would then apply to it and the earlier code points with the same behavior. That situation contradicts a fundamental assumption of IDNA that is discussed in more detail below.

NOTE IN DRAFT:

This working draft discusses six alternatives, including an idea (an IETF-specific normalization form) that seemed too drastic to be considered when IDNA2008 was designed or even when the review of Unicode 7.0 for IDAN purposes began. In retrospect, it not only would have been appropriate to discuss when the IDNA2008 specifications were being developed but is appearing more attractive now. The authors suggest that the community discuss the relevant tradeoffs and make a decision and that the document then be revised to reflect that decision, with the other alternatives discussed as options not chosen. Because there is no ideal choice, the discussion of the issues in Section 3 is probably as or more important than the particular choice of how to handle this code point. In addition to providing information for this document, that section should be considered as an updating addendum to RFC 5894 [RFC5894] and should be incorporated into any future revision of that document.

As the result of this version of the document containing several alternate proposals, some of the text is also a little bit redundant. That will be corrected in future versions.

1.2. IDNA2008 and Special or Exceptional Cases

IDNA2008 contains several type of explicit provisions for characters (code points) that require special treatment when the requirements of the DNS cannot easily be met by calculations based on stable Unicode properties. Those provisions are
[[CREF1: ... to be supplied]]

As anticipated when IDNA2008, and RFC 5892 in particular, were written, exceptions and explicit updates are likely to be needed only if there is disagreement between the Unicode Consortium’s view about what is best for the Standard and its very diverse user community and the IETF’s view of what is best for IDNs, the DNS, and IDNA. It was hoped that a situation would never arise in which the the two

perspectives would disagree, but the possibility was anticipated and considerable mechanism added to RFC 5890 and 5982 as a result. It is probably important to note that a disagreement in this context does not imply that anyone is "wrong", only that the two different groups have different needs and therefore criteria about what is acceptable. In particular, it appears that the Unicode Consortium has made assumptions about the availability (by explicit designation or context) of information about applicable languages or other context for a give string that are not possible for IDNA. For that reason, the IETF has, in the past, allowed some characters for IDNA that active Unicode Technical Committee members suggested be disallowed to avoid a change in derived tables [RFC6452]. This document describes a set of cases for which the IETF must consider disallowing sets of characters that the various properties would otherwise treat as PVALID.

This document provides the "flagging for the IESG" specified by Section 5.1 of RFC 5892. As specified there, the change itself requires IETF review because it alters the rules of Section 2 of that document.

[[RFC Editor: please remove the following comment and note if they get to you.]]

[[IESG: It might not be a bad idea to incorporate some version of the following into the Last Call announcement.]]

NOTE IN DRAFT to IETF Reviewers: The issues in this document, and particularly the choices among options for either adding exception cases to RFC 5892 or ignoring the issue, warning people, and hoping the results do not include or enable serious problems, are fairly esoteric. Understanding them requires that one have at least some understanding of how scripts in which precomposed characters are preferred over combining sequences as a Unicode design and extension principle work. Those scripts include Arabic but, unlike the assumption when the issues were first discovered, are by no means limited to it. Readers should also understand the reasons the Unicode Standard gives various Arabic Script characters a fairly extended discussion [Unicode70-Arabic] but should treat that only as an example and note that most other cases are much less well documented. It also requires understanding of a number of Unicode principles, including the Normalization Stability rules [UAX15-Versioning] as applied to new precomposed characters and guidelines for adding new characters. There is considerable discussion of the issues in Section 3 and references are provided for those who want to pursue them, but potential reviewers should assume that the background needed to understand the reasons for this change is no less deep in the

subject matter than would be expected of someone reviewing a proposed change in, e.g., the fundamentals of BGP, TCP congestion control, or some cryptographic algorithm. Put more bluntly, one's ability to read or speak languages other than English, or even one or more languages that use the Arabic script or other scripts similarly affected, does not make one an expert in these matters.

1.3. Terminology

This document assumes that the reader is reasonably familiar with the terminology of IDNA [RFC5890] and Unicode [Unicode7] and with the IETF conventions for representing Unicode code points [RFC5137]. Some terms used here may not be used in the same way in those two sets of documents. From one point of view, those differences may have been the results of, or led to, misunderstandings that may, in turn, be part of the root cause of the problems explored in this document. In particular, this document uses the term "precomposed character" to describe characters that could reasonably be composed by a combining sequence using code points with appropriate appearance in common type styles but for which a single code point that does not require combining sequences is available. That definition is strictly about mechanical composition and does not involve any considerations about how the character is used. It is closely related to this document's definition of "identical". When a precomposed character exists and either applying NFC to the combining sequence does not yield that character or applying NFD to that character's code point does not yield the combining sequence, it is referred to in this document as "non-decomposable".

The document also uses some terms that are familiar to those who have been involved with IDNs and IDNA for a long time, but uses them more precisely than may be common in other quarters. For example, the term "Punycode" is not used at all in the rest of this document because it is the name of a very specific encoding algorithm [RFC3492] that does not incorporate the rules and algorithms for domain name labels that are produced by that encoding. Instead, the generic terms "ACE" or "ACE string" for "ASCII-compatible encoding" is used to refer to strings that abstractly contain characters outside the ASCII repertoire [RFC0020] but are encoded so that only ASCII characters appear in the string that would be encountered by a user or protocol and the terms "A-label" and "U-label", as defined in RFC 5890, to refer to the ACE and more conventional (or "native") character forms in which those non-ASCII characters appear in conventional Unicode encodings (typically UTF-8).

2. Document Aspirations

This document, in its present form, is not a proposal for a solution. Instead, it is intended to be (or evolve into) a comprehensive description of the issues and problems and to outline some possible approaches to a solution. A perfect solution -- one that would resolve all of the issues identified in this document -- would involve a relatively small set of relatively simple rules and hence would be comprehensible and predictable for and by non-expert end users, would not require code point by code point or even block by block exception lists, and would not leave uses of any script or language feeling that their particular writing system have been treated less fairly than others.

Part of the reality we need to accept is that IDNA, in its present form, represents compromises that does not completely satisfy those criteria and whatever is done about these issues will probably make it (or the job of administering zones containing IDNs) more complex. Similarly, as the Unicode Standard suggests when it identifies ten Design Principles and the text then says "Not all of these principles can be satisfied simultaneously..." [Unicode70-Design], while there are guidelines and principles, a certain amount of subjective judgment is involved in making determinations about normalization, decomposition, and some property values. For Unicode itself, those issues are resolved by multiple statements (at least one cited below) that one needs to rely on per-code point information in the Unicode Character Database rather than on rules or principles. The design of IDNA and the effort to keep it largely independent of Unicode versions requires rules, categories, and principles that can be relied upon and applied algorithmically. There is obviously some tension between the two approaches.

3. Problem Description

3.1. IDNA assumptions about Unicode normalization

IDNA makes several assumptions about Unicode, Unicode "characters", and the effects of normalization. Those assumptions were based on careful reading of the Unicode Standard at the time [Unicode5], guided by advice and commitments by members of the Unicode Technical Committee. Those assumptions, and the associated requirements, are necessitated by three properties of DNS labels that typically do not apply to blocks of running text:

1. There is no language context for a label. While particular DNS zones may impose restrictions, including language or script restrictions, on what labels can be registered, neither the DNS nor IDNA impose either type of restriction or give the user of a

label any indication about the registration or other restrictions that may have been imposed.

2. Labels are often mnemonics rather than words in any language. They may be abbreviations or acronyms or contain embedded digits and have other characteristics that are not typical of words.

3. Labels are, in practice, usually short. Even when they are the maximum length allowed by the DNS and IDNA, they are typically too short to provide significant context. Statements that suggest that languages can almost always be determined from relatively short paragraphs or equivalent bodies of text do not apply to DNS labels because of their typical short length and because, as noted above, they are not required to be formed according to language-based rules.

At the same time, because the DNS is an exact-match system, there must be no ambiguity about whether two labels are equal. Although there have been extensive discussions about "confusingly similar" characters, labels, and strings, such tests between scripts are always somewhat subjective: they are affected by choices of type styles and by what the user expects to see. In spite of the fact that the glyphs that represent many characters in different scripts are identical in appearance (e.g., basic Latin "a" (U+0061) and the identical-appearing Cyrillic character (U+0430), the most important test is that, if two glyphs are the same within a given script, they must represent the same character no matter how they are formed.

Unicode normalization, as explained in [UAX15], is expected to resolve those "same script, same glyph, different formation methods" issues. Within the Latin script, the code point sequence for lower case "o" (U+006F) and combining diaeresis (U+0308) will, when normalized using the "NFC" method required by IDNA, produce the precomposed small letter o with diaeresis (U+00F6) and hence the two ways of forming the character will compare equal (and the combining sequence is effectively prohibited from U-labels).

NFC was preferred over other normalization methods for IDNA because it is more compact, more likely to be produced on keyboards on which the relevant characters actually appeared, and because it does not lose substantive information (e.g., some types of compatibility equivalence involves judgment calls as to whether two characters are actually the same -- they may be "the same" in some contexts but not others -- while canonical equivalence is about different ways to produce the glyph for the same abstract character).

IDNA also assumed that the extensive Unicode stability rules would be applied and work as specified when new code points were added. Those

rules, as described in The Unicode Standard and the normative annexes identified below, provide that:

1. New code points representing precomposed characters that can be formed from combining sequences will not be added to Unicode unless neither the relevant base character nor required combining character(s) are part of the Standard within the relevant script [UAX15-Versioning].

2. If circumstances require that principle be violated, normalization stability requires that the newly-added character decompose (even under NFC) to the previously-available combining sequence [UAX15-Exclusion].

At least at the time IDNA2008 was being developed, there was no explicit provision in the Standard's discussion of conditions for adding new code points, nor of normalization stability, for an exception based on different languages using the same script or ambiguities about the shape or positioning of combining characters.

3.2. The discovery and the Arabic script cases

While the set of problems with normalization discussed above were discovered with a newly-added code point for the Arabic Script and some characteristics of Unicode handling of that script seem to make the problem more complex going forward, these are not issues specific to Arabic. This section describes the Arabic-specific problems; subsequent ones (starting with Section 3.3) discuss the problem more generally and include illustrations from other scripts.

3.2.1. New code point U+08A1, decomposition, and language dependency

Unicode 7.0.0 introduces the new code point U+08A1, ARABIC LETTER BEH WITH HAMZA ABOVE. As can be deduced from the name, it is visually identical to the glyph that can be formed from a combining sequence consisting of the code point for ARABIC LETTER BEH (U+0628) and the code point for Combining Hamza Above (U+0654). The two rules summarized above (see the last part of Section 3.1) suggest that either the new code point should not be allocated at all or that it should have a decomposition to \u'0628'\u'0654'.

Had the issues outlined in this document been better understood at the time, it probably would have been wise for RFC 5892 to disallow either the precomposed character or the combining sequence of each pair in those cases in which Unicode normalization rules do not cause the right thing to happen, i.e., the combining sequence and precomposed character to be treated as equivalent. Failure to do so at the time places an extra burden on registries to be sure that

conflicts (and the potential for confusion and attacks) do not exist. Oddly, had the exclusion been made part of the specification at that time, the preference for precomposed forms noted above would probably have dictated excluding the combining sequence, something not otherwise done in IDNA2008 because the NFC requirement serves the same purpose. Today, the only thing that can be excluded without the potential disruption of disallowing a previously-PVALID combining sequence is the to exclude the newly-added code point so whatever is done, or might have been contemplated with hindsight, will be somewhat inconsistent.

3.2.2. Other examples of the same behavior within the Arabic Script

One of the things that complicates the issue with the new U+08A1 code point is that there are several other Arabic-script code points that behave in the same way for similar language-specific reasons.

In particular, at least three other grapheme clusters that have been present for many version of Unicode can be seen as involving issues similar to those for the newly-added ARABIC LETTER BEH WITH HAMZA ABOVE. ARABIC LETTER HAH WITH HAMZA ABOVE (U+0681) and ARABIC LETTER REH WITH HAMZA ABOVE (U+076C) do not have decomposition forms and are preferred over combining sequences using HAMZA ABOVE (U+0654) [Unicode70-Hamza]. By contrast, ARABIC LETTER ALEF WITH HAMZA ABOVE (U+0623) decomposes into \u’0627’\u’0654’, ARABIC LETTER WAW WITH HAMZA ABOVE (U+0624) decomposes into \u’0648’\u’0654’, and ARABIC LETTER YEH WITH HAMZA ABOVE (U+0626) decomposes into \u’064A’\u’0654’ so the precomposed character and combining sequences compare equal when both are normalized, as this specification prefers.

There are other variations in which a precomposed character involving HAMZA ABOVE has a decomposition to a combining sequence that can form it. For example, ARABIC LETTER U WITH HAMZA ABOVE (U+0677) has a compatibility decomposition. but not a canonical one, into the combining sequence \u’06C7’\u’0674’.

3.2.3. Hamza and Combining Sequences

As the Unicode Standard points out at some length [Unicode70-Arabic], Hamza is a problematic abstract character and the "Hamza Above" construction even more so [Unicode70-Hamza]. Those sections explain a distinction made by Unicode between the use of a Hamza mark to denote a glottal stop and one used as a diacritic mark to denote a separate letter. In the first case, the combining sequence is used. In the second, a precomposed character is assigned.

Unlike Unicode generally and because of concerns about identifier spoofing and attacks based on similarities, character distinctions in

IDNA are based much more strictly on the appearance of characters; language and pronunciation distinctions within a script are not considered. So, for IDNA, BEH WITH HAMZA ABOVE is not-quite-tautologically the same as BEH WITH HAMZA ABOVE, even if one of them is written as U+08A1 (new to Unicode 7.0.0) and the other as the sequence \u'0628'\u'0654' (feasible with Unicode 7.0.0 but also available in versions of Unicode going back at least to the version [Unicode32] used in the original version of IDNA [RFC3490]. Because the precomposed form and combining sequence are, for IDNA purposes, the same, IDNA expects that normalization (specifically the requirement that all U-labels be in NFC form) will cause them to compare equal.

If Unicode also considered them the same, then the principle would apply that new precomposed ("composition") forms are not added unless one of the code points that could be used to construct it did not exist in an earlier version (and even then is discouraged) [UAX15-Versioning]. When exceptions are made, they are expected to conform to the rules and classes in the "Composition Exclusion Table", with class 2 being relevant to this case [UAX15-Exclusion]. That rule essentially requires that the normalization for the old combining sequence to itself be retained (for stability) but that the newly-added character be treated as canonically decomposable and decompose back to the older sequence even under NFC. That was not done for this particular case, presumably because of the distinction about pronunciation modifiers versus separate letters noted above. Because, for IDNA and the DNS, there is a possibility that the composing sequence \u'0628'\u'0654' already appears in labels, the only choice other than allowing an otherwise-identical, and identically-appearing, label with U+08A1 substituted to identify a different DNS entry is to DISALLOW the new character.

3.3. Precomposed characters without decompositions more generally

3.3.1. Description of the general problem

As mentioned above, IDNA made a strong assumption that, if there were two ways to form the same abstract character in the same script, normalization would result in them comparing equal. Work on IDNA2008 recognized that early version of Unicode might also contain some inconsistencies; see Section 3.3.2.3.2 below.

Having precomposed code points exist that don't have decompositions, or having code points of that nature allocated in the future, is problematic for those IDNA assumptions about character comparison. It seems to call for either excluding some set of code points that IDNA's rules do not now identify, development and use of a normalization procedure that behaves as expected (those two options

may be nearly equivalent for many purposes), or deciding to accept a risk that, apparently, will only increase over time.

It is not clear whether the reasons the IDNABIS WG did not understand and allow for these cases are important except insofar as they inform considerations about what to do in the future. It seemed (and still seems to some people) that the Unicode Standard is very clear on the matter (or at least was when IDNA2008 was being developed). In addition to the normalization stability rules cited in the last part of Section 3.1. the discussion in the Core Standard seems quite clear. For example, "Where characters are used in different ways in different languages, the relevant properties are normally defined outside the Unicode Standard" in Section 2.2, subsection titled "Semantics" [Unicode7] did not suggest to most readers that sometimes separate code points would be allocated within a script based on language considerations. Similarly, the same section of the Standard says, in a subsection titled "Unification", "The Unicode Standard avoids duplicate encoding of characters by unifying them within scripts across language" and does not list exceptions to that rule or limit it to a single script although it goes on to list "CJK" as an example. Another subsection, "Equivalent Sequences" indicates "Common precomposed forms ... are included for compatibility with current standards. For static precomposed forms, the standard provides a mapping to an equivalent dynamically composed sequence of characters". The latter appears to be precisely the "all precomposed characters decompose into the relevant combining sequences if the relevant base and combining characters exist in the Standard" rule that IDNA needs and assumed and, again, there is no mention of exceptions, language-dependent of otherwise. The summary of stability policies cited in the Standard [Unicode70-Stability] does not appear to shed any additional light on these issues.

The Standard now contains a subsection titled "Non-decomposition of Overlaid Diacritics" [Unicode70-Overlay] that identifies a list of diacritics that do not normally form characters that have decompositions. The rule given has its own exceptions and the text clearly states that there is actually no way to know whether a code point has a decomposition other than consulting the Unicode Character Database entry for that code point. The subsequent section notes that this can be a security problem. While the issues with IDNA go well beyond what is normally considered security, that comment now seems clear. While that subsection is helpful in explaining the problem, especially for European scripts, it does not appear in the Unicode versions that were current when IDNA2008 was being developed.

3.3.2. Latin Examples and Cases

While this set of problems was discovered because of a code point added to the Arabic script in precombined form to support a particular language, there are actually far more examples for, e.g., Latin script than there are for Arabic script. Many of them are associated with the "non-decomposition of combining diacriticals" issues mentioned above, but the next subsections describe other cases that are not directly bound to decomposition.

3.3.2.1. The font exclusion and compatability relationships

Unicode contains a large collection of characters that are identified as "Mathematical Symbols". A large subset of them are basic or decorated Latin characters, differing from the ordinary ones only by their usage and, in appearance, by font or type styling (despite the general principle that font distinctions are not used as the basis for assigning separate code points. Most of these have canonical mappings to the base form, which eliminates them from IDNA, but others do not and, because the same marks that are used as phonetic diacritical markings in conventional alphabetical use have special mathematical meanings, applications that permit the use of these characters have their own issues with normalization and equality.

3.3.2.2. The phonetic notation characters and extensions

Another example involves various Phonetic Alphabet and Extension characters. many of which, unlike the Mathematical ones, do not have normalizations that would make them compare equal to the basic characters with essentially identical representations. This would not be a problem for IDNA if they were identified with a specialized script or as symbols rather than letters, but neither is the case: they are generally identified as lower case Latin Script letters even when they are visually upper-case, another issue for IDNA.

3.3.2.3. The stroke (solidus) ambiguity

Some combining characters have two or more forms. for example, in the case of the character popularly known as "slash", "stroke", or "solidus" (sometime prefixed by "forward"), there are "short" and "long" combining forms, U+0337 (COMBINING SHORT SOLIDUS OVERLAY) and U+0338 (COMBINING LONG SOLIDUS OVERLAY). It is not clear how long a short one needs to be to make it "long" or how short a long one needs to be to make it "short". Perhaps for that reason, U+00F8 has no decomposition and neither U+006F U+0337 nor U+006F U+0338 combine to it with NFC.

Adding to the confusion, at least when one attempts to use Unicode character names to identify places to look for problems, U+00F8 is formally called LATIN SMALL LETTER O WITH STROKE but, in combining character terminology, the term "stroke" refers to a horizontal bar, not an angled one, as in U+0335 and U+0336 (also short and long versions). However, when one overlays one of those on an "o" (U+006F), one gets U+0275, LATIN SMALL LETTER BARRED O, not "...o with stroke". That character, by the way, does not decompose either. This does illustrate the principle that it is not feasible to rely on Unicode code point names to identify confusable character sequences, even ones that produce the same, more or less font-independent, grapheme clusters.

3.3.2.3.1. Combining dots and other shapes combine... unless...

The discussion of "Non-decomposition of Overlaid Diacritics" [Unicode70-Overlay] indirectly exhibits at least one reason why it has been difficult to characterize the problem. If one combines that subsection with others, one gets a set of rules that might be described as:

1. If the precomposed character and the code points that make up the combining sequence exist, then canonical composition and decomposition work as expected, except...

2. If the precomposed character was added to Unicode after the code points that make up the combining sequence, normalization stability for the combining sequences requires that NFC applied to the precomposed character decomposes rather than having the combining sequence compose to the new character, however...

3. If the combining sequence involves a diacritic or other mark that actually touches the base character when composed, the precomposed character does not have a decomposition, unless...

4. The combining diacritic involved is Cedilla (U+0327), Ogonek (U+0328), or Horn (U+031B), in which case the precomposed characters that contain them "regularly" (but presumably not always) decomposes, and...

5. There are further exceptions for Hamza which does not overlay the associated base character in the same way the Latin-derived combining diacritics and other marks do. Those decisions to decompose a precomposed character (or not) are based on language or phonetic considerations, not the combining mechanism or appearance, or perhaps,...

6. Some characters have compatibility decompositions rather than canonical ones [Unicode70-CompatDecomp]. Because compatibility relationships are treated differently by IDNA, PRECIS [RFC8264], and, potentially, other protocols involving identifiers for Internet use, the existence of compatibility relationship may or may not be helpful. Finally,...

7. There is no reason to believe the above list is complete. In particular, if whether a precomposed character decomposes or not is determined by language or phonetic distinctions or by a decision that all new characters for some scripts will be precomposed while new ones for others will be added (if needed) as combining sequences, one may need additional rules on a per-script and/or per-character basis.

The above list only covers the cases involving combining sequences. It does not cover cases such as those in Section 3.3.2.1 and Section 3.3.2.2 and there may be additional groups of cases not yet identified.

3.3.2.3.2. "Legacy" characters and new additions

The development of categories and rules for IDNA recognized that early version of Unicode might contain some inconsistencies if evaluated using more contemporary rules about code point assignments and stability. In particular, there might be some exceptions from different practices in early version of Unicode or anomalies caused by copying existing single- or dual-script standards into Unicode as block rather than individual character additions to the repertoire. The possibility of such "legacy" exceptions was one reason why the IDNA category rules include explicit provisions for exception lists (even though no such code points were identified prior to 2014).

3.3.3. Unexpected Combining Sequances

Most combining characters have the script property "Inherited" or "Common", i.e., are not members of any particular script and will not cause rules against mixed-script labels to be triggered. Normalization rules are generally structured around the base character, so unexpected combinations of base characters with combining ones may lead to cases where normalization might normally be expected to produce a precombined character but does not do so (in the most common situation because no such precombined character exists. For example, the Latin script characters "a" and "a with acute accent" are both coded (as U+0061 and U+00E1). If the latter is coded as the combining sequence U+0061 U+0301, NFC will turn that sequence into U+00E1 and everything will work as users expect. However, the Cyrillic "a" character (U+0430) is notoriously similar

in appearance in most type styles to U+0061 and the U+0439 U+0301 and that sequence does not normalize to anything else. Because thre is no code point assigned for Cyrillic small letter a with acute accent and unlike many of the other examples in this document, that is Unicode working exactly as would be expected. Whether it is an issue or not depends on the questions that are being asked and what rules are being applied.

3.3.4. Examples and Cases from Other Scripts

Research into these issues has not yet turned up a comprehensive list of affected scripts and code points. As discussed elsewhere in this document, it is clear that Arabic and Latin Scripts are significantly affected, that some Han and Kangxu radicals and ideographs are affected, and that other examples do exist -- it is just not known how many of those examples there are and what patterns, if any, characterize them.

3.3.4.1. Scripts with precomposed preferences and ones with combining preferences

While the authors have been unable to find an explanation for the differentiation in the Unicode Standard, we have been told that there are differences among scripts as to whether the action preference is to add new combining sequences only (and resist adding precomposed characters) as suggested in Section 3.3.2.3.1 or to add precomposed characters, often ones that do not have decompositions. If those difference in preference do exist, it is probably important to have them documented so that they can be reflected in IDNA review procedures and elsewhere. It will also require IETF discussion of whether combining sequences should be deprecated when the corresponding precomposed characters are added or to disallow combining sequences entirely for those scripts (as has been implicitly suggested for Arabic language use [RFC5564]).

[[CREF2: The above isn’t quite right and probably needs additional discussion and text.]]

3.3.4.2. The Han and Kangxu Cases

[[CREF3: .. to be supplied ..]]

3.4. Confusion and the Casual User

To the extent to which predictability for relatively casual users is a desired and important feather of relevant application or application support protocols, it is probably worth observing that the complex of rules and cases suggested or implied above is almost

certainly too involved for the typical such user to develop a good intuitive understanding of how things behave and what relationships exist. Conversely, the nature of writing systems for natural languages, especially those that have evolved and diverged over centuries, implies that no set of rules about allowable characters will guarantee complete safety (however that is defined).

4. Implementation options and issues: Unicode properties, exceptions, and the nature of stability

4.1. Unicode Stability compared to IETF (and ICANN) Stability

The various stability rules in Unicode [Unicode70-Stability] all appear to be based on the model that once a value is assigned, it can never be changed. That is probably appropriate for a character coding system with multiple uses and applications. It is probably the only option when normative relationships are expressed in tables of values rather than by rules. One consequence of such a model is that it is difficult or impossible to fix mistakes (for some stability rules, the Unicode Standard does provide for exceptions) and even harder to make adjustments that would normally be dictated by evolution.

"No changes" provides a very strong and predictable type of stability. There are many reasons to take that path. As in some of the cases that motivated this document, the difficulty is that simply adding new code points (in Unicode) or features (in a protocol or application) may be destabilizing. One then has complete stability for systems that never use or allow the new code points or features, but rough edges for newer systems that see the discrepancies and rough edges. IDNA2003 (inadvertently) took that approach by freezing on Unicode 3.2 -- if no code points added after Unicode 3.2 had ever been allowed, we would have had complete stability even as Unicode libraries changed. Unicode has been quite ingenious about working around those difficulties with such provisions as having code points for newly-added precomposed characters decompose rather than altering the normalization for the combining sequences. Other cases, such as newly-added precomposed characters that do not decompose for, e.g., language or phonetic reasons, are more problematic.

The IETF (and ICANN and standards development bodies such as ISO and ISO/IEC JTC1) have generally adopted a different type of stability model, one which considers experience in use and the ill effects of not making changes as well as the disruptive effects of doing so. In the IETF model, if an earlier decision is causing sufficient harm and there is consensus in the communities that are most affected that a change is desirable enough to make transition costs acceptable, then the change is made.

The difference and its implications are perhaps best illustrated by a disagreement when IDNA2008 was being approved. IDNA2003 had effectively prevented some characters, notably (measured by intensity of the protests) the Sharp S character (U+00DF) from being used in DNS labels by mapping them to other characters before conversion to ACE form. It has also prohibited some other code points, notably ZWJ (U+200D) and ZWNJ (U+200C), by discarding them. In both cases, there were strong voices from the relevant language communities, supported by the registry communities, that the characters were important enough that it was more desirable to undergo the short-term pain of a transition and some uncertainty than to continue to exclude those characters and the IDNA2008 rules and repertoire are consistent with that preference. The Unicode Consortium apparently believed that stability --elimination of any possibility of label invalidation or different interpretations of the same string-- was more important than those writing system requirements and community preferences. That view was expressed through what was effectively a fork in (or attempt to nullify) the IETF Standard [UTS46] a result that has probably been worse for the overall Internet than either of the possible decision choices.

4.2. New Unicode Properties

One suggestion about the way out of these problems would be to create one or more new Unicode properties, maintained along with the rest of Unicode, and then incorporated into new or modified rules or categories in IDNA. Given the analysis in this document, it appears that that property (or properties) would need to provide:

1. Identification of combining characters that, when used in combining sequences, do not produce decomposable characters. [[CREF4: Wording on the above is not quite right but, for the present, maybe the intent is clear.]]

2. Identification of precomposed characters that might reasonably be expected to decompose, but that do not.

3. Identification of character forms that are distinct only because of language or phonetic distinctions within a script.

4. Identification of scripts for which precomposed forms are strongly preferred and combining sequences should either be viewed as temporary mechanisms until precomposed characters are assigned or banned entirely.

5. Identification of code points that represent symbols for specific, non-language, purposes even if identified as letters or numerals by their General Property. This would include all

characters given separate code points because of specialized "mathematical" and "phonetic" characters (see Section 3.3.2.2 and Section 3.3.2.1), but there are probably additional cases.

Some of these properties (or characteristics or values of a single property) would be suitable for disallowing characters, code points, or contextual sequences that otherwise might be allowed by IDNA. Others would be more suitable for making equality comparisons come out as needed by IDNA, particularly to eliminate distinctions based on language context.

While it would appear that appropriate rules and categories could be developed for IDNA (and, presumably, for PRECIS, etc.) if the problem areas are those identified in this document, it is not yet known whether the list is complete (and, hence, whether additional properties or information would be needed).

Even with such properties, IDNA would still almost certainly need exception lists. In addition, it is likely that stability rules for those properties would need to reflect IETF norms with arrangements for bringing the IETF and other communities into the discussion when tradeoffs are reviewed.

4.3. The need for exception lists

[[CREF5: Note in draft: this section is a partial placeholder and may need more elaboration.]]
Issues with exception lists and the requirements for them are discussed in Section 2 above and in RFC 5894 [RFC5894].

5. Proposed/ Alternative Changes to RFC 5892 for the issues first exposed by new code point U+08A1

NOTE IN DRAFT: See the comments in the Introduction, Section 1 and the first paragraph of each Subsection below for the status of the Subsections that follow. Each one, in combination with the material in Section 3 above, also provides information about the reasons why that particular strategy might or might not be appropriate.

When the term "Category" followed by an upper-case letter appears below, it is s reference to a rule in RFC 5892.

5.1. Disallow This New Code Point

This option is almost certainly too Arabic-specific and does not solve, or even address, the underlying problem. It also does not inherently generalize to non-decomposing precomposed code points that might be added in the future (whether to Arabic or other scripts)

even though one could add more code points to Category F in the same way.

If chosen by the community, this subsection would update the portion of the IDNA2008 specification that identifies rules for what characters are permitted [RFC5892] to disallow that code point.

With the publication of this document, Section 2.6 ("Exceptions (F)") of RFC 5892 [RFC5892] is updated by adding 08A1 to the rule in Category F so that the rule itself reads:

```
F: cp is in {00B7, 00DF, 0375, 03C2, 05F3, 05F4, 0640, 0660,
             0661, 0662, 0663, 0664, 0665, 0666, 0667, 0668,
             0669, 06F0, 06F1, 06F2, 06F3, 06F4, 06F5, 06F6,
             06F7, 06F8, 06F9, 06FD, 06FE, 07FA, 08A1, 0F0B,
             3007, 302E, 302F, 3031, 3032, 3033, 3034, 3035,
             303B, 30FB}
```

and then add to the subtable designated
"DISALLOWED -- Would otherwise have been PVALID"
after the line that begins "07FA", the additional line:

```
08A1; DISALLOWED # ARABIC LETTER BEH WITH HAMZA ABOVE
```

This has the effect of making the cited code point DISALLOWED independent of application of the rest of the IDNA rule set to the current version of Unicode. Those wishing to create domain name labels containing Beh with Hamza Above may continue to use the sequence

U+0628, ARABIC LETTER BEH
followed by

U+0654, ARABIC HAMZA ABOVE

which was valid for IDNA purposes in Unicode 5.0 and earlier and which continues to be valid.

In principle, much the same thing could be accomplished by using the IDNA "BackwardCompatible" category (IDNA Category G, RFC 5892 Section 5.3). However, that category is described as applying only when "property values in versions of Unicode after 5.2 have changed in such a way that the derived property value would no longer be PVALID or DISALLOWED". Because U+08A1 is a newly-added code point in Unicode 7.0.0 and no property values of code points in prior versions have changed, category G does not apply. If that section of RFC 5892 were to be replaced in the future, perhaps consideration should be

given to adding Normalization Stability and other issues to that description but, at present, it is not relevant.

5.2. Disallow This New Code Point and All Future Precomposed Additions that Do Not Decompose

At least in principle, the approach suggested above (Section 5.1) could be expanded to disallow all future allocations of non-decomposing precomposed characters. This would probably require either a new Unicode property to identify such characters and/or more emphasis on the manual, individual code point, checking of the new Unicode version review proces (i.e,. not just application of the existing rules and algorithm). It might require either a new rule in IDNA or a modification to the structure of Category F to make additions less tedious. It would do nothing for different ways to form identical characters within the same script that were not associated with decomposition and so would have to be used in conjunction with other appropaches. Finally, for scripts (such as Arabic) where there is a very strong preference to avoid combining sequences, this approach would exclude exactly the wrong set of characters.

5.3. Disallow the combining sequences for these characters

As in the approach discussed in Section 5.1, this approach is too Arabic-specific to address the more general problem. However, it illustrates a single-script approach and a possible mechanism for excluding combining sequences whose handling is connected to language information (information that, as discussed above, is not relevant to the DNS).

If chosen by the community, this subsection would update the portion of the IDNA2008 specification that identifies contextual rules [RFC5892] to prohibit (combining) Hamza Above (U+0654) in conjunction with Arabic BEH (U+0628), HAH (U+062D), and REH (U+0631). Note that the choice of this option is consistent with the general preference for precomposed characters discussed above but would ban some labels that are valid today and that might, in principle, be in use.

The required prohibition could be imposed by creating a new contextual rule in RFC 5892 to constrain combining sequences containing Hamza Above.

As the Unicode Standard points out at some length [Unicode70-Arabic], Hamza is a problematic abstract character and the "Hamza Above" construction even more so. IDNA has historically associated characters whose use is reasonable in some contexts but not others with the special derived property "CONTEXTO" and then specified

specific, context-dependent, rules about where they may be used. Because Hamza Above is problematic (and spawns edge cases, as discussed in the Unicode Standard section cited above), it was suggested that a contextual rule might be appropriate. There are at least two reasons why a contextual rule would not be suitable for the present situation.

1. As discussed above, the present situation is a normalization stability and predictability problem, not a contextual one. Had the same issues arisen with a newly-added precomposed character that could previously be constructed from non-problematic base and combining characters, it would be even more clearly a normalization issue and, following the principles discussed there and particularly in UAX 15 [UAX15-Exclusion], might not have been assigned at all.

2. The contextual rule sets are designed around restricting the use of code points to a particular script or adjacent to particular characters within that script. Neither of these cases applies to the newly-added character even if one could imagine rules for the use of Hamza Above (U+0654) that would reflect the considerations of Chapter 8 of Unicode 6.2. Even had the latter been desired, it would be somewhat late now -- Hamza Above has been present as a combining character (U+0654) in many versions of Unicode. While that section of the Unicode Standard describes the issues, it does not provide actionable guidance about what to do about it for cases going forward or when visual identity is important.

5.4. Use Combinnig Classes to Develop Additional Contextual Rules

This option may not be of any practical use, but Unicode supports a property called "Combining_Class". That property has been used in IDNA only to construct a contextual rule for Zero-Width Non-Joiner [RFC5892, Appendix A.1] but speculation has arisen during discussions of work on Arabic combining characters and rendering [UTR53] as to whether Combining Classes could be used to build additional contextual rules that would restrict problematic cases. Unless such rules were applied only to new code points, they would also not be backward compatable.

The question of whether Combining Classes could be used to reduce the number of problematic labels is at least worth examination.

5.5. Disallow all Combining Characters for Specific Scripts

[[CREF6: This subsection needs to be turned into prose, but the follow bullet points are probably sufficient to identify the issues.]]

- Might work for Arabic and other "precomposed preference" scripts if those can be identified in an orderly and stable way (see Section 3.3.4.1; recommended by the Arabic language community for IDNs [RFC5564]).

- Unworkable for Latin because many characters that do not decompose are, at least in part, historical accidents resulting from combining prior national standards (this probably may exist for other scripts as well).

- No effect at all on special-use representations of identical characters within a script (see Section 3.3.2.1 and Section 3.3.2.2).

- Not backwards compatible.

5.6. Do Nothing Other Than Warn

A recommendation from UTC and others has been to simply warn registries, at all levels of the tree, to be careful with this set of characters. Doing that well would probably require making language distinctions within zones, which would violate the important IDNA principles that labels are not necessarily "words", do not carry language information, and may, at the protocol level, even deliberately mix languages and scripts. It is also problematic because the relevant set of characters is not easily defined in a precise way. This suggestion is problematic because the DNS and IDNA cannot make or enforce language distinctions, but it would avoid having the IETF either invalidate label strings that are potentially now in use or creating inconsistencies among the characters that combine with selected base characters but that also have precomposed forms that do not have decompositions. The potential would still exist for registries to respect the warning and deprecate such labels if they existed.

More generally, while there are already requirements in IDNA for registries to be knowledgeable and responsible about the labels they register (a separate document discusses that requirement [Klensin-rfc5891bis]), experience indicates that those requirements are often ignored. At least as important, warning registries about what should or should not be registered and even calling out specific code points as dangerous and in need of extra attention [Freytag-dangerous] does nothing to address the many cases in which lookup-time checking for IDNA conformance and deliberately misleading label constructions is important.

5.7. Normalization Form IETF (NFI))

The most radical possibility for the comparison issue would be to decide that none of the Unicode Normalization Forms specified in UAX 15 [UAX15] are adequate for use with the DNS because, contrary to their apparent descriptions, normalization tables are actually determined using language information. However, use of language information is unacceptable for IDNA for reasons described elsewhere in this document. The remedy would be to define an IETF-specific (or DNS-specific) normalization form (sometimes called "NFI" in discussions), building on NFC but adhering strictly to the rule that normalization causes two different forms of the same character (glyph image) within the same script to be treated as equal. In practice such a form could be implemented for IDNA purposes as an additional rule within RFC 5892 (and its successors) that constituted an exception list for the NFC tables. For this set of characters, the special IETF normalization form would be equivalent to the exclusion discussed in Section 5.3 above.

An Internet-identifier-specific normalization form, especially if specified somewhat separately from the IDNA core, would have a small marginal advantage over the other strategies in this section (or in combination with some of them), even though most of the end result and much of the implementation would be the same in practice. While the design of IDNA requires that strings be normalized as part of the process of determining label validity (and hence before either storage of values in the DNS or name resolution), there is an ongoing debate about whether normalization should be performed before storing a string or putting it on the wire or only when the string is actually compared or otherwise used.

If a normalization procedure with the right properties for the IETF was defined, that argument could be bypassed and the best decisions made for different circumstances. The separation would also allow better comparison of strings that lack language context in applications environments in which the additional processing and character classifications of IDNA and/or PRECIS were not applicable. Having such a normalization procedure defined outside IDNA would also minimize changes to IDNA itself, which is probably an advantage.

If the new normalizstion form were, in practice, simply an overlay on NFC with modifications dictated by exception and/or property lists, keeping its definition separate from IDNA would also avoid interweaving those exceptions and property lists with the rules and categories of IDNA itself, avoiding some unnecessary complexity.

6. Editorial clarification to RFC 5892

Verified RFC Editor Erratum 3312 [RFC5892Erratum] provides a clarification to Appendix A and Section A.1 of RFC 5892. This section of this document updates the RFC to apply that clarification.

1. In Appendix A, add a new paragraph after the paragraph that begins "The code point...". The new paragraph should read:

   "For the rule to be evaluated to True for the label, it MUST be evaluated separately for every occurrence of the Code point in the label; each of those evaluations must result in True."

2. In Appendix A, Section A.1, replace the "Rule Set" by

```
Rule Set:
  False;
  If Canonical_Combining_Class(Before(cp)) .eq.  Virama Then True;
  If cp .eq. \u200C And
         RegExpMatch((Joining_Type:{L,D})(Joining_Type:T)*cp
    (Joining_Type:T)*(Joining_Type:{R,D})) Then True;
```

7. Acknowledgements

The Unicode 7.0.0 changes were extensively discussed within the IAB's Internationalization Program. The authors are grateful for the discussions and feedback there, especially from Andrew Sullivan and David Thaler. Additional information was requested and received from Mark Davis and Ken Whistler and while they probably do not agree with the necessity of excluding this code point or taking even more drastic action as their responsibility is to look at the Unicode Consortium requirements for stability, the decision would not have been possible without their input. Thanks to Bill McQuillan and Ted Hardie for reading versions of the document carefully enough to identify and report some confusing typographical errors. Several experts and reviewers who prefer to remain anonymous also provided helpful input and comments on preliminary versions of this document.

8. IANA Considerations

When the IANA registry and tables are updated to reflect Unicode 7.0.0, changes should be made according to the decisions the IETF makes about Section 5.

9. Security Considerations

From at least one point of view, this document is entirely a discussion of a security issue or set of such issues. While the "similar-looking characters" issue that has been a concern since the earliest days of IDNs [HomographAttack] and that has driven assorted "character confusion" projects [ICANN-VIP], if a user types in a string on one device and can get different results that do not compare equal when it is typed on a different device (with both behaving correctly and both keyboards appearing to be the same and for the same script) then all security mechanism that depend on the underlying identifiers, including the practical applications of DNS response integrity checks via DNSSEC [RFC4033] and DNS-embedded public key mechanisms [RFC6698], are at risk if different parties, at least one of them malicious, obtain or register some of the identical-appearing and identically-typed strings and get them into appropriate zones.

Mechanisms that depend on trusting registration systems (e.g., registries and registrars in the DNS IDN case, see Section 5.6 above) are likely to be of only limited utility because fully-qualified domains that may be perfectly reasonable at the first level or two of the DNS may have differences of this type deep in the tree, into levels where name management, and often accountability, are weak. Similar issues obviously apply when names are user-selected or unmanaged.

When the issue is not a deliberate attack but simple accidental confusion among similar strings, most of our strategies depend on the acceptability of false negatives on matching if there is low risk of false positives (see, for example, the discussion of false negatives in identifier comparison in Section 2.1 of RFC 6943 [RFC6943]). Aspects of that issue appear in, for example, RFC 3986 [RFC3986] and the PRECIS effort [RFC8264]. However, because the cases covered here are connected, not just to what the user sees but to what is typed and where, there is an increased risk of false positives (accidental as well as deliberate).

[[CREF7: Note in Draft: The paragraph that follows was written for a much earlier version of this document. It is obsolete, but is being retained as a placeholder for future developments.]]

This specification excludes a code point for which the Unicode-specified normalization behavior could result in two ways to form a visually-identical character within the same script not comparing equal. That behavior could create a dream case for someone intending to confuse the user by use of a domain name that looked identical to

another one, was entirely in the same script, but was still considered different.

Internet Security in areas that involve internationalized identifiers that might contain the relevant characters is therefore significantly dependent on some effective resolution for the issues identified in this document, not just hand waving, devout wishes, or appointment of study committees about it.

10. References

10.1. Normative References

[RFC5137] Klensin, J., "ASCII Escaping of Unicode Characters", BCP 137, RFC 5137, DOI 10.17487/RFC5137, February 2008, <https://www.rfc-editor.org/info/rfc5137>.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <https://www.rfc-editor.org/info/rfc5890>.

[RFC5892] Faltstrom, P., Ed., "The Unicode Code Points and Internationalized Domain Names for Applications (IDNA)", RFC 5892, DOI 10.17487/RFC5892, August 2010, <https://www.rfc-editor.org/info/rfc5892>.

[RFC5892Erratum] "RFC5892, "The Unicode Code Points and Internationalized Domain Names for Applications (IDNA)", August 2010, Errata ID: 3312", Errata ID 3312, August 2012, <http://www.rfc-editor.org/errata_search.php?rfc=5892>.

[RFC5894] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Background, Explanation, and Rationale", RFC 5894, DOI 10.17487/RFC5894, August 2010, <https://www.rfc-editor.org/info/rfc5894>.

[RFC6943] Thaler, D., Ed., "Issues in Identifier Comparison for Security Purposes", RFC 6943, DOI 10.17487/RFC6943, May 2013, <https://www.rfc-editor.org/info/rfc6943>.

[RFC8264] Saint-Andre, P. and M. Blanchet, "PRECIS Framework: Preparation, Enforcement, and Comparison of Internationalized Strings in Application Protocols", RFC 8264, DOI 10.17487/RFC8264, October 2017, <https://www.rfc-editor.org/info/rfc8264>.

[UAX15] Davis, M., Ed., "Unicode Standard Annex #15: Unicode Normalization Forms", June 2014, <http://www.unicode.org/reports/tr15/>.

[UAX15-Exclusion]
"Unicode Standard Annex #15: ob. cit., Section 5", <http://www.unicode.org/reports/tr15/#Primary_Exclusion_List_Table>.

[UAX15-Versioning]
"Unicode Standard Annex #15, ob. cit., Section 3", <http://www.unicode.org/reports/tr15/#Versioning>.

[Unicode5]
The Unicode Consortium, "The Unicode Standard, Version 5.0", ISBN 0-321-48091-0, 2007.

Boston, MA, USA: Addison-Wesley. ISBN 0-321-48091-0. This printed reference has now been updated online to reflect additional code points. For code points, the reference at the time RFC 5890-5894 were published is to Unicode 5.2.

[Unicode62]
The Unicode Consortium, "The Unicode Standard, Version 6.2.0", ISBN 978-1-936213-07-8, 2012, <http://www.unicode.org/versions/Unicode6.2.0/>.

Preferred citation: The Unicode Consortium. The Unicode Standard, Version 6.2.0, (Mountain View, CA: The Unicode Consortium, 2012. ISBN 978-1-936213-07-8)

[Unicode7]
The Unicode Consortium, "The Unicode Standard, Version 7.0.0", ISBN 978-1-936213-09-2, 2014, <http://www.unicode.org/versions/Unicode7.0.0/>.

Preferred Citation: The Unicode Consortium. The Unicode Standard, Version 7.0.0, (Mountain View, CA: The Unicode Consortium, 2014. ISBN 978-1-936213-09-2)

[Unicode70-Arabic]
"The Unicode Standard, Version 7.0.0, ob.cit., Chapter 9.2: Arabic", Chapter 9, 2014, <http://www.unicode.org/versions/Unicode7.0.0/ch09.pdf>.

Subsection titled "Encoding Principles", paragraph numbered 4, starting on page 362.

[Unicode70-CompatDecomp]
"The Unicode Standard, Version 7.0.0, ob.cit., Chapter 2.3: Compatibility Characters", Chapter 2, 2014, <http://www.unicode.org/versions/Unicode7.0.0/ch02.pdf>.

Subsection titled "Compatibility Decomposable Characters" starting on page 26.

[Unicode70-Design]
"The Unicode Standard, Version 7.0.0, ob.cit., Chapter 2.2: Unicode Design Principles", Chapter 2, 2014, <http://www.unicode.org/versions/Unicode7.0.0/ch02.pdf>.

[Unicode70-Hamza]
"The Unicode Standard, Version 7.0.0, ob.cit., Chapter 9.2: Arabic", Chapter 9, 2014, <http://www.unicode.org/versions/Unicode7.0.0/ch09.pdf>.

Subsection titled "Combining Hamza Above" starting on page 378.

[Unicode70-Overlay]
"The Unicode Standard, Version 7.0.0, ob.cit., Chapter 2.2: Unicode Design Principles", Chapter 2, 2014, <http://www.unicode.org/versions/Unicode7.0.0/ch02.pdf>.

Subsection titled "Non-decomposition of Overlaid Diacritics" starting on page 64.

[Unicode70-Stability]
"The Unicode Standard, Version 7.0.0, ob.cit., Chapter 2.2: Unicode Design Principles", Chapter 2, 2014, <http://www.unicode.org/versions/Unicode7.0.0/ch02.pdf>.

Subsection titled "Stability" starting on page 23 and containing a link to http://www.unicode.org/policies/stability_policy.html..

[UTS46] Davis, M. and M. Suignard, "Unicode Technical Standard #46: Unicode IDNA Compatibility Processing", Version 7.0.0, June 2014, <http://unicode.org/reports/tr46/>.

10.2. Informative References

[Dalby] Dalby, A., "Dictionary of Languages: The definitive reference to more than 400 languages", Columbia Univeristy Press , 2004.

pages 206-207

[Daniels] Daniels, P. and W. Bright, "The World's Writing Systems", Oxford University Press , 1986.

page 744

[Freytag-dangerous]
Freytag, A., Klensin, J., and A. Sullivan, "Those Troublesome Characters: A Registry of Unicode Code Points Needing Special Consideration When Used in Network Identifiers", June 2017, <https://datatracker.ietf.org/doc/draft-freytag-troublesome-characters/>.

[HomographAttack]
Gabrilovich, E. and A. Gontmakher, "The Homograph Attack", Communications of the ACM 45(2):128, February 2002, <http://www.cs.technion.ac.il/~gabr/papers/homograph_full.pdf>.

[ICANN-VIP]
ICANN, "The IDN Variant Issues Project: A Study of Issues Related to the Management of IDN Variant TLDs (Integrated Issues Report)", February 2012, <https://www.icann.org/en/system/files/files/idn-vip-integrated-issues-final-clean-20feb12-en.pdf>.

[Klensin-rfc5891bis]
Klensin, J., "Internationalized Domain Names in Applications (IDNA): Registry Restrictions and Recommendations", September 2017, <https://datatracker.ietf.org/doc/draft-klensin-idna-rfc5891bis/>.

[Omniglot-Fula]
Ager, S., "Omniglot: Fula (Fulfulde, Pulaar, Pular'Fulaare)", <http://www.omniglot.com/writing/fula.htm>.

Captured 2015-01-07

[RFC0020] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <https://www.rfc-editor.org/info/rfc20>.

[RFC3490] Faltstrom, P., Hoffman, P., and A. Costello, "Internationalizing Domain Names in Applications (IDNA)", RFC 3490, DOI 10.17487/RFC3490, March 2003, <https://www.rfc-editor.org/info/rfc3490>.

[RFC3492] Costello, A., "Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)", RFC 3492, DOI 10.17487/RFC3492, March 2003, <https://www.rfc-editor.org/info/rfc3492>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC4033] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "DNS Security Introduction and Requirements", RFC 4033, DOI 10.17487/RFC4033, March 2005, <https://www.rfc-editor.org/info/rfc4033>.

[RFC5564] El-Sherbiny, A., Farah, M., Oueichek, I., and A. Al-Zoman, "Linguistic Guidelines for the Use of the Arabic Language in Internet Domains", RFC 5564, DOI 10.17487/RFC5564, February 2010, <https://www.rfc-editor.org/info/rfc5564>.

[RFC6452] Faltstrom, P., Ed. and P. Hoffman, Ed., "The Unicode Code Points and Internationalized Domain Names for Applications (IDNA) - Unicode 6.0", RFC 6452, DOI 10.17487/RFC6452, November 2011, <https://www.rfc-editor.org/info/rfc6452>.

[RFC6698] Hoffman, P. and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA", RFC 6698, DOI 10.17487/RFC6698, August 2012, <https://www.rfc-editor.org/info/rfc6698>.

[Unicode32]
The Unicode Consortium, "The Unicode Standard, Version 3.2.0".

The Unicode Standard, Version 3.2.0 is defined by The Unicode Standard, Version 3.0 (Reading, MA, Addison-Wesley, 2000. ISBN 0-201-61633-5), as amended by the Unicode Standard Annex #27: Unicode 3.1 (http://www.unicode.org/reports/tr27/) and by the Unicode Standard Annex #28: Unicode 3.2 (http://www.unicode.org/reports/tr28/).

[UTR53] Unicode Consortium, "Proposed Draft: Unicode Technical Report #53: Unicode Arabic Mark Ordering Algorithm", August 2017, <http://www.unicode.org/reports/tr53/>.

Note: this is a Proposed Draft, out for public review when this version of the current I-D is posted, and should not be considered either an approved/ final document or a stable reference.

Appendix A. Change Log

RFC Editor: Please remove this appendix before publication.

A.1. Changes from version -00 (2014-07-21)to -01

- Version 01 of this document is an extensive rewrite and reorganization, reflecting discussions with UTC members and adding three more options for discussion to the original proposal to simply disallow the new code point.

A.2. Changes from version -01 (2014-12-07) to -02

Corrected a typographical error in which Hamza Above was incorrectly listed with the wrong code point.

A.3. Changes from version -02 (2014-12-07) to -03

Corrected a typographical error in the Abstract in which RFC 5892 was incorrectly shown as 5982.

A.4. Changes from version -03 (2015-01-06) to -04

- Explicitly identified the applicability of U+08A1 with Fula and added references that discuss that language and how it is written.

- Updated several Unicode 6.2 references to point to Unicode 7.0 since the latter is now available in stable form (it was done when work on this I-D started).

- Extensively revised to discuss the non-Arabic cases, non-decomposing diacritics, other types of characters that don’t compare equal after normalization, and more general problem and approaches.

A.5. Changes from version -04 (2015-03-11) to -05

- Modified a few citation labels to make them more obvious.
- Restructured Section 1 and added additional terminology comments.
- Added discussion about non-decomposable character cases, including the "slash" example, and associated references for which -04 contained only placeholders.
- The examples and discussion of Latin script issues has been expanded considerably. It is unfortunate that many readers in the IETF community apparently cannot understand examples well enough to believe a problem is significant unless they is a discussion of Latin script examples, but, at least for this working draft, that is the way it is.
- Rewrote the discussion of several of the alternatives and added the discussion of combining classes.
- Rewrote and extended the discussion of the "warn only" alternative.
- Several other sections modified to improve technical or editorial clarity.
- Note that, while some references have been updated, others have not. In particular, Unicode references are still tied to versions 6 or 7. In some cases, those non-historical references are and will remain appropriate; others will best be replaced with information about current versions of documents.

Authors' Addresses

John C Klensin
1770 Massachusetts Ave, Ste 322
Cambridge, MA 02140
USA

Phone: +1 617 245 1457
Email: john-ietf@jck.com

Patrik Faltstrom
Netnod
Franzengatan 5
Stockholm 112 51
Sweden

Phone: +46 70 6059051
Email: paf@netnod.se

Network Working Group
Internet-Draft
Updates: 5890, 5891, 5894 (if approved)
Intended status: Standards Track
Expires: March 16, 2018

J. Klensin

A. Freytag
ASMUS, Inc.

September 12, 2017

Internationalized Domain Names in Applications (IDNA): Registry Restrictions and Recommendations
draft-klensin-idna-rfc5891bis-01

Abstract

The IDNA specifications for internationalized domain names combine rules that determine the labels that are allowed in the DNS without violating the protocol itself and an assignment of responsibility, consistent with earlier specifications, for determining the labels that are allowed in particular zones. Conformance to IDNA by registries and other implementations requires both parts. Experience strongly suggests that the language describing those responsibilities was insufficiently clear to promote safe and interoperable use of the specifications and that more details and some specific examples would have been helpful. Without making any substantive changes to IDNA, this specification updates two of the core IDNA documents (RFC 5980 and 5891) and the IDNA explanatory document (RFC 5894) to provide that guidance and to correct some technical errors in the descriptions.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on March 16, 2018.

Table of Contents

1. Introduction

Parts of the specifications for Internationalized Domain Names in Applications (IDNA) [RFC5890] [RFC5891] [RFC5894] (collectively known, along with RFC 5892 [RFC5892], RFC 5893 [RFC5893] and updates to them, as "IDNA2008" (or just "IDNA") impose a requirement that domain name system (DNS) registries restrict the characters they allow in domain name labels (see Section 2 below), and the contents and structure of those labels. That requirement and restriction are consistent with the "trustee for the community" requirements of the original specification for DNS naming and authority [RFC1591]. The restrictions are intended to limit the permitted characters and strings to those for which the registries or their advisers have a

thorough understanding and for which they are willing to take responsibility.

That provision is centrally important because it recognized that historical relationships and variations among scripts and writing systems, the continuing evolution of those systems, differences in the uses of characters among languages (and locations) that use the same script, and so on make it impossible for a single list of characters and simple rules to be able to generate an "if we use these, we will be safe from confusion and various attacks" guideline.

Instead, the algorithm and rules of RFC 5981 and 5982 eliminate many of the most dangerous and otherwise problematic cases, but cannot eliminate the need for registries and registrars to understand what they are doing and taking responsibility for the decisions they make.

The way in which the IDNA2008 specifications expressed these requirements may have obscured the intention that they actually are requirements. Section 2.3.2.3 of the Definitions document [RFC5890] mentions the need for the restrictions, indicates that they are mandatory, and points the reader to section 4.3 of the Protocol document [RFC5891], which in turn points to Section 3.2 of the Rationale document [RFC5894], with each document providing further detail, discussion, and clarification.

This specification is intended to unify and clarify these requirements for registry decisions and responsibility and to emphasize the importance of registry restrictions at all levels of the DNS. It also makes a specific recommendation for character repertoire subsetting intermediate between the code points allowed by RFC 5891 and 5892 and those allowed by individual registries. It does not alter the basic IDNA2008 protocols and rules themselves in any way.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Registry Restrictions in IDNA2008

As mentioned above, IDNA2008 specifies that the registries for each zone in the DNS that supports IDN labels are required to develop and apply their own rules to restrict the allowable labels, including limiting characters they allow to be used in labels in that zone. The chosen list MUST BE smaller than the collection of code points specified as "PVALID", "CONTEXTJ", and "CONTEXTO" by the rules established by the protocols themselves. The latter two categories, and labels containing any characters that are normally part of a

script written right to left [RFC5893], require that additional rules, specified in the protocols and known as "contextual rules" and "bidi rules", be applied. The entire collection of rules and restrictions required by the IDNA2008 protocols themselves are known as "protocol restrictions".

As mentioned above, registries may apply (and generally are required to apply) additional rules to further restrict the list of permitted code points, contextual rules (perhaps applied to normally PVALID code points) that apply additional restrictions, and/or restrictions on labels. The most obvious of those restrictions include provisions for restricting suggested new registrations based on conflicts with labels already registered in the zone and specifications of what constitutes such conflicts based on the properties of the labels in question. They further include prohibitions on code points and labels that are not consistent with the intended function of the zone or the subtree in which it is embedded (see Section 3) or limitations on where in a label allowable code points may be placed.

These per-registry (or per-zone) rules are commonly known as "registry restrictions" to distinguish them from the protocol restrictions described above. By necessity, the latter are somewhat generic, having to cater both to the union of the needs for all zones, as well as to the most permissive zones. In consequence, additional Registry restrictions are essential to provide for the necessary security in the face of the tremendous variations and differences in writing systems, their ongoing evolution and development, as well as the human ability to recognize and distinguish characters in different scripts around the world and under different circumstances.

3. Progressive Subsets of Allowed Characters

The algorithm and rules of RFC 5891 and 5892 set an absolute upper bound on the code points that can be used in domain name labels; registries MUST NOT include code points unless they are allowed by those rules. Each registry that intends to allow IDN registrations MUST then determine which code points will be allowed by that registry. It SHOULD also consider additional rules, including contextual and whole label restrictions that provide further protection for registrants and users. For example, the widely-used principle that bars labels containing characters from more than one script is not an IDNA2008 requirement. It has been adopted by many registries but, as Section 4.4 of RFC 5890 indicates, there may be circumstances in which is it not required or appropriate.

In formulating their own rules, registries SHOULD normally consult carefully-developed consensus recommendations about global maximum

repertoires to be used such as the ICANN Maximal Starting Repertoire 2 (MSR-2) for the Development of Label Generation Rules for the Root Zone [ICANN-MSR2] (or its successor documents). Additional recommendations of similar quality about particular scripts or languages exist, including, but not limited to, the RFCs for Cyrillic [RFC5992] or Arabic Language [RFC5564] or script-based repertoires from the approved ICANN Root Zone Label Generation Rules (LGR-1) [ICANN-LGR1] (or its successor documents).

It is the responsibility of the registry to determine which, if any, of those recommendations are applicable and to further subset or extend them as needed. For example, several of the recommendations are designed for the root zone and therefore exclude digits and U+002D HYPHEN-MINUS; this restriction is not generally appropriate for other zones. On the other hand, some zones may be designed to not cater for all users of a given script, but perhaps only for the needs of selected languages, in which case a more selective repertoire may be appropriate.

In making these determinations, a registry SHOULD follow the IAB guidance in RFC 6912 [RFC6912]. Those guidelines include a number of principles for use in making decisions about allowable code points. In addition, that document notes that the closer a particular zone is to the root, the more restrictive the space of permitted labels should be. RFC 5894 provides some suggestions for any registry that may decide to reduce opportunities for confusion or attacks by constructing policies that disallow characters used in historic writing systems (whether these be archaic scripts or extensions of modern scripts for historic or obsolete orthographies) or characters whose use is restricted to specialized, or highly technical contexts. These suggestions were among the principles guiding the design of ICANN’s Maximal Starting Repertoires [LGR-Procedure].

Particularly for a zone for which all labels to be delegated are not for the use of the same organization or enterprise, a registry decision to allow only those code points in the full repertoire of the MSR (plus digits and hyphen) would already avoid a number of issues inherent in a more permissive policy like "use anything permitted by IDNA2008", while still supporting the native languages and scripts for the vast majority of users today. However, it is unlikely, by itself, to fully satisfy the mandate set out above for three reasons.

1. The MSR, like the set of code points permissible under IDNA2008 itself, was conceived merely as an upper bound on permissible letter code points (it excludes digits and the hyphen). It was always intended to be used as a starting point for setting registry policy, with the expectation that some of the code

points in the MSR would not be included in the final registry policy, whether for lack of actual usage, or for being inherently problematic.

2. It was recognized that many scripts require contextual rules for many more code points than are covered by CONTEXTO or CONTEXTJ rules defined in IDNA2008. This is particularly true for combining marks, typically used to encode diacritics, tone marks, vowel signs and the like. While, theoretically, any combining mark may occur in any context in Unicode, in practice rendering and other software that users rely on in viewing or entering labels will not support arbitrary combining sequences, or indeed arbitrary combinations of code points, in the case of complex scripts.

   Contextual rules are required to limit allowable code point sequences to those that can be expected to be rendered reliably. Identifying those requires knowledge about the way code points are used in a script, whence the mandate for registries to only support code points they understand. In this, some of the other recommendations, such as the Informational RFCs for specific scripts (e.g., Cyrillic [RFC5992]) or languages (e.g., Arabic [RFC5564] or Chinese [RFC4713]), or the Root Zone LGRs developed by ICANN, may provide useful guidance.

3. Third, because of the widely accepted practice of limiting any given label to a single script, a universal repertoire, such as the MSR, would have to be divided on a per script basis into subrepertoires to make it useful, with some of those repertoires overlapping, for example, in the case of East Asian shared usage of the Han ideographs.

Registries choosing to make exceptions and allow code points that recommendations such as the MSR do not allow should make such decisions only with great care and only if they have considerable understanding of, and great confidence in, their appropriateness. The obvious exception from the MSR would be to allow digits and the hyphen. Neither were allowed by the MSR, but only because they are not allowed in the Root Zone.

Nothing in this document permits a registry to allow code points or labels that are disallowed or otherwise prohibited by IDNA2008.

4. Other corrections and updates

After the initial IDNA2008 documents were published (and RFC 5892 was updated for Unicode 6.0 by RFC 6452 [RFC6452]) several errors or instances of confusing text were noted. For the convenience of the

community, the relevant corrections for RFC 5890 and 5891 are noted below and update the corresponding documents. There are no errata for RFC 5893 or 5894 as of the date this document was published. Because further updates to RFC 5892 would require addressing other pending issues, the outstanding erratum for that document is not considered here. For consistency with the original documents, references to Unicode 5.0 are preserved.

4.1. Updates to RFC 5890

The outstanding errata against RFC 5890 (Errata ID 4695, 4696, 4823, and 4824 [RFC-Editor-5890Errata]) are all associated with the same issue, the number of Unicode characters that can be associated with a maximum-length (63 octet) A-label. In retrospect and contrary to some of the suggestions in the errata, that value should not be expressed in octets because RFC 5890 and the other IDNA 2008 documents are otherwise careful to not specify Unicode encoding forms but, instead, work exclusively with Unicode code points. Consequently the relevant material in RFC 5890 should be corrected as follows:

Section 2.3.2.1

Old: expansion of the A-label form to a U-label may produce strings that are much longer than the normal 63 octet DNS limit (potentially up to 252 characters).

New: expansion of the A-label form to a U-label may produce strings that are much longer than the normal 63 octet DNS limit (See Section 4.2).

Comment: If the length limit is going to be a source of confusion or careful calculations, it should appear in only one place.

Section 4.2

Old: Because A-labels (the form actually used in the DNS) are potentially much more compressed than UTF-8 (and UTF-8 is, in general, more compressed that UTF-16 or UTF-32), U-labels that obey all of the relevant symmetry (and other) constraints of these documents may be quite a bit longer, potentially up to 252 characters (Unicode code points).

New: A-labels (the form actually used in the DNS) and the Punycode algorithm used as part of the process to produce them [RFC3492] are strings that are potentially much more compressed than any standard Unicode Encoding Form. [[CREF1: Do we need a reference for this here??]] A 63 octet A-label cannot

represent more than 58 Unicode code points (four octet overhead and the requirement that at least one character lie outside the ASCII range) but implementations allocating buffer space for the conversion should allow significantly more space depending on the encoding form they are using.

4.2. Updates to RFC 5891

Errata ID 3969: Improve reference for combining marks There is only one erratum for RFC 5891, Errata ID 3969 [RFC5891Erratum]. Combining marks are explained in the cited section, but not, as the text indicates, exactly defined.

Old: The Unicode string MUST NOT begin with a combining mark or combining character (see The Unicode Standard, Section 2.11 [Unicode] for an exact definition).

New: The Unicode string MUST NOT begin with a combining mark or combining character (see The Unicode Standard, Section 2.11 [Unicode] for an explanation and Section 3.6, definition D52) for an exact definition).

Comment: When RFC 5891 is actually updated, the references in the text should be updated to the current version of Unicode and the section numbers checked.

5. Related Discussions

This document is one of a series of measures that have been suggested to address IDNA issues raised in other documents, including mechanisms for dealing with combining sequences and single-code point characters with the same appearance that normalization neither combines nor decomposes as IDNA2008 assumed [IDNA-Unicode], including the IAB response to that issue [IAB-2015], and to take a higher-level view of issues, demands, and proposals for new uses of the DNS. Those documents also include a discussion of issues with IDNA and character graphemes for which abstractions exist in Unicode in precomposed form but that can be generated from combining sequences and a suggested registry of code points known to be problematic [Freytag-troublesome]. The discussion of combining sequences and non-decomposing characters is intended to lay the foundation for an actual update to the IDNA code points document [RFC5892]. Such an update will presumably also address the existing errata against that document.

6. Security Considerations

As discussed in IAB recommendations about internationalized domain names [RFC4690], [RFC6912], and elsewhere, poor choices of strings for DNS labels can lead to opportunities for attacks, user confusion, and other issues less directly related to security. This document clarifies the importance of registries carefully establishing design policies for the labels they will allow and that having such policies and taking responsibility for them is a requirement, not an option. If that clarification is useful in practice, the result should be an improvement in security.

7. Acknowledgments

Many thanks to Patrik Faltstrom who provided an important review on the initial version.

8. IANA Considerations

[[CREF2: RFC Editor: Please remove this section before publication.]]

This memo includes no requests to or actions for IANA. In particular, it does not contain any provisions that would alter any IDNA-related registries or tables.

9. References

9.1. Normative References

[ICANN-LGR1] ICANN, "Root Zone Label Generation Rules (LGR-1)", June 2015, <https://www.icann.org/resources/pages/root-zone-lgr-2015-06-21-en>.

[ICANN-MSR2] ICANN, "Maximal Starting Repertoire Version 2 (MSR-2) for the Development of Label Generation Rules for the Root Zone", April 2015, <https://www.icann.org/news/announcement-2-2015-04-27-en>.

[RFC1591] Postel, J., "Domain Name System Structure and Delegation", RFC 1591, DOI 10.17487/RFC1591, March 1994, <https://www.rfc-editor.org/info/rfc1591>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <https://www.rfc-editor.org/info/rfc5890>.

[RFC5891] Klensin, J., "Internationalized Domain Names in Applications (IDNA): Protocol", RFC 5891, DOI 10.17487/RFC5891, August 2010, <https://www.rfc-editor.org/info/rfc5891>.

[RFC5891Erratum] "RFC 5891, "Internationalized Domain Names in Applications (IDNA): Protocol"", Errata ID 3969, April 2014, <http://www.rfc-editor.org/errata_search.php?rfc=5891>.

[RFC5894] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Background, Explanation, and Rationale", RFC 5894, DOI 10.17487/RFC5894, August 2010, <https://www.rfc-editor.org/info/rfc5894>.

9.2. Informative References

[Freytag-troublesome] Freytag, A., Klensin, J., and A. Sullivan, "Those Troublesome Characters: A Registry of Unicode Code Points Needing Special Consideration When Used in Network Identifiers", June 2017, <draft-freytag-troublesome-characters-01>.

[IAB-2015] Internet Architecture Board (IAB), "IAB Statement on Identifiers and Unicode 7.0.0", February 2015, <https://www.iab.org/documents/correspondence-reports-documents/2015-2/iab-statement-on-identifiers-and-unicode-7-0-0/>.

[IDNA-Unicode] Klensin, J. and P. Falstrom, "IDNA Update for Unicode 7.0.0", September 2017, <draft-klensin-idna-5892upd-unicode70-05>.

[LGR-Procedure] Internet Corporation for Assigned Names and Numbers (ICANN), "Procedure to Develop and Maintain the Label Generation Rules for the Root Zone in Respect of IDNA Labels", March 2013, <https://www.icann.org/en/system/files/files/draft-lgr-procedure-20mar13-en.pdf>.

[RFC-Editor-5890Errata]
RFC Editor, "RFC Errata: RFC 5890, "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", August 2010", Note to RFC Editor: Please figure out how you would like this referenced and make it so., Captured 2017-09-10, 2016, <https://www.rfc-editor.org/errata_search.php?rfc=5890>.

[RFC3492] Costello, A., "Punycode: A Bootstring encoding of Unicode for Internationalized Domain Names in Applications (IDNA)", RFC 3492, DOI 10.17487/RFC3492, March 2003, <https://www.rfc-editor.org/info/rfc3492>.

[RFC4690] Klensin, J., Faltstrom, P., Karp, C., and IAB, "Review and Recommendations for Internationalized Domain Names (IDNs)", RFC 4690, DOI 10.17487/RFC4690, September 2006, <https://www.rfc-editor.org/info/rfc4690>.

[RFC4713] Lee, X., Mao, W., Chen, E., Hsu, N., and J. Klensin, "Registration and Administration Recommendations for Chinese Domain Names", RFC 4713, DOI 10.17487/RFC4713, October 2006, <https://www.rfc-editor.org/info/rfc4713>.

[RFC5564] El-Sherbiny, A., Farah, M., Oueichek, I., and A. Al-Zoman, "Linguistic Guidelines for the Use of the Arabic Language in Internet Domains", RFC 5564, DOI 10.17487/RFC5564, February 2010, <https://www.rfc-editor.org/info/rfc5564>.

[RFC5892] Faltstrom, P., Ed., "The Unicode Code Points and Internationalized Domain Names for Applications (IDNA)", RFC 5892, DOI 10.17487/RFC5892, August 2010, <https://www.rfc-editor.org/info/rfc5892>.

[RFC5893] Alvestrand, H., Ed. and C. Karp, "Right-to-Left Scripts for Internationalized Domain Names for Applications (IDNA)", RFC 5893, DOI 10.17487/RFC5893, August 2010, <https://www.rfc-editor.org/info/rfc5893>.

[RFC5992] Sharikov, S., Miloshevic, D., and J. Klensin, "Internationalized Domain Names Registration and Administration Guidelines for European Languages Using Cyrillic", RFC 5992, DOI 10.17487/RFC5992, October 2010, <https://www.rfc-editor.org/info/rfc5992>.

[RFC6452] Faltstrom, P., Ed. and P. Hoffman, Ed., "The Unicode Code Points and Internationalized Domain Names for Applications (IDNA) - Unicode 6.0", RFC 6452, DOI 10.17487/RFC6452, November 2011, <https://www.rfc-editor.org/info/rfc6452>.

[RFC6912] Sullivan, A., Thaler, D., Klensin, J., and O. Kolkman, "Principles for Unicode Code Point Inclusion in Labels in the DNS", RFC 6912, DOI 10.17487/RFC6912, April 2013, <https://www.rfc-editor.org/info/rfc6912>.

Appendix A. Change Log

RFC Editor: Please remove this appendix before publication.

A.1. Changes from version -00 (2017-03-11) to -01

- Added Acknowledgments and adjusted references.
- Filled in Section 4 with updates to respond to errata.
- Added Section 5 to discuss relationships to other documents.
- Modified the Abstract to note specifically updated documents.
- Several small editorial changes and corrections.

Authors' Addresses

John C Klensin
1770 Massachusetts Ave, Ste 322
Cambridge, MA 02140
USA

Phone: +1 617 245 1457
Email: john-ietf@jck.com

Asmus Freytag
ASMUS, Inc.

Email: asmus@unicode.org

Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 16, 2019

Y. Collet
M. Kucherawy, Ed.
Facebook

July 15, 2018

Zstandard Compression and The application/zstd Media Type
draft-kucherawy-dispatch-zstd-03

Abstract

Zstandard, or "zstd" (pronounced "zee standard"), is a data compression mechanism. This document describes the mechanism, and registers a media type and content encoding to be used when transporting zstd-compressed content via Multipurpose Internet Mail Extensions (MIME).

Despite use of the word "standard" as part of its name, readers are advised that this document is not an Internet Standards Track specification, and is being published for informational purposes only.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 16, 2019.

Copyright Notice

Table of Contents

1. Introduction

Zstandard, or "zstd" (pronounced "zee standard") is a data compression mechanism, akin to gzip [RFC1952].

Despite use of the word "standard" as part of its name, readers are advised that this document is not an Internet Standards Track specification, and is being published for informational purposes only.

This document describes the Zstandard format. Also, to enable the transport of a data object compressed with Zstandard, this document registers a media type that can be used to identify such content when it is used in a payload encoded using Multipurpose Internet Mail Extensions (MIME).

2. Definitions

Some terms used elsewhere in this document are defined here for clarity.

uncompressed: Describes an arbitrary set of bytes in their original form, prior to being subjected to compression.

compress, compression: The act of processing a set of bytes via the compression mechanism described here.

compressed: Describes the result of passing a set of bytes through this mechanism. The original input has thus been compressed.

decompress, decompression: The act of processing a set of bytes through the inverse of the compression mechanism described here, in an attempt to recover the original set of bytes prior to compression.

decompressed: Describes the result of passing a set of bytes through the reverse of this mechanism. When this is successful, the decompressed payload and the uncompressed payload are indistinguishable.

encode: The process of translating data from one form to another; this may include compression or it may refer to other translations done as part of this specification.

decode: The reverse of "encode"; describes a process of reversing a prior encoding to recover the original content.

frame: Content compressed by Zstandard is transformed into a Zstandard frame. Multiple frames can be appended into a single file or stream. A frame is completely independent, has a defined beginning and end, and a set of parameters which tells the decoder how to decompress it.

block: A frame encapsulates one or multiple blocks. Each block can be compressed or not, and has a guaranteed maximum content size, which depends on frame parameters. Unlike frames, each block depends on previous blocks for proper decoding. However, each block can be decompressed without waiting for its successor, allowing streaming operations.

natural order: A sequence or ordering of objects or values that is typical of that type of object or value. A set of unique integers, for example, is in "natural order" if when progressing from one element in the set or sequence to the next, there is never a decrease in value.

The naming convention for identifiers within the specification is Mixed_Case_With_Underscores. Identifiers inside square brackets indicate the identifier is optional in the presented context.

3. Compression Algorithm

This section describes the Zstandard algorithm.

The purpose of this document is to define a lossless compressed data format, that is independent of CPU type, operating system, file system and character set, and is suitable for file compression, pipe and streaming compression, using the Zstandard algorithm. The text of the specification assumes a basic background in programming at the level of bits and other primitive data representations.

The data can be produced or consumed, even for an arbitrarily long sequentially presented input data stream, using only an a priori bounded amount of intermediate storage, and hence can be used in data communications. The format uses the Zstandard compression method, and optional xxHash-64 checksum method [XXHASH], for detection of data corruption.

The data format defined by this specification does not attempt to allow random access to compressed data.

Unless otherwise indicated below, a compliant compressor must produce data sets that conform to the specifications presented here. However, it does not need to support all options.

A compliant decompressor must be able to decompress at least one working set of parameters that conforms to the specifications presented here. It may also ignore informative fields, such as the checksum. Whenever it does not support a parameter defined in the compressed stream, it must produce a non-ambiguous error code and associated error message explaining which parameter is unsupported.

This specification is intended for use by implementers of software to compress data into Zstandard format and/or decompress data from Zstandard format. The Zstandard format is supported by an open source reference implementation, written in portable C, and available at [ZSTD].

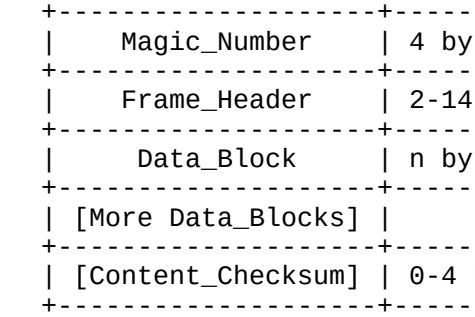
3.1. Frames

Zstandard compressed data is made up of one or more frames. Each frame is independent and can be decompressed independently of other frames. The decompressed content of multiple concatenated frames is the concatenation of each frame's decompressed content.

There are two frame formats defined for Zstandard: Zstandard frames and Skippable frames. Zstandard frames contain compressed data, while skippable frames contain custom user metadata.

3.1.1. Zstandard Frames

The structure of a single Zstandard frame is as follows:



Magic_Number: Four bytes, little-endian format. Value: 0xFD2FB528.

Frame_Header: Two to 14 bytes, detailed in Section 3.1.1.1.

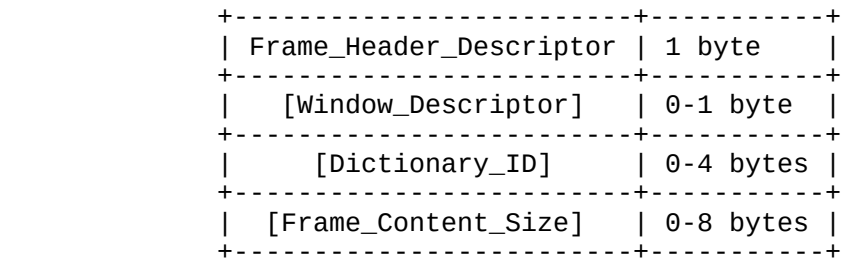
Data_Block: Detailed in Section 3.1.1.2. This is where data appears.

Content_Checksum: An optional 32-bit checksum, only present if Content_Checksum_Flag is set. The content checksum is the result of the XXH64() hash function [XXHASH] digesting the original (decoded) data as input, and a seed of zero. The low four bytes of the checksum are stored in little-endian format.

The magic number was selected to be less probable to find at the beginning of an arbitrary file. It avoids trivial patterns (0x00, 0xFF, repeated bytes, increasing bytes, etc.), contains byte values outside of ASCII range, and doesn't map into UTF-8 space, all of which reduce the likelihood of its appearance at the top of a text file.

3.1.1.1. Frame Header

The frame header has a variable size, with a minimum of two bytes and up to 14 bytes depending on optional parameters. The structure of Frame_Header is as follows:



3.1.1.1.1. Frame_Header_Descriptor

The first header's byte is called the Frame_Header_Descriptor. It describes which other fields are present. Decoding this byte is enough to tell the size of Frame_Header.

Bit Number	Field Name
7-6	Frame_Content_Size_Flag
5	Single_Segment_Flag
4	(unused)
3	(reserved)
2	Content_Checksum_Flag
1-0	Dictionary_ID_Flag

In this table, bit 7 is the highest bit, while bit 0 is the lowest one.

3.1.1.1.1.1. Frame_Content_Size_Flag

This is a two-bit flag (equivalent to Frame_Header_Descriptor right-shifted six bits) specifying whether Frame_Content_Size (the decompressed data size) is provided within the header. Flag_Value provides FCS_Field_Size, which is the number of bytes used by Frame_Content_Size according to the following table:

Flag_Value	0	1	2	3
FCS_Field_Size	0 or 1	2	4	8

When Flag_Value is 0, FCS_Field_Size depends on Single_Segment_Flag: If Single_Segment_Flag is set, FCS_Field_Size is 1. Otherwise, FCS_Field_Size is 0; Frame_Content_Size is not provided.

3.1.1.1.1.2. Single_Segment_Flag

If this flag is set, data must be regenerated within a single continuous memory segment.

In this case, Window_Descriptor byte is skipped, but Frame_Content_Size is necessarily present. As a consequence, the decoder must allocate a memory segment of size equal or larger than Frame_Content_Size.

In order to protect the decoder from unreasonable memory

requirements, a decoder is allowed to reject a compressed frame that requests a memory size beyond the decoder's authorized range.

For broader compatibility, decoders are recommended to support memory sizes of at least 8 MB. This is only a recommendation; each decoder is free to support higher or lower limits, depending on local limitations.

3.1.1.1.1.3. Unused Bit

A decoder compliant with this specification version shall not interpret this bit. It might be used in a future version, to signal a property which is not mandatory to properly decode the frame. An encoder compliant with this specification must set this bit to zero.

3.1.1.1.1.4. Reserved Bit

This bit is reserved for some future feature. Its value must be zero. A decoder compliant with this specification version must ensure it is not set. This bit may be used in a future revision, to signal a feature that must be interpreted to decode the frame correctly.

3.1.1.1.1.5. Content_Checksum_Flag

If this flag is set, a 32-bits Content_Checksum will be present at the frame's end. See the description of Content_Checksum above.

3.1.1.1.1.6. Dictionary_ID_Flag

This is a two-bit flag (= Frame_Header_Descriptor & 0x3) indicating whether a dictionary ID is provided within the header. It also specifies the size of this field as DID_Field_Size:

Flag_Value	0	1	2	3
DID_Field_Size	0	1	2	4

3.1.1.1.2. Window Descriptor

Provides guarantees on minimum memory buffer required to decompress a frame. This information is important for decoders to allocate enough memory.

The Window_Descriptor byte is optional. When Single_Segment_Flag is set, Window_Descriptor is not present. In this case, Window_Size is

Frame_Content_Size, which can be any value from 0 to 2^64-1 bytes (16 ExaBytes).

Bit numbers	7-3	2-0
Field name	Exponent	Mantissa

The minimum memory buffer size is called Window_Size. It is described by the following formulae:

```
windowLog = 10 + Exponent;
windowBase = 1 << windowLog;
windowAdd = (windowBase / 8) * Mantissa;
Window_Size = windowBase + windowAdd;
```

The minimum Window_Size is 1 KB. The maximum Window_Size is (1<<41) + 7*(1<<38) bytes, which is 3.75 TB.

In general, larger Window_Size values tend to improve compression ratio, but at the cost of increased memory usage.

To properly decode compressed data, a decoder will need to allocate a buffer of at least Window_Size bytes.

In order to protect decoders from unreasonable memory requirements, a decoder is allowed to reject a compressed frame which requests a memory size beyond decoder's authorized range.

For improved interoperability, it's recommended for decoders to support values of Window_Size up to 8 MB, and for encoders not to generate frames requiring a Window_Size larger than 8 MB. It's merely a recommendation though, and decoders are free to support larger or lower limits, depending on local limitations.

3.1.1.1.3. Dictionary_ID

This is a variable size field, which contains the ID of the dictionary required to properly decode the frame. This field is optional. When it's not present, it's up to the decoder to know which dictionary to use.

Dictionary_ID field size is provided by DID_Field_Size. DID_Field_Size is directly derived from the value of Dictionary_ID_Flag. One byte can represent an ID 0-255; two bytes can represent an ID 0-65535; four bytes can represent an ID 0-4294967295. Format is little-endian.

It is permitted to represent a small ID (for example 13) with a large four-byte dictionary ID, even if it is less efficient.

Within private environments, any dictionary ID can be used. However, for frames and dictionaries distributed in public space, Dictionary_ID must be attributed carefully. The following ranges are reserved for use only with dictionaries that have been registered with IANA (see Section 6.3):

low range: <= 32767

high range: >= (1 << 31)

Any other value for Dictionary_ID can be used by private arrangement between participants.

Any payload presented for decompression that references an unregistered reserved dictionary ID results in an error.

3.1.1.1.4. Frame Content Size

This is the original (uncompressed) size. This information is optional. Frame_Content_Size uses a variable number of bytes, provided by FCS_Field_Size. FCS_Field_Size is provided by the value of Frame_Content_Size_Flag. FCS_Field_Size can be equal to 0 (not present), 1, 2, 4 or 8 bytes.

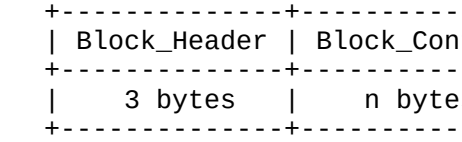
FCS Field Size	Range
0	unknown
1	0 - 255
2	256 - 65791
4	0 - 2^32 - 1
8	0 - 2^64 - 1

Frame_Content_Size format is little-endian. When FCS_Field_Size is 1, 4 or 8 bytes, the value is read directly. When FCS_Field_Size is 2, the offset of 256 is added. It’s allowed to represent a small size (for example 18) using any compatible variant.

3.1.1.2. Blocks

After Magic_Number and Frame_Header, there are some number of blocks. Each frame must have at least one block, but there is no upper limit on the number of blocks per frame.

The structure of a block is as follows:



Block_Header	Block_Content
3 bytes	n bytes

Block_Header uses three bytes, written using little-endian convention. It contains three fields:

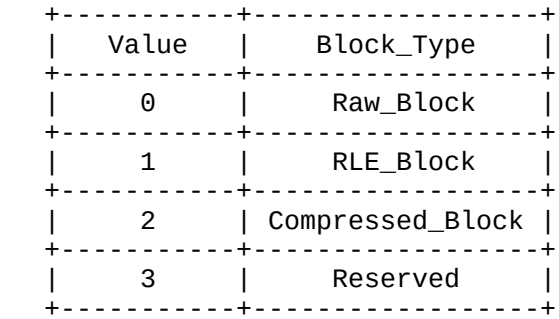
Last_Block	Block_Type	Block_Size
bit 0	bits 1-2	bits 3-23

3.1.1.2.1. Last_Block

The lowest bit (Last_Block) signals whether this block is the last one. The frame will end after this last block. It may be followed by an optional Content_Checksum (see Section 3.1.1).

3.1.1.2.2. Block_Type

The next two bits represent the Block_Type. There are four block types:



Value	Block_Type
0	Raw_Block
1	RLE_Block
2	Compressed_Block
3	Reserved

Raw_Block: This is an uncompressed block. Block_Content contains Block_Size bytes.

RLE_Block: This is a single byte, repeated Block_Size times. Block_Content consists of a single byte. On the decompression side, this byte must be repeated Block_Size times.

Compressed_Block: This is a compressed block as described in Section 3.1.1.3. Block_Size is the length of Block_Content, namely the compressed data. The decompressed size is not known, but its maximum possible value is guaranteed (see below).

Reserved: This is not a block. This value cannot be used with the current specification. If such a value is present, it is considered to be corrupt data.

3.1.1.2.3. Block_Size

The upper 21 bits of Block_Header represent the Block_Size. Block_Size is the size of the block excluding the header. A block can contain any number of bytes (even zero), up to Block_Maximum_Decompressed_Size, which is the smallest of:

- Window_Size
- 128 KB

A Compressed_Block has the extra restriction that Block_Size is always strictly less than the decompressed size. If this condition cannot be respected, the block must be sent uncompressed instead (i.e., treated as a Raw_Block).

3.1.1.3. Compressed Blocks

To decompress a compressed block, the compressed size must be provided from Block_Size field within Block_Header.

A compressed block consists of two sections: a Literals Section (Section 3.1.1.3.1) and a Sequences_Section (Section 3.1.1.3.2). The results of the two sections are then combined to produce the decompressed data in Sequence Execution (Section 3.1.1.4).

To decode a compressed block, the following elements are necessary:

- Previous decoded data, up to a distance of Window_Size, or the beginning of the Frame, whichever is smaller. Single_Segment_Flag will be set in the latter case.

- List of "recent offsets" from the previous Compressed_Block.
- The previous Huffman tree, required by Treeless_Literals_Block type.
- Previous FSE decoding tables, required by Repeat_Mode, for each symbol type (literals lengths, match lengths, offsets).

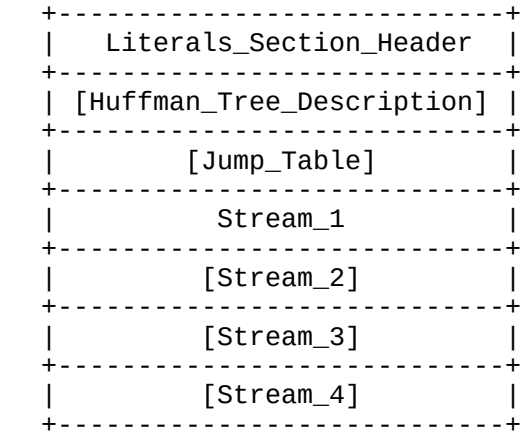
Note that decoding tables are not always from the previous Compressed_Block:

- Every decoding table can come from a dictionary.
- The Huffman tree comes from the previous Compressed_Literals_Block.

3.1.1.3.1. Literals_Section_Header

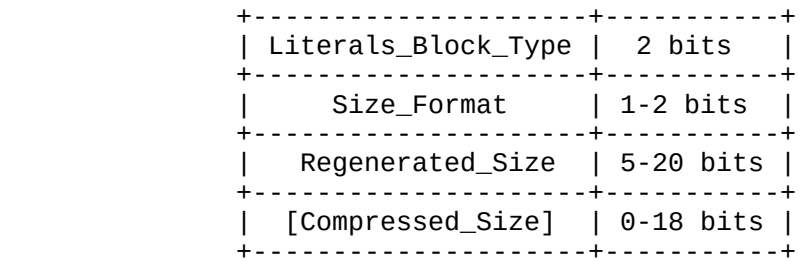
All literals are regrouped in the first part of the block. They can be decoded first, and then copied during Sequence Execution (see Section 3.1.1.4), or they can be decoded on the flow during Sequence Execution.

Literals can be stored uncompressed or compressed using Huffman prefix codes. When compressed, an optional tree description can be present, followed by one or four streams.



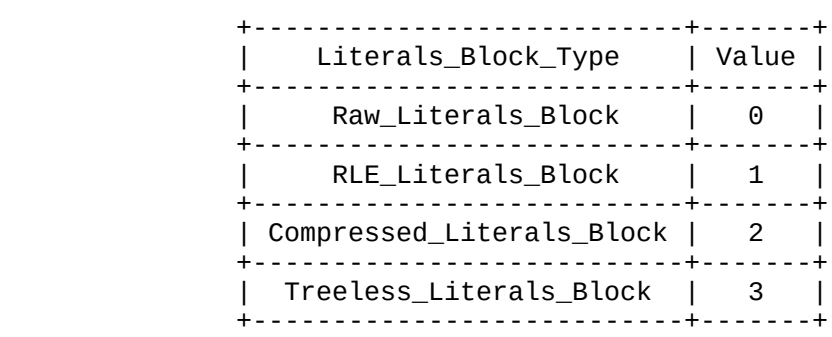
3.1.1.3.1.1. Literals_Section_Header

This field describes how literals are packed. It’s a byte-aligned variable-size bitfield, ranging from one to five bytes, using little-endian convention.



In this representation, bits at the top are the lowest bits.

The Literals_Block_Type field uses the two lowest bits of the first byte, describing four different block types:



Raw_Literals_Block: Literals are stored uncompressed. Literals_Section_Content is Regenerated_Size.

RLE_Literals_Block: Literals consist of a single byte value repeated Regenerated_Size times. Literals_Section_Content is one.

Compressed_Literals_Block: This is a standard Huffman-compressed block, starting with a Huffman tree description. See details below. Literals_Section_Content is Compressed_Size.

Treeless_Literals_Block: This is a Huffman-compressed block, using the Huffman tree from the previous Compressed_Literals_Block, or a dictionary if there is no previous Huffman-compressed literals block. Huffman_Tree_Description will be skipped. Note that if this mode is triggered without any previous Huffman-table in the frame (or dictionary, per Section 5), this should be treated as data corruption. Literals_Section_Content is Compressed_Size.

The Size_Format is divided into two families:

- For Raw_Literals_Block and RLE_Literals_Block, it's only necessary to decode Regenerated_Size. There is no Compressed_Size field.

- For Compressed_Block and Treeless_Literals_Block, it's required to decode both Compressed_Size and Regenerated_Size (the decompressed size). It's also necessary to decode the number of streams (1 or 4).

For values spanning several bytes, the convention is little-endian.

Size_Format for Raw_Literals_Block and RLE_Literals_Block uses 1 or 2 bits. Its value is (Literals_Section_Header[0]>>2) & 0x3.

Size_Format == 00 or 10: Size_Format uses one bit. Regenerated_Size uses five bits (value 0-31). Literals_Section_Header uses one byte. Regenerated_Size = Literal_Section_Header[0]>>3.

Size_Format == 01: Size_Format uses two bits. Regenerated_Size uses 12 bits (values 0-4095). Literals_Section_Header uses two bytes. Regenerated_Size = (Literals_Section_Header[0]>>4) + (Literals_Section_Header[1]<<4).

Size_Format == 11: Size_Format uses two bits. Regenerated_Size uses 20 bits (values 0-1048575). Literals_Section_Header uses three bytes. Regenerated_Size = (Literals_Section_Header[0]>>4) + (Literals_Section_Header[1]<<4) + (Literals_Section_Header[2]<<12)

Only Stream_1 is present for these cases. Note that it is permitted to represent a short value (for example 13) using a long format, even if it's less efficient.

Size_Format for Compressed_Literals_Block and Treeless_Literals_Block always uses two bits.

Size_Format == 00: A single stream. Both Regenerated_Size and Compressed_Size use ten bits (values 0-1023). Literals_Section_Header uses three bytes.

Size_Format == 01: Four streams. Both Regenerated_Size and Compressed_Size use ten bits (values 0-1023). Literals_Section_Header uses three bytes.

Size_Format == 10: Four streams. Both Regenerated_Size and Compressed_Size use 14 bits (values 0-16383). Literals_Section_Header uses four bytes.

Size_Format == 11: Four streams. Both Regenerated_Size and Compressed_Size use 18 bits (values 0-262143). Literals_Section_Header uses five bytes.

Both the Compressed_Size and Regenerated_Size fields follow little-endian convention. Note that Compressed_Size includes the size of the Huffman_Tree_Description when it is present.

3.1.1.3.1.2. Raw_Literals_Block

The data in Stream_1 is Regenerated_Size bytes long. It contains the raw literals data to be used during Sequence Execution (Section 3.1.1.3.2).

3.1.1.3.1.3. RLE_Literals_Block

Stream_1 consists of a single byte which should be repeated Regenerated_Size times to generate the decoded literals.

3.1.1.3.1.4. Compressed_Literals_Block and Treeless_Literals_Block

Both of these modes contain Huffman encoded data. For Treeless_Literals_Block the Huffman table comes the previously compressed literals block, or from a dictionary. (see Section 5).

3.1.1.3.1.5. Huffman_Tree_Description

This section is only present when the Literals_Block_Type type is Compressed_Literals_Block (2). The format of Huffman_Tree_Description can be found in Section 4.2.1. The size of Huffman_Tree_Description is determined during the decoding process. It must be used to determine where streams begin.

```
Total_Streams_Size = Compressed_Size
                     - Huffman_Tree_Description_Size
```

3.1.1.3.1.6. Jump_Table

The Jump_Table is only present when there are four Huffman-coded streams.

(Reminder: Huffman compressed data consists of either one or four Huffman-coded streams.)

If only one stream is present, it is a single bitstream occupying the entire remaining portion of the literals block, encoded as described within Section 4.2.2.

If there are four streams, Literals_Section_Header only provides enough information to know the decompressed and compressed sizes of all four streams combined. The decompressed size of each stream is equal to (Regenerated_Size+3)/4, except for the last stream which may be up to three bytes smaller, to reach a total decompressed size as specified in Regenerated_Size.

The compressed size of each stream is provided explicitly in the Jump_Table. The Jump_Table is six bytes long and consists of three two-byte little-endian fields, describing the compressed sizes of the first three streams. Stream4_Size is computed from Total_Streams_Size minus sizes of other streams.

```
Stream4_Size = Total_Streams_Size - 6
               - Stream1_Size - Stream2_Size
               - Stream3_Size
```

Note that if Stream1_Size + Stream2_Size + Stream3_Size exceeds Total_Streams_Size, the data are considered corrupted.

Each of these four bitstreams is then decoded independently as a Huffman-Coded stream, as described in Section 4.2.2.

3.1.1.3.2. Sequences_Section

A compressed block is a succession of sequences. A sequence is a literal copy command, followed by a match copy command. A literal copy command specifies a length. It is the number of bytes to be copied (or extracted) from the Literals Section. A match copy command specifies an offset and a length.

When all sequences are decoded, if there are literals left in the literal section, these bytes are added at the end of the block.

This is described in more detail in Section 3.1.1.4.

The Sequences_Section regroups all symbols required to decode commands. There are three symbol types: literals lengths, offsets, and match lengths. They are encoded together, interleaved, in a single "bitstream".

The Sequences_Section starts by a header, followed by optional probability tables for each symbol type, followed by the bitstream.

```
Sequences_Section_Header
  [Literals_Length_Table]
  [Offset_Table]
  [Match_Length_Table]
  bitStream
```

To decode the Sequences_Section, it's necessary to know its size. This size is deduced from the literals section size: Sequences_Section_Size = Block_Size - Literals_Section_Header - Literals_Section_Content

3.1.1.3.2.1. Sequences_Section_Header

This header consists of two items:

- Number_of_Sequences
- Symbol_Compression_Modes

Number_of_Sequences is a variable size field using between one and three bytes. If the first byte is "byte0":

- if (byte0 == 0): there are no sequences. The sequence section stops here. Decompressed content is defined entirely as Literals Section content. The FSE tables used in Repeat_Mode are not updated.
- if (byte0 < 128): Number_of_Sequences = byte0. Uses 1 byte.
- if (byte0 < 255): Number_of_Sequences = ((byte0 - 128) << 8) + byte1. Uses 2 bytes.
- if (byte0 == 255): Number_of_Sequences = byte1 + (byte2 << 8) + 0x7F00. Uses 3 bytes.

Symbol_Compression_Modes is a single byte, defining the compression mode of each symbol type.

Bit Number	Field Name
7-6	Literal_Lengths_Mode
5-4	Offsets_Mode
3-2	Match_Lengths_Mode
1-0	Reserved

```
  +------------+----------------------+
```

The last field, Reserved, must be all zeroes.

Literals_Lengths_Mode, Offsets_Mode, and Match_Lengths_Mode define the Compression_Mode of literals lengths, offsets, and match lengths symbols respectively. They follow the same enumeration:

Value	Compression_Mode
0	Predefined_Mode
1	RLE_Mode
2	FSE_Compressed_Mode
3	Repeat_Mode

Predefined_Mode: A predefined FSE (see Section 4.1) distribution table is used, defined in Section 3.1.1.3.2.2. No distribution table will be present.

RLE_Mode: The table description consists of a single byte, which contains the symbol’s value. THis symbol will be used for all sequences.

FSE_Compressed_Mode: Standard FSE compression. A distribution table will be present. The format of this distribution table is described in Section 4.1.1. Note that the maximum allowed accuracy log for literals length and match length tables is 9, and the maximum accuracy log for the offsets table is 8. This mode must not be used when only one symbol is present; RLE_Mode should be used instead (although any other mode will work).

Repeat_Mode: The table used in the previous Compressed_Block with Number_Of_Sequences > 0 will be used again, or if this is the first block, the table in the dictionary will be used. Note that this includes RLE_Mode, so if Repeat_Mode follows RLE_Mode, the same symbol will be repeated. It also includes Predefined_Mode, in which case Repeat_Mode will have the same outcome as Predefined_Mode. No distribution table will be present. If this mode is used without any previous sequence table in the frame (or dictionary; see Section 5) to repeat, this should be treated as corruption.

3.1.1.3.2.1.1. Sequence Codes for Lengths and Offsets

Each symbol is a code in its own context, which specifies Baseline and Number_of_Bits to add. Codes are FSE compressed, and interleaved with raw additional bits in the same bitstream.

Literals length codes are values ranging from 0 to 35 inclusive. They define lengths from 0 to 131071 bytes. The literals length is equal to the decoded Baseline plus the result of reading Number_of_Bits bits from the bitstream, as a little-endian value.

Literals_Length_Code	Baseline	Number_of_Bits
0-15	length	0
16	16	1
17	18	1
18	20	1
19	22	1
20	24	2
21	28	2
22	32	3
23	40	3
24	48	4
25	64	6
26	128	7
27	256	8
28	512	9
29	1024	10
30	2048	11
31	4096	12
32	8192	13
33	16384	14
34	32768	15
35	65536	16

Match length codes are values ranging from 0 to 52 included. They define lengths from 3 to 131074 bytes. The match length is equal to

the decoded Baseline plus the result of reading Number_of_Bits bits from the bitstream, as a little-endian value.

Match_Length_Code	Baseline	Number_of_Bits
0-31	Match_Length_Code + 3	0
32	35	1
33	37	1
34	39	1
35	41	1
36	43	2
37	47	2
38	51	3
39	59	3
40	67	4
41	83	4
42	99	5
43	131	7
44	259	8
45	515	9
46	1027	10
47	2051	11
48	4099	12
49	8195	13
50	16387	14
51	32771	15
52	65539	16

Offset codes are values ranging from 0 to N.

A decoder is free to limit its maximum supported value for N. Support for values of at least 22 is recommended. At the time of this writing, the reference decoder supports a maximum N value of 31.

An offset code is also the number of additional bits to read in little-endian fashion, and can be translated into an Offset_Value using the following formulas:

```
Offset_Value = (1 << offsetCode) + readNBits(offsetCode);
if (Offset_Value > 3) Offset = Offset_Value - 3;
```

This means that maximum Offset_Value is (2^(N+1))-1, supporting back-reference distance up to (2^(N+1))-4, but is limited by the maximum back-reference distance (see Section 3.1.1.1.2).

Offset_Value from 1 to 3 are special: they define "repeat codes". This is described in more detail in Section 3.1.1.5.

3.1.1.3.2.1.2. Decoding Sequences

FSE bitstreams are read in reverse direction than written. In zstd, the compressor writes bits forward into a block and the decompressor must read the bitstream backwards.

To find the start of the bitstream it is therefore necessary to know the offset of the last byte of the block which can be found by counting Block_Size bytes after the block header.

After writing the last bit containing information, the compressor writes a single 1-bit and then fills the byte with 0-7 zero bits of padding. The last byte of the compressed bitstream cannot be zero for that reason.

When decompressing, the last byte containing the padding is the first byte to read. The decompressor needs to skip 0-7 initial zero bits until the first one bit occurs. Afterwards, the useful part of the bitstream begins.

FSE decoding requires a ’state’ to be carried from symbol to symbol. For more explanation on FSE decoding, see Section 4.1.

For sequence decoding, a separate state keeps track of each literal lengths, offsets, and match lengths symbols. Some FSE primitives are also used. For more details on the operation of these primitives, see Section 4.1.

The bitstream starts with initial FSE state values, each using the required number of bits in their respective accuracy, decoded previously from their normalized distribution. It starts with Literals_Length_State, followed by Offset_State, and finally Match_Length_State.

Note that all values are read backward, so the 'start' of the bitstream is at the highest position in memory, immediately before the last one bit for padding.

After decoding the starting states, a single sequence is decoded Number_Of_Sequences times. These sequences are decoded in order from first to last. Since the compressor writes the bitstream in the forward direction, this means the compressor must encode the sequences starting with the last one and ending with the first.

For each of the symbol types, the FSE state can be used to determine the appropriate code. The code then defines the baseline and number of bits to read for each type. The description of the codes for how to determine these values can be found in Section 3.1.1.3.2.1.

Decoding starts by reading the Number_of_Bits required to decode Offset. It then does the same for Match_Length, and then for Literals_Length. This sequence is then used for sequence execution (see Section 3.1.1.4).

If it is not the last sequence in the block, the next operation is to update states. Using the rules pre-calculated in the decoding tables, Literals_Length_State is updated, followed by Match_Length_State, and then Offset_State. See Section 4.1 for details on how to update states from the bitstream.

This operation will be repeated Number_of_Sequences times. At the end, the bitstream shall be entirely consumed, otherwise the bitstream is considered corrupted.

3.1.1.3.2.2. Default Distributions

If Predefined_Mode is selected for a symbol type, its FSE decoding table is generated from a predefined distribution table defined here. For details on how to convert this distribution into a decoding table, see Section 4.1.

3.1.1.3.2.2.1. Literals Length

The decoding table uses an accuracy log of 6 bits (64 states).

```
short literalsLength_defaultDistribution[36] =
  { 4, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1,
    2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 2, 1, 1, 1, 1, 1,
    -1,-1,-1,-1
  };
```

3.1.1.3.2.2.2. Match Length

The decoding table uses an accuracy log of 6 bits (64 states).

```
short matchLengths_defaultDistribution[53] =
  { 1, 4, 3, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,-1,-1,
    -1,-1,-1,-1,-1
  };
```

3.1.1.3.2.2.3. Offset Codes

The decoding table uses an accuracy log of 5 bits (32 states), and supports a maximum N value of 28, allowing offset values up to 536,870,908.

If any sequence in the compressed block requires a larger offset than this, it’s not possible to use the default distribution to represent it.

```
short offsetCodes_defaultDistribution[29] =
  { 1, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1
  };
```

3.1.1.4. Sequence Execution

Once literals and sequences have been decoded, they are combined to produce the decoded content of a block.

Each sequence consists of a tuple of (literals_length, offset_value, match_length), decoded as described in the Sequences_Section (Section 3.1.1.3.2). To execute a sequence, first copy literals_length bytes from the literals section to the output.

Then match_length bytes are copied from previous decoded data. The offset to copy from is determined by offset_value:

o if Offset_Value > 3, then the offset is Offset_Value - 3;

- if Offset_Value is from 1-3, the offset is a special repeat offset value. See Section 3.1.1.5 for how the offset is determined in this case.

The offset is defined as from the current position (after copying the literals), so an offset of 6 and a match length of 3 means that 3 bytes should be copied from 6 bytes back. Note that all offsets leading to previously decoded data must be smaller than Window_Size defined in Frame_Header_Descriptor (Section 3.1.1.1.1).

3.1.1.5. Repeat Offsets

As seen above, the first three values define a repeated offset and we will call them Repeated_Offset1, Repeated_Offset2, and Repeated_Offset3. They are sorted in recency order, with Repeated_Offset1 meaning "most recent one".

If offset_value is 1, then the offset used is Repeated_Offset1, etc.

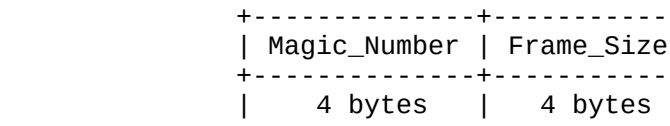
There is one exception: When the current sequence’s literals_length is 0, repeated offsets are shifted by one, so an offset_value of 1 means Repeated_Offset2, an offset_value of 2 means Repeated_Offset3, and an offset_value of 3 means Repeated_Offset1 - 1_byte.

For the first block, the starting offset history is populated with the following values: Repeated_Offset1 (1), Repeatead_Offset2 (4), and Repeatead_Offset3 (8), unless a dictionary is used, in which case they come from the dictionary.

Then each block gets its starting offset history from the ending values of the most recent Compressed_Block. Note that blocks that are not Compressed_Block are skipped; they do not contribute to offset history.

The newest offset takes the lead in offset history, shifting others back (up to its previous place if it was already present). This means that when Repeated_Offset1 (most recent) is used, history is unmodified. When Repeated_Offset2 is used, it is swapped with Repeated_Offset1. If any other offset is used, it becomes Repeated_Offset1 and the rest are shifted back by one.

3.1.2. Skippable Frames



```
  +--------------+------------+-----------+
```

Skippable frames allow the insertion of user-defined metadata into a flow of concatenated frames.

Skippable frames defined in this specification are compatible with skippable frames in [LZ4].

From a compliant decoder perspective, skippable frames simply need to be skipped, and their content ignored, resuming decoding after the skippable frame.

It should be noted that a skippable frame can be used to watermark a stream of concatenated frames embedding any kind of tracking information (even just a UUID). Users wary of such possibility should scan the stream of concatenated frames in an attempt to detect such frame for analysis or removal.

The fields are:

Magic_Number: Four bytes, little-endian format. Value: 0x184D2A5?, which means any value from 0x184D2A50 to 0x184D2A5F. All 16 values are valid to identify a skippable frame. This specification does not detail any specific tagging methods for skippable frames.

Frame_Size: This is the size, in bytes, of the following User_Data (without including the magic number nor the size field itself). This field is represented using four bytes, little-endian format, unsigned 32-bits. This means User_Data can’t be bigger than (2^32-1) bytes.

User_Data: This field can be anything. Data will just be skipped by the decoder.

4. Entropy Encoding

Two types of entropy encoding are used by the Zstandard format: FSE, and Huffman coding. Huffman is used to compress literals, while FSE is used for all other symbols (Literals_Length_Code, Match_Length_Code, offset codes) and to compress Huffman headers.

4.1. FSE

FSE, short for Finite State Entropy, is an entropy codec based on [ANS]. FSE encoding/decoding involves a state that is carried over between symbols, so decoding must be done in the opposite direction as encoding. Therefore, all FSE bitstreams are read from end to beginning. Note that the order of the bits in the stream is not

reversed; they are simply read in the reverse order from which they were written.

For additional details on FSE, see Finite State Entropy [FSE].

FSE decoding involves a decoding table that has a power of two size, and contains three elements: Symbol, Num_Bits, and Baseline. The base two logarithm of the table size is its Accuracy_Log. An FSE state value represents an index in this table.

To obtain the initial state value, consume Accuracy_Log bits from the stream as a little-endian value. The next symbol in the stream is the Symbol indicated in the table for that state. To obtain the next state value, the decoder should consume Num_Bits bits from the stream as a little-endian value and add it to Baseline.

4.1.1. FSE Table Description

To decode FSE streams, it is necessary to construct the decoding table. The Zstandard format encodes FSE table descriptions as described here.

An FSE distribution table describes the probabilities of all symbols from 0 to the last present one (included) on a normalized scale of (1 << Accuracy_Log). Note that there must be two or more symbols with nonzero probability.

A bitstream is read forward, in little-endian fashion. It is not necessary to know its exact size, since the size will be discovered and reported by the decoding process. The bitstream starts by reporting on which scale it operates. If low4bits designates the lowest four bits of the first byte, then Accuracy_Log = low4bits + 5.

This is followed by each symbol value, from 0 to the last present one. The number of bits used by each field is variable and depends on:

Remaining probabilities + 1: For example, presuming an Accuracy_Log of 8, and presuming 100 probabilities points have already been distributed, the decoder may read any value from 0 to (256 - 100 + 1) == 157, inclusive. Therefore, it must read log2sup(157) == 8 bits.

Value decoded: Small values use one fewer bit. For example, presuming values from 0 to 157 (inclusive) are possible, 255 - 157 = 98 values are remaining in an 8-bit field. The first 98 values (hence from 0 to 97) use only 7 bits, and values from 98 to 157 use 8 bits. This is achieved through this scheme:

Value read	Value decoded	Bits used
0 - 97	0 - 97	7
98 - 127	98 - 127	8
128 - 225	0 - 97	7
226 - 255	128 - 157	8

Symbol probabilities are read one by one, in order. The probability is obtained from Value decoded using the formula P = Value - 1. This means the value 0 becomes the negative probability -1. This is a special probability that means "less than 1". Its effect on the distribution table is described below. For the purpose of calculating total allocated probability points, it counts as 1.

When a symbol has a probability of zero, it is followed by a 2-bit repeat flag. This repeat flag tells how many probabilities of zeroes follow the current one. It provides a number ranging from 0 to 3. If it is a 3, another 2-bit repeat flag follows, and so on.

When the last symbol reaches a cumulated total of (1 << Accuracy_Log), decoding is complete. If the last symbol makes the cumulated total go above (1 << Accuracy_Log), distribution is considered corrupted.

Finally, the decoder can tell how many bytes were used in this process, and how many symbols are present. The bitstream consumes a round number of bytes. Any remaining bit within the last byte is simply unused.

The distribution of normalized probabilities is enough to create a unique decoding table. The table has a size of (1 << Accuracy_Log). Each cell describes the symbol decoded, and instructions to get the next state.

Symbols are scanned in their natural order for "less than 1" probabilities as described above. Symbols with this probability are being attributed a single cell, starting from the end of the table and retreating. These symbols define a full state reset, reading Accuracy_Log bits.

All remaining symbols are allocated in their natural order. Starting from symbol 0 and table position 0, each symbol gets allocated as many cells as its probability. Cell allocation is spread, not

linear; each successor position follows this rule:

```
position += (tableSize >> 1) + (tableSize >> 3) + 3;
position &= tableSize - 1;
```

A position is skipped if it is already occupied by a "less than 1" probability symbol. Position does not reset between symbols; it simply iterates through each position in the table, switching to the next symbol when enough states have been allocated to the current one.

The result is a list of state values. Each state will decode the current symbol.

To get the Number_of_Bits and Baseline required for the next state, it is first necessary to sort all states in their natural order. The lower states will need one more bit than higher ones. The process is repeated for each symbol.

For example, presuming a symbol has a probability of 5, it receives five state values. States are sorted in natural order. The next power of two is 8. The space of probabilities is divided into 8 equal parts. Presuming the Accuracy_Log is 7, this defines 128 states, and each share (divided by 8) is 16 in size. In order to reach 8, 8 - 5 = 3 lowest states will count "double", doubling the number of shares (32 in width), requiring one more bit in the process.

Baseline is assigned starting from the higher states using fewer bits, and proceeding naturally, then resuming at the first state, each taking its allocated width from Baseline.

state order	0	1	2	3	4
width	32	32	32	16	16
Number_of_Bits	5	5	5	4	4
range number	2	4	6	0	1
Baseline	32	64	96	0	16
range	32-63	64-95	96-127	0-15	16-31

The next state is determined from the current state by reading the

required Number_of_Bits, and adding the specified Baseline.

See Appendix B for the results of this process applied to the default distributions.

4.2. Huffman Coding

Zstandard Huffman-coded streams are read backwards, similar to the FSE bitstreams. Therefore, to find the start of the bitstream, it is necessary to know the offset of the last byte of the Huffman-coded stream.

After writing the last bit containing information, the compressor writes a single 1-bit and then fills the byte with 0-7 0 bits of padding. The last byte of the compressed bitstream cannot be 0 for that reason.

When decompressing, the last byte containing the padding is the first byte to read. The decompressor needs to skip 0-7 initial 0-bits and the first 1-bit that occurs. Afterwards, the useful part of the bitstream begins.

The bitstream contains Huffman-coded symbols in little-endian order, with the codes defined by the method below.

4.2.1. Huffman Tree Description

Prefix coding represents symbols from an a priori known alphabet by bit sequences (codewords), one codeword for each symbol, in a manner such that different symbols may be represented by bit sequences of different lengths, but a parser can always parse an encoded string unambiguously symbol-by-symbol.

Given an alphabet with known symbol frequencies, the Huffman algorithm allows the construction of an optimal prefix code using the fewest bits of any possible prefix codes for that alphabet.

The prefix code must not exceed a maximum code length. More bits improve accuracy but yield a larger header size, and require more memory or more complex decoding operations. This specification limits the maximum code length to 11 bits.

All literal values from zero (included) to the last present one (excluded) are represented by Weight with values from 0 to Max_Number_of_Bits. Transformation from Weight to Number_of_Bits follows this pseudocode:

```
if Weight == 0
  Number_of_Bits = 0
else
  Number_of_Bits = Max_Number_of_Bits + 1 - Weight
```

The last symbol's Weight is deduced from previously decoded ones, by completing to the nearest power of 2. This power of 2 gives Max_Number_of_Bits, the depth of the current tree.

For example, presume the following Huffman tree must be described:

literal value	Number_of_Bits
0	1
1	2
2	3
3	0
4	4
5	4

The tree depth is four, since its longest element uses four bits. (The longest elements are those with the smallest frequencies.) Value 5 will not be listed as it can be determined from the values for 0-4, nor will values above 5 as they are all 0. Values from 0 to 4 will be listed using Weight instead of Number_of_Bits. The pseudocode to determine Weight is:

```
if Number_of_Bits == 0
  Weight = 0
else
  Weight = Max_Number_of_Bits + 1 - Number_of_Bits
```

It gives the following series of weights:

literal value	Weight
0	4
1	3
2	2
3	0
4	1

The decoder will do the inverse operation: having collected weights of literals from 0 to 4, it knows the last literal, 5, is present with a non-zero weight. The weight of 5 can be determined by advancing to the next power of 2. The sum of 2^(Weight-1) (excluding 0's) is 15. The nearest power of 2 is 16. Therefore, Max_Number_of_Bits = 4 and Weight[5] = 16 - 15 = 1.

4.2.1.1. Huffman Tree Header

This is a single byte value (0-255), which describes how to the series of weights is encoded.

headerByte < 128: The series of weights is compressed using FSE (see below). The length of the FSE-compressed series is equal to headerByte (0-127).

headerByte >= 128: This is a direct representation, where each Weight is written directly as a four-bit field (0-15). They are encoded forward, two weights to a byte with the first weight taking the top four bits and the second taking the bottom four (e.g. the following operations could be used to read the weights:

```
Weight[0] = (Byte[0] >> 4)
Weight[1] = (Byte[0] & 0xf),
etc.
```

The full representation occupies ceiling(Number_of_Symbols/2) bytes, meaning it uses only full bytes even if Number_of_Symbols is odd. Number_of_Symbols = headerByte - 127. Note that maximum Number_of_Symbols is 255 - 127 = 128. If any literal has a value over 128, raw header mode is not possible and it is necessary to use FSE compression.

4.2.1.2. FSE Compression of Huffman Weights

In this case, the series of Huffman weights is compressed using FSE compression. It is a single bitstream with two interleaved states, sharing a single distribution table.

To decode an FSE bitstream, it is necessary to know its compressed size. Compressed size is provided by headerByte. It's also necessary to know its maximum possible decompressed size, which is 255, since literal values span from 0 to 255, and the last symbol's weight is not represented.

An FSE bitstream starts by a header, describing probabilities distribution. It will create a Decoding Table. For a list of Huffman weights, the maximum accuracy log is 6 bits. For more description see Section 4.1.1.

The Huffman header compression uses two states, which share the same FSE distribution table. The first state (State1) encodes the even indexed symbols, and the second (State2) encodes the odd indexes. State1 is initialized first, and then State2, and they take turns decoding a single symbol and updating their state. For more details on these FSE operations, see the FSE section.

The number of symbols to decode is determined by tracking the bitStream overflow condition: If updating state after decoding a symbol would require more bits than remain in the stream, it is assumed that extra bits are zero. Then, symbols for each of the final states are decoded and the process is complete.

4.2.1.3. Conversion from Weights to Huffman Prefix Codes

All present symbols will now have a Weight value. It is possible to transform weights into Number_of_Bits, using this formula:

```
if Weight > 0
    Number_of_Bits = Max_Number_of_Bits + 1 - Weight
else
    Number_of_Bits = 0
```

Symbols are sorted by Weight. Within the same Weight, symbols keep natural sequential order. Symbols with a Weight of zero are removed. Then, starting from lowest weight, prefix codes are distributed in sequential order.

For example, assume the following list of weights has been decoded:

Literal	Weight
0	4
1	3
2	2
3	0
4	1
5	1

Sorted by weight and then the natural sequential order, yielding the following distribution:

Literal	Weight	Number_Of_Bits	prefix codes
3	0	0	N/A
4	1	4	0000
5	1	4	0001
2	2	3	001
1	3	2	01
0	4	1	1

4.2.2. Huffman-coded Streams

Given a Huffman decoding table, it is possible to decode a Huffman-coded stream.

Each bitstream must be read backward, that is starting from the end up to the beginning. Therefore, it is necessary to know the size of each bitstream.

It is also necessary to know exactly which bit is the latest. This is detected by a final bit flag: the highest bit of latest byte is a final-bit-flag. Consequently, a last byte of 0 is not possible. And the final-bit-flag itself is not part of the useful bitstream.

Hence, the last byte contains between 0 and 7 useful bits.

Starting from the end, it is possible to read the bitstream in a little-endian fashion, keeping track of already used bits. Since the bitstream is encoded in reverse order, starting from the end, read symbols in forward order.

For example, if the literal sequence "0145" was encoded using above prefix code, it would be encoded (in reverse order) as:

Symbol	Encoding
5	0000
4	0001
1	01
0	1
Padding	00001

This results in the following two-byte bitstream:

```
00010000 00001101
```

Here is an alternative representation with the symbol codes separated by underscores:

```
0001_0000 00001_1_01
```

Reading the highest Max_Number_of_Bits bits, it’s possible to compare the extracted value to the decoding table, determining the symbol to decode and number of bits to discard.

The process continues up to reading the required number of symbols per stream. If a bitstream is not entirely and exactly consumed, hence reaching exactly its beginning position with all bits consumed, the decoding process is considered faulty.

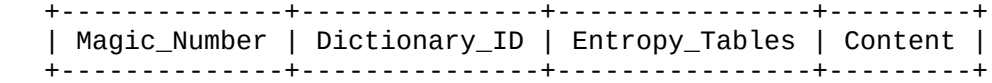
5. Dictionary Format

Zstandard is compatible with "raw content" dictionaries, free of any format restriction, except that they must be at least eight bytes. These dictionaries function as if they were just the Content part of a formatted dictionary.

However, dictionaries created by "zstd --train" in the reference implementation follow a specific format, described here.

Dictionaries are not included in the compressed content, but rather are provided out-of-band. That is, the Dictionary_ID identifies which should be used, but this specification does not describe the mechanism by which the dictionary is obtained prior to use during compression or decompression.

A dictionary has a size, defined either by a buffer limit or a file size. The general format is:



Magic_Number: 4 bytes ID, value 0xEC30A437, little-endian format

Dictionary_ID: 4 bytes, stored in little-endian format. Dictionary_ID can be any value, except 0 (which means no Dictionary_ID). It is used by decoders to check if they use the correct dictionary. If the frame is going to be distributed in a private environment, any Dictionary_ID can be used. However, for public distribution of compressed frames, the following ranges are reserved and shall not be used:

- low range : <= 32767
- high range : >= (2^31)

Entropy_Tables: Follow the same format as the tables in compressed blocks. See the relevant FSE and Huffman sections for how to decode these tables. They are stored in following order: Huffman tables for literals, FSE table for offsets, FSE table for match lengths, and FSE table for literals lengths. These tables populate the Repeat Stats literals mode and Repeat distribution mode for sequence decoding. It is finally followed by 3 offset values, populating repeat offsets (instead of using {1,4,8}), stored in order, 4-bytes little-endian each, for a total of 12 bytes. Each repeat offset must have a value less than the dictionary size.

Content: The rest of the dictionary is its content. The content act as a "past" in front of data to compress or decompress, so it can be referenced in sequence commands. As long as the amount of data decoded from this frame is less than or equal to Window_Size, sequence commands may specify offsets longer than the total length of decoded output so far to reference back to the dictionary, even parts of the dictionary with offsets larger than Window_Size.

After the total output has surpassed Window_Size, however, this is no longer allowed and the dictionary is no longer accessible.

6. IANA Considerations

This document contains two registration actions for IANA.

6.1. The 'application/zstd' Media Type

The 'application/zstd' media type identifies a block of data that is compressed using zstd compression. The data is a stream of bytes as described in this document. IANA is requested to add the following to the Media Types registry:

Type name: application

Subtype name: zstd

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: See Section 7 of [this document]

Interoperability considerations: N/A

Published specification: [this document]

Applications that use this media type: anywhere data size is an issue

Additional information:

Magic number(s): 4 Bytes, little-endian format. Value : 0xFD2FB528

File extension(s): zstd

Macintosh file type code(s): N/A

For further information: See [ZSTD]

Intended usage: common

Restrictions on usage: N/A

Author: Murray S. Kucherawy

Change Controller: IETF

Provisional registration: yes

6.2. Content Encoding

IANA is requested to add the following entry to the HTTP Content Coding Parameters subregistry within the Hypertext Transfer Protocol (HTTP) registry:

Name: zstd

Description: A stream of bytes compressed using the Zstandard protocol

Pointer to specification text: [this document]

6.3. Dictionaries

Work in progress incluces development of dictionaries that will optimize compression and decompression of particular types of data. Specification of such dictionaries for public use will necessitate registration of a code point from the reserved range described in Section 3.1.1.1.3 and its association with a specific dictionary.

However, there are at present no such dictionaries published for public use, so this document makes no immediate request of IANA to create such a registry.

7. Security Considerations

Any data compression method involves the reduction of redundancy in the data. Zstandard is no exception, and the usual precautions apply.

One should never compress together a message whose content must remain secret with a message generated by a third party. This can be used to guess the content of the secret message through analysis of entropy reduction. This was demonstrated in the [CRIME] attack, for example.

A decoder has to demonstrate capabilities to detect and prevent any kind of data tampering in the compressed frame from triggering system faults, such as reading or writing beyond allowed memory ranges.

This can be guaranteed either by the implementation language, or by careful bound checkings. Of particular note is the encoding of Number_of_Sequences values that cause the decoder to read into the block header (and beyond), as well as the indication of a Frame_Content_Size that is smaller than the actual decompressed data, in an attempt to trigger a buffer overflow. It is highly recommended to fuzz-test (i.e., provide invalid, unexpected, or random input and verify safe operation of) decoder implementations to test and harden their capability to detect bad frames and deal with them without any adverse system side-effect.

An attacker may provide correctly formed compressed frames with unreasonable memory requirements. A decoder must always control memory requirements and enforce some (system-specific) limits in order to protect memory usage from such scenarios.

Compression can be optimized by training a dictionary on a variety of related content payloads. This dictionary must then be available at the decoder for decompression of the payload to be possible. While this document does not specify how to acquire a dictionary for a given compressed payload, it is worth noting that third-party dictionaries may interact unexpectedly with a decoder, leading to possible memory or other resource exhaustion attacks. We expect such topics to be discussed in further detail in the Security Considerations section of a forthcoming RFC for dictionary acquisition and transmission, but highlight this issue now out of an abundance of caution.

As discussed in Section 3.1.2, it is possible to store arbitrary user metadata in skippable frames. While such frames are ignored during decompression of the data, they can be used as a watermark to track the path of the compressed payload.

8. Implementation Status

Source code for a C language implementation of a "Zstandard" compliant library is available at [ZSTD-GITHUB]. This implementation is considered to be the reference implementation and is production ready, implementing the full range of the specification. It is tested against security hazards, and widely deployed within Facebook infrastructure.

The reference version is speed optimised and highly portable. It has been proven to run safely on multiple architectures (x86, x64, ARM, MIPS, PowerPC, IA64) featuring 32 or 64-bits addressing schemes, little or big endian storage scheme, a number of different operating systems, UNIX (including Linux, BSD, OS-X and Solaris), and Windows, and a number of compilers (gcc, clang, visual, icc).

[RFC EDITOR: Please remove the remainder of this section prior to publication.]

The C reference version is also used to bind into multiple languages, a partial list of which (~20 of them) is being maintained at [ZSTD-OTHER].

The reference repository also contains an independently developed educational decoder, by Sean Purcell, created from the Zstandard format specification and built for clarity to help third party implementers. This is available at [ZSTD-EDU].

A specific version has been created for integration into the Linux kernel in order to provide compatibility with relevant memory restrictions. It was released in version 4.14 of the kernel. See [ZSTD-LINUX].

A Java native implementation of the decoder has been developed and open-sourced by the Presto team. This is available at [ZSTD-JAVA].

As of early July 2017, we are aware of one other decoder implementation in assembler, two full codec hardware implementations (programmable and ASIC) being actively developed, and a third one being evaluated. We are not permitted to disclose them at this stage.

The popular UNIX command line HTTP client "curl" has expressed intent to support zstd in a future release.

9. References

9.1. Normative References

[XXHASH] "XXHASH Algorithm", 2017, <http://www.xxhash.org>.

[ZSTD] "Zstandard - Real-time data compression algorithm", 2017, <http://www.zstd.net>.

9.2. Informative References

[ANS] "Asymmetric Numeral Systems: Entropy Coding Combining Speed of Huffman Coding with Compression Rate of Arithmetic Coding", 2017, <https://arvix.org/abs/1311.2540>.

[CRIME] "Compression Ratio Info-leak Made Easy", 2017, <https://en.wikipedia.org/wiki/CRIME>.

[FSE] "Finite State Entropy", 2017, <https://github.com/Cyan4973/FiniteStateEntropy/>.

[LZ4] "LZ4 Frame Format Description", 2017, <https://github.com/lz4/lz4/blob/master/doc/lz4_Frame_format.md>.

[RFC1952] Deutsch, P., "GZIP file format specification version 4.3", RFC 1952, DOI 10.17487/RFC1952, May 1996, <https://www.rfc-editor.org/info/rfc1952>.

[ZSTD-EDU] "Zstandard Educational Decoder", 2017, <https://github.com/facebook/zstd/tree/dev/doc/educational_decoder>.

[ZSTD-GITHUB] "Zstandard Github Repository", 2017, <https://github.com/facebook/zstd>.

[ZSTD-JAVA] "Zstandard Github Repository", 2017, <https://github.com/prestodb/presto/tree/master/presto-orc/src/main/java/com/facebook/presto/orc/zstd>.

[ZSTD-LINUX] "Zstandard Github Repository", 2017, <https://github.com/facebook/zstd/tree/dev/contrib/linux-kernel>.

[ZSTD-OTHER] "Zstandard Language Bindings", 2017, <http://facebook.github.io/zstd/#other-languages>.

Appendix A. Acknowledgments

zstd was developed by Yann Collet.

Bobo Bose-Kolanu, Felix Handte, Kyle Nekritz, Nick Terrell, and David Schleimer provided helpful feedback during the development of this document.

Appendix B. Decoding Tables for Predefined Codes

This appendix contains FSE decoding tables for the predefined literal length, match length, and offset codes. The tables have been constructed using the algorithm as given above in chapter "from normalized distribution to decoding tables". The tables here can be used as examples to crosscheck that an implementation build its decoding tables correctly.

B.1. Literal Length Code Table

State	Symbol	Number_Of_Bits	Base
0	0	0	0
0	0	4	0
1	0	4	16
2	1	5	32
3	3	5	0
4	4	5	0
5	6	5	0
6	7	5	0
7	9	5	0
8	10	5	0
9	12	5	0
10	14	6	0
11	16	5	0
12	18	5	0
13	19	5	0
14	21	5	0
15	22	5	0
16	24	5	0
17	25	5	32
18	26	5	0
19	27	6	0

20	29	6	0
21	31	6	0
22	0	4	32
23	1	4	0
24	2	5	0
25	4	5	32
26	5	5	0
27	7	5	32
28	8	5	0
29	10	5	32
30	11	5	0
31	13	6	0
32	16	5	32
33	17	5	0
34	19	5	32
35	20	5	0
36	22	5	32
37	23	5	0
38	25	4	0
39	25	4	16
40	26	5	32
41	28	6	0
42	30	6	0
43	0	4	48

44	1	4	16
45	2	5	32
46	3	5	32
47	5	5	32
48	6	5	32
49	8	5	32
50	9	5	32
51	11	5	32
52	12	5	32
53	15	6	0
54	17	5	32
55	18	5	32
56	20	5	32
57	21	5	32
58	23	5	32
59	24	5	32
60	35	6	0
61	34	6	0
62	33	6	0
63	32	6	0

B.2. Match Length Code Table

State	Symbol	Number_Of_Bits	Base
0	0	0	0

0	0	6	0
1	1	4	0
2	2	5	32
3	3	5	0
4	5	5	0
5	6	5	0
6	8	5	0
7	10	6	0
8	13	6	0
9	16	6	0
10	19	6	0
11	22	6	0
12	25	6	0
13	28	6	0
14	31	6	0
15	33	6	0
16	35	6	0
17	37	6	0
18	39	6	0
19	41	6	0
20	43	6	0
21	45	6	0
22	1	4	16
23	2	4	0

24	3	5	32
25	4	5	0
26	6	5	32
27	7	5	0
28	9	6	0
29	12	6	0
30	15	6	0
31	18	6	0
32	21	6	0
33	24	6	0
34	27	6	0
35	30	6	0
36	32	6	0
37	34	6	0
38	36	6	0
39	38	6	0
40	40	6	0
41	42	6	0
42	44	6	0
43	1	4	32
44	1	4	48
45	2	4	16
46	4	5	32
47	5	5	32

48	7	5	32
49	8	5	32
50	11	6	0
51	14	6	0
52	17	6	0
53	20	6	0
54	23	6	0
55	26	6	0
56	29	6	0
57	52	6	0
58	51	6	0
59	50	6	0
60	49	6	0
61	48	6	0
62	47	6	0
63	46	6	0

B.3. Offset Code Table

State	Symbol	Number_Of_Bits	Base
0	0	0	0
0	0	5	0
1	6	4	0
2	9	5	0

3	15	5	0
4	21	5	0
5	3	5	0
6	7	4	0
7	12	5	0
8	18	5	0
9	23	5	0
10	5	5	0
11	8	4	0
12	14	5	0
13	20	5	0
14	2	5	0
15	7	4	16
16	11	5	0
17	17	5	0
18	22	5	0
19	4	5	0
20	8	4	16
21	13	5	0
22	19	5	0
23	1	5	0
24	6	4	16
25	10	5	0
26	16	5	0

27	28	5	0
28	27	5	0
29	26	5	0
30	25	5	0
31	24	5	0

Authors' Addresses

Yann Collet
Facebook
1 Hacker Way
Menlo Park, CA 94025
United States

EMail: cyan@fb.com

Murray S. Kucherawy (editor)
Facebook
1 Hacker Way
Menlo Park, CA 94025
United States

EMail: msk@fb.com