Praxis
High Integrity
Systems

Document Set	Tokeneer ID Station	Reference S.P1229.50.1

Title : Formal Design

Synopsis : This document is the formal design of the core Token ID Station (TIS), which forms part of Tokeneer.

Contents : See table of contents

Status : Definitive

Issue Number : 1.3

Date : 19th August 2008

Copied To :

NSA
Randolph Johnson

SPRE Inc.

Praxis High Integrity Systems
Project Team

Front Sheet : Quality

Originators : Janet Barnes **Signed** :

Approver : David Cooper **Approved** :

0 DOCUMENT CONTROL

Changes History

All issues of this document have been type-checked with ƒUZZ *and have given no errors.*

Issue 0.1 (16th April 2003) Initial draft for formal review.

Issue 0.2 (28th May 2003) Updated following review comments from David Cooper.

Issue 1.0 (4th July 2003) Updated following completion of precondition checks. Incorporates changes resulting from NSA review comments on the Formal Specification.

Issue 1.1 (22nd July 2003) Updated following review comments from NSA. Added appendix containing an example refinement proof. Updated to incorporate corrections detailed in incident reports S.P1229.6.8-10, 16 and 18.

Issue 1.2 (22nd August 2003) Definitive issue correcting faults found during implementation and system test

- S.P1229.6.21 - Token information cleared too early in shutdown.
- S.P1229.6.24 - Correct classification of audit entries.
- S.P1225.6.28 - Audit Element descriptions missing for archive entries.
- S.P1229.6.30 - *AdminFinishOpC* missing from *FinishArchiveLogContext*.
- S.P1229.6.31 - Wrong description for audit entry in *FinishArchiveLogBadMatchC*.
- S.P1229.6.32 - Improve poor text messages on screen.
- S.P1229.6.33 - Make initial configuration realistic.
- S.P1229.6.34 - Contraints required on config data to ensure issued auth certs allow entry.
- S.P1229.6.35 - Audit entry required for *AdminTokenTimeout*.
- S.P1229.6.36 - Screen should show busy message when a user entry is in progress.
- S.P1229.6.38 - Operation failures should be reported on screen.
- S.P1229.6.42 - System faults need not always be critical.

Issue 1.3 (19th August 2008) Updated for public release.

Changes Forecast

None. This document is now under change control.

References

1 The Z Notation: A Reference Manual, J.M Spivey, Prentice Hall, Second Edition, 1992

2 TIS Software Requirements Specification, S.P1229.41.1.

3 TIS Kernel Protection Profile, SPRE Inc, Version 1.0, 5 February 2003.

4 TIS Formal Specification, S.P1229.50.1.

Abbreviations

AA	Attribute Authority
ATR	Answer-to-Reset
CA	Certification Authority
FAR	False Acceptance Rate
I&A	Identification and Authentication
RSA	Rivest Shamir Adelman algorithm
SPARK	SPADE Ada Kernel (analysable Ada subset from Praxis)
SRS	Software Requirements Specification
TIS	Token ID Station

1 TABLE OF CONTENTS

2 INTRODUCTION

In order to demonstrate that developing highly secure systems to the level of rigour required by the higher assurance levels of the Common Criteria is possible, the NSA has asked Praxis High Integrity Systems to undertake a research project to develop part of an existing secure system (the Tokeneer System) in accordance with their high-integrity development process. This development work will then be used to show the security community that is is possible to develop secure systems rigorously in a cost effective manner.

This document is the formal design, written using the Z notation. This document specifies the behaviour of the core of the Token ID Station (TIS) that is being developed. It documents the third step in the Praxis high integrity systems development approach. The whole process consists of:

1. Requirements Analysis (the REVEAL process)
2. Formal Specification (using the formal notation Z)
3. **Formal Design and the INFORMED process**
4. Implementation in SPARK Ada
5. Verification (using the SPARK Examiner toolset).

2.1 Structure of this Design

This design is presented as a formal model of the TIS core function using concrete representations for the state. The model is presented using the Z notation. The structure of this design follows very closely the structure of the formal specification [4], from which it is refined.

The design models TIS as a number of state components and a number of operations that change the state. The operations presented in this design cover:

- user authentication and entry into the enclave;
- enrolment of TIS;
- administrator logon/logoff;
- archiving the log;
- updating of configuration data;
- shutdown;
- overriding the enclave door.

The design is structured by presenting type constructs useful in the modelling of TIS in the remainder of this section.

Section 3 introduces the refined state that defines the TIS.

Section 4 covers accepting data from the real world and updating the real world.

Section 5 presents a number of operations on parts of the TIS state, these are later used in the construction of the TIS system operations.

Section 6 presents the multi-phase user authentication and entry operation.

Section 7 describes all the system operations that take place within the enclave. These are administrative operations.

Section 8 defines the initial system and the state of TIS at start-up.

Section 9 describes how the whole TIS core works. Here we pull together the operations described through the remainder of the specification.

Appendix A discusses a number of issues that were raised during the production of this design.

Appendix B presents the refinement relation between the abstract state in the Formal Specification [4] and the state presented here.

Appendix C presents part of the refinement argument that the Formal Design is a correct refinement of the Formal Specification [4].

2.2 Design decisions

This section discusses the key design descisions that are addressed in this formal design.

2.2.1 Prioritisation

Within the formal specification there were a number of activities that could happen non-deterministically, in that the specification allowed a choice between two actions given the initial conditions.

Within the design we eliminate the non-determinism and thus define the priority of actions where there was an arbitrary choice in the Formal Specification.

A logged on administrator may tear their token during a user entry operation. Processing the Admin Token tear should take priority since information only presented to administrators will be displayed on the TIS Console screen until the token tear is processed.

With this in mind the assigned priority is as follows:

1. Progressing the initial system enrolment.
2. Handling an admin logout due to token tear or timeout.
3. Handling a user token tear.
4. Progressing any current user entry.
5. Handling any outstanding admin token tear (where the admin is not logged on).
6. Progressing any administrator activity.
7. Starting a user entry activity.
8. Starting an administrator logon or operation.

It should be noticed that constraints in the formal specification already prevent all administrator activities (apart from token tears) occuring concurrently with user entry processing.

The structure of the administrator operations has been altered slightly from the specification to assist in the implementation of this priorisation. Bad logouts due to a token tear during an operation are no longer presented as part of the operation, instead they are presented as part of administrator logout.

2.2.2 Clearing secure data

Data extracted from the tokens and held internally will be cleared when the token is removed or the system shutdown. This ensures that it is not possible to inadvertently transfer information from one user to another.

Fingerprint data is cleared from the fingerprint reader before and after data is read to ensure that no stale data is inadvertently read as valid.

2.2.3 Reading on demand

Within the Formal Specification it is assumed that all data is read from the real world on a periodic basis. In reality much of the data is time consuming to read so should only be read when required. Within the formal design we show which data should be polled frequently and which simply read when it is neaded. The design still falls short in this respect in the area of the Tokens, since within the implementation only sufficient data items will be read from a token to perform the validation, however this design shows all of the token being read. This is discussed futher in Appendix A.

2.2.4 Elaboration of Audit

A major refinement in the design is to define the structure and types of audit entries that will be logged. Also the definition of the audit log now models how this log should be implemented internally using a number of files.

2.2.5 Configuration Data

This design considers the configuration data in terms of simple parameters that can be supplied by the operator to define aspects such as authentication periods. This significantly restricts the possible system configurations as compared with the Formal Specification.

2.2.6 Encryption and Keys

Within the design the model of encryption and keys has been refined. However, since the core TIS will make use of libraries to supply the crypto functions, the formal model makes several assumptions about keys. The model simply aims to demonstrate the correct use of library utilities to perform the desired validation.

2.2.7 Certificates

The design model of certificates is refined to capture the concept that certificates take the form of raw data and a signature. The validity of the data is checked using the signature and various fields can be extracted from a certificate. This provides a more concrete model of how a certificate is formed and used than the specification provided. The mechanism by which extraction and construction of certificates is performed is not specified formally; this is because these facilities are provided by a certificate processing library that is considered outside of the core TIS function.

2.3 Trace units

Each section of the design has been tagged with a named *traceunit* which will be used as a reference from later design documents. All trace units in this document have the prefix "FD" identifying them as originating in the Formal Design.

Most traceunits contain a list of requirements that are relevent to that part of the specification. These are taken from the SRS [2].

2.4 Z basics

This formal design is written using the Z formal notation.

It provides a concrete implementation of the TIS system specified in the Formal Specification [4]. All state is refined to the concrete components that will be used in the implementation.

2.4.1 Z naming conventions

The convention used is to terminate each type, value and Schema name with the letter *C* where the entity is a direct refinement of an entity presented in the Formal Specification. So *AuditLogC* is the concrete version of the *AuditLog*. Similarly retrieval relations names have the letter *R* as a suffix, so *AuditLogR* is the retrieval relation between *AuditLogC* and *AuditLog*.

2.4.2 Z comments

The intention is that someone unfamiliar with Z should be able to read this specification and gain a complete understanding of the functionality of the TIS system specified within.

We have attempted to make the informal commentary as complete and unambiguous as possible. We have also separated out the parts of the commentary that are only relevant for understanding the formal model, as below:

▷ Readers who are not interested in the formal model can skip these sections of the commentary.

2.4.3 Z type checking

In order to make this document stand alone for reading purposes all definitions used unchanged from the specification are repeated in this document. Where this occurs the Z text is not type checked, the reason being that this document is type checked with the formal specification and the original declaration from the specification is used by the type checker. All Z statements repeated from the formal specification are annotated as such.

Section B defines the retrieval relations between the abstract state and the concrete state. This section makes reference to declaration in the formal specification [4] without representation.

2.5 TIS Basic Types

FD.Types.RawTypes

Within the TIS implmentation many entities are stored using Unsigned 32 bit integers.

$$\mathit{maxDigestLength} == 32$$
$$\mathit{maxSigLength} == 128$$

$$\mathit{BYTE} == 0 \mathinner{\ldotp\ldotp} 255$$

$$\mathit{INTEGER32} == -2147483648 \mathinner{\ldotp\ldotp} 2147483647$$

$$\mathit{RAWDATA} == \operatorname{seq} \mathit{BYTE}$$

$$\mathit{DIGESTDATA} == \{x : \mathit{RAWDATA} \mid \#x \leq \mathit{maxDigestLength}\}$$
$$\mathit{SIGDATA} == \{x : \mathit{RAWDATA} \mid \#x \leq \mathit{maxSigLength}\}$$

▷ See: *maxDigestLength* (p. 11), *maxSigLength* (p. 11)

FD.Types.Time
FS.Types.Time

Time and date is some universal clock, which for our purposes can be modelled as just the naturals. Our only requirement is that the granularity of our clock is sufficiently fine to distinguish times differing by a second, although to order audit entries effectively we choose 1/10 second as the unit of time.

$$\mathit{TIME} == \mathbb{N}$$

▷ Definition repeated from Formal Specification [4]

We define a constant *zeroTime* used at system initialisation.

$$\mathit{zeroTime} == 0$$

▷ Definition repeated from Formal Specification [4]

We introduce the concept of the length of a day. This is because some of the configuration data will relate to a single day.

$$\mathit{dayLength} : \mathit{TIME}$$

▷ See: *TIME* (p. 11)

$$\mathit{DAYTIME} == 0 \mathinner{\ldotp\ldotp} (\mathit{dayLength} - 1)$$

▷ See: *dayLength* (p. 11)

FD.Types.Presence
FS.Types.Presence

Many entities such as tokens, fingers and floppy disks may be presented to the system and removed by the user. We monitor the presence of these entities.

$$\mathit{PRESENCE} ::= \mathit{present} \mid \mathit{absent}$$

▷ Definitions repeated unchanged from Formal Specification [4]

FD.Types.Clearance
FS.Types.Clearance

CLASS is the ordered classifications on document, areas, and people.

$$CLASS ::= unmarked \mid unclassified \mid restricted \mid confidential \mid secret \mid topsecret$$

▷ Definition repeated unchanged from Formal Specification [4]

We define functions returning the minimum and maximum of a set of *CLASS*es.

$$\begin{array}{l} minClass : \mathbb{P}_1\ CLASS \longrightarrow CLASS \\ maxClass : \mathbb{P}_1\ CLASS \longrightarrow CLASS \\ \hline \exists\, ordering : \operatorname{seq} CLASS \bullet \\ \quad ordering = \langle unmarked, unclassified, restricted, confidential, secret, topsecret \rangle \\ \quad \land\ minClass = \{S : \mathbb{P}_1\ CLASS \bullet S \mapsto (ordering\ (min\ (\operatorname{dom}(ordering \rhd S))))\} \\ \quad \land\ maxClass = \{S : \mathbb{P}_1\ CLASS \bullet S \mapsto (ordering\ (max\ (\operatorname{dom}(ordering \rhd S))))\} \end{array}$$

▷ See: *CLASS* (p. 12), *unmarked* (p. 12), *unclassified* (p. 12), *restricted* (p. 12), *confidential* (p. 12), *secret* (p. 12), *topsecret* (p. 12)

$$\begin{array}{l} \underline{Clearance} \\ class : CLASS \\ \hline \end{array}$$

▷ See: *CLASS* (p. 12)

▷ Definition repeated unchanged from Formal Specification [4]

There is an ordering on the type *Clearance*. The function *minClearance* and *maxClearance* give the minimum and maximum of a pair of elements of type *Clearance*. This is defined in terms of the ordering on class.

$$\begin{array}{l} minClearance : Clearance \times Clearance \longrightarrow Clearance \\ maxClearance : Clearance \times Clearance \longrightarrow Clearance \\ \hline \forall\, a, b : Clearance \bullet \\ \quad minClearance(a, b).class = minClass\{a.class, b.class\} \end{array}$$

▷ See: *Clearance* (p. 12), *minClass* (p. 12)

▷ Declarations repeated unchanged from Formal Specification [4]. The definitions are new.

FD.Types.Privilege
FS.Types.Privilege

PRIVILEGE is the role held by the Token user. This will determine the privileges that the Token user has when interacting with the ID station.

$$PRIVILEGE ::= userOnly \mid guard \mid securityOfficer \mid auditManager$$

▷ Definition repeated unchanged from Formal Specification [4]

FD.Types.User
FD.Types.User

An *User* is a unique identification of an certificate owner. An user will have a common name which does not contribute to the unique identification.

$[USERID, USERNAME]$

```
┌─ User ──────────────────────────
│ id : USERID
│ name : USERNAME
└─────────────────────────────────
```

▷ See: *USERNAME* (p. 13)

FD.Types.Issuer
FS.Types.Issuer

An *Issuer* is a unique identification of an issuing body. Issuers are privileged users with the ability to issue certificates.

```
┌─ Issuer ────────────────────────
│ User
└─────────────────────────────────
```

▷ See: *User* (p. 13)

FD.Types.Fingerprint
FS.Types.Fingerprint

FINGERPRINT will need to include sufficient control information to allow us to compare with templates and decide a match or not.

$[FINGERPRINT]$

▷ Definition repeated unchanged from Formal Specification [4]

FD.Types.FingerprintTemplate
FS.Types.FingerprintTemplate

A *FINGERPRINTTEMPLATE* contains abstracted information, derived from a number of sample readings of a fingerprint.

$[FINGERPRINTTEMPLATE]$

▷ Definition repeated unchanged from Formal Specification [4]

The fingerprint template and will be accompanied by additional information, the FAR (False Acceptance Rate) threshold level to be applied to any comparisons.

A fingerprint template will need additional information, such as the False Acceptance Rate to be applied.

$$\begin{array}{|l}
\hline \textit{FingerprintTemplateC} \\
\textit{templateC} : \textit{FINGERPRINTTEMPLATE} \\
\textit{far} : \textit{INTEGER}32 \\
\hline
\end{array}$$

▷ See: *INTEGER*32 (p. 11)

The biometrics library provides facilities to tell whether a fingerprint read from a user matches a template.

$MATCHRESULT ::= match \mid noMatch$

$$\begin{array}{|l}
\textit{verifyBio} : \textit{INTEGER}32 \longrightarrow \textit{FINGERPRINTTEMPLATE} \longrightarrow \textit{FINGERPRINTTRY} \longrightarrow \textit{MATCHRESULT} \times \textit{INTEGER}32 \\
\hline
\forall \textit{maxFar} : \textit{INTEGER}32; \textit{fTemplate} : \textit{FINGERPRINTTEMPLATE}; \textit{finger} : \textit{FINGERPRINTTRY} \bullet \\
\quad \exists \textit{result} : \textit{MATCHRESULT}; \textit{achievedFar} : \textit{INTEGER}32 \bullet \\
\qquad \textit{verifyBio}\ \textit{maxFar}\ \textit{fTemplate}\ \textit{finger} = (\textit{result}, \textit{achievedFar}) \\
\qquad \land \textit{result} = \textit{match} \Rightarrow \textit{achievedFar} \leq \textit{maxFar}
\end{array}$$

▷ See: *INTEGER*32 (p. 11), *MATCHRESULT* (p. 14), *match* (p. 14)

2.6 Keys, Encryption and the Crypto Library

FD.KeyTypes.Keys
FS.KeyTypes.Keys

The signing and validation of certificates used in Tokeneer relies on the use of asymetric keys, which comprise two parts, one which is public and one which is private.

$[KEYPART]$

▷ Definition repeated unchanged from Formal Specification [4]

The core TIS makes use of a Crypto Library to maintain all the keys it knows about. This library maintains a database of the currently known keys.

A key part has a number of characteristics that aid identification of the key, in addition to the key data. It is either a *public* or *private* key and it has an owner, the issuer who holds the private part.

$KEYTYPE ::= public \mid private$

$$\begin{array}{|l}
\hline \textit{KeyPart} \\
\textit{keyType} : \textit{KEYTYPE} \\
\textit{keyOwner} : \textit{Issuer} \\
\textit{keyData} : \textit{KEYPART} \\
\hline
\end{array}$$

▷ See: *KEYTYPE* (p. 14), *Issuer* (p. 13)

Certificates are signed by an issuer using the private part, and can be verified by anyone who holds the public part.

The Crypto Library also provides utility functions that allow digests to be created and signatures to be created and verified. These functions support a number of digest algorithms and asymmetric signing alorithms.

$$ALGORITHM ::= rsa \mid md2 \mid md5 \mid sha1 \mid ripemd128 \mid ripemd160 \mid rsaWithMd2 \mid rsaWithMd5 \mid rsaWithSha1 \mid rsaWithRipemd128 \mid rsaWithRipemd160$$

$$\begin{array}{|l}
digest : ALGORITHM \pfun RAWDATA \longrightarrow DIGESTDATA \\
sign : ALGORITHM \pfun KEYPART \longrightarrow DIGESTDATA \longrightarrow SIGDATA \\
_ \ \mathsf{isVerifiedBy} \ _ : (ALGORITHM \times DIGESTDATA \times SIGDATA) \leftrightarrow KEYPART \\
\hline
\forall privateKey, publicKey : KeyPart; data : RAWDATA; mechanism : ALGORITHM; \\
\quad theDigest : DIGESTDATA; theSignature : SIGDATA \bullet \\
\qquad mechanism \in \mathrm{dom}\, digest \cap \mathrm{dom}\, sign \\
\qquad \land privateKey.keyType = private \land publicKey.keyType = public \\
\qquad \land publicKey.keyOwner = privateKey.keyOwner \\
\qquad \land theDigest = digest\, mechanism\, data \\
\qquad \land theSignature = sign\, mechanism\, privateKey.keyData\, theDigest \Leftrightarrow \\
\qquad\quad (mechanism, theDigest, theSignature)\ \mathsf{isVerifiedBy}\ publicKey.keyData
\end{array}$$

▷ See: *ALGORITHM* (p. 15), *RAWDATA* (p. 11), *DIGESTDATA* (p. 11), *SIGDATA* (p. 11), *KeyPart* (p. 14), *private* (p. 14), *public* (p. 14)

Knowing an issuer is equivalent to having a copy of the issuer's public key part. While possessing an issuer's private key part means that you are that issuer.

2.7 Certificates, Tokens and Enrolment Data

2.7.1 Certificates

FD.Types.Certificates
FS.Types.Certificates

All certificates consist of data and a signature. A number of attributes are encoded within the data.

$$\begin{array}{|l}
\multicolumn{1}{l}{RawCertificate} \\
data : RAWDATA \\
signature : SIGDATA \\
signedData : RAWDATA \\
\hline
signedData = data \frown signature
\end{array}$$

▷ See: *RAWDATA* (p. 11), *SIGDATA* (p. 11)

Each certificate is signed and can be verified using a key, typically the public key of an issuer.

Some attributes are common to all certificates.

All certificates can be uniquely identified by their issuer and the serial Number supplied by the issuer when the certificate is created.

$$\begin{array}{|l}
\hline \textit{CertificateIdC} \\
\textit{issuerC} : \textit{Issuer} \\
\textit{serialNumber} : \mathbb{N} \\
\hline
\end{array}$$

▷ See: *Issuer* (p. 13)

In addition to the unique certificate id all certificates contain a validity period during which time they are valid. This is defined by two times *notBefore* and *notAfter*. The validity period is any time t satisfying $notBefore \leq t$ and $t \leq notAfter$. The mechanism is the althorithm by which the signature is signed.

$$\begin{array}{|l}
\hline \textit{CertificateContents} \\
\textit{idC} : \textit{CertificateIdC} \\
\textit{notBefore} : \textit{TIME} \\
\textit{notAfter} : \textit{TIME} \\
\textit{mechanism} : \textit{ALGORITHM} \\
\hline
\end{array}$$

▷ See: *CertificateIdC* (p. 16), *TIME* (p. 11), *ALGORITHM* (p. 15)

$$\begin{array}{|l}
\textit{certificateValidity} : \textit{CertificateContents} \longrightarrow \mathbb{P}\, \textit{TIME} \\
\hline
\textit{certificateValidity} = (\lambda\, \textit{CertificateContents} \bullet \textit{notBefore} \mathrel{..} \textit{notAfter})
\end{array}$$

▷ See: *CertificateContents* (p. 16), *TIME* (p. 11)

Each type of certificate potentially expands on these attributes.

The ID certificate is an X.509 certificate. ID certificates are used during enrolment as well as being present on tokens.

The subject is the name of the entity being identified by the certificate and the key is the entity's public key.

We don't need to know about the key of the Token although we do need to know about the key of an issuer supplied at enrolment.

$$\begin{array}{|l}
\hline \textit{IDCertContents} \\
\textit{CertificateContents} \\
\textit{subjectC} : \textit{Issuer} \\
\textit{subjectPubKC} : \textit{KEYPART} \\
\hline
\end{array}$$

▷ See: *CertificateContents* (p. 16), *Issuer* (p. 13)

The certificates containing attributes all share some common attributes.

An attribute certificate contains the identification of the ID certificate to which it relates, this ID certificate is referred to as the base certificate for the attribute certificate. The base certificate should be the ID certificate on the Token.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AttCertificateContents}} \\ \hline
CertificateContents \\
baseCertIdC : CertificateIdC \\ \hline
\end{array}$$

▷ See: *CertificateContents* (p. 16), *CertificateIdC* (p. 16)

A privilege certificate additionally contains a role and clearance.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{PrivCertContents}} \\ \hline
AttCertificateContents \\
roleC : PRIVILEGE \\
clearanceC : Clearance \\ \hline
\end{array}$$

▷ See: *AttCertificateContents* (p. 16), *PRIVILEGE* (p. 12), *Clearance* (p. 12)

An authorisation certificate has the same structure as a privilege certificate.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AuthCertContents}} \\ \hline
AttCertificateContents \\
roleC : PRIVILEGE \\
clearanceC : Clearance \\ \hline
\end{array}$$

▷ See: *AttCertificateContents* (p. 16), *PRIVILEGE* (p. 12), *Clearance* (p. 12)

An I&A (Identification and Authentication) certificate additionally contains a fingerprint template.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{IandACertContents}} \\ \hline
AttCertificateContents \\
templateC : FingerprintTemplateC \\ \hline
\end{array}$$

▷ See: *AttCertificateContents* (p. 16), *FingerprintTemplateC* (p. 14)

All certificates can be extracted from raw certificate data. The extraction functions are provided by a Certificate Processing Library, which is outside the scope of the core TIS, but will be utilised by TIS.

The certificate processing library also provides a function to generate the raw certificate data from an authorisation certificate contents.

$$\begin{array}{|l}
extractIDCert : RawCertificate \rightarrowtail\!\!\!\!\!+ IDCertContents \\
extractPrivCert : RawCertificate \rightarrowtail\!\!\!\!\!+ PrivCertContents \\
extractIandACert : RawCertificate \rightarrowtail\!\!\!\!\!+ IandACertContents \\
extractAuthCert : RawCertificate \rightarrowtail\!\!\!\!\!+ AuthCertContents \\
constructAuthCert : AuthCertContents \longrightarrow RAWDATA
\end{array}$$

▷ See: *RawCertificate* (p. 15), *IDCertContents* (p. 16), *PrivCertContents* (p. 17), *IandACertContents* (p. 17), *AuthCertContents* (p. 17), *RAWDATA* (p. 11)

Each type of certificate comprises a *RawCertficate*, from which the required certificate type can be extracted.

We can extract the contents of an ID certificate from an ID certificate.

$$IDCertC \mathrel{\widehat=} [\, RawCertificate \mid \theta RawCertificate \in \operatorname{dom} extractIDCert \,]$$

▷ See: *RawCertificate* (p. 15), *extractIDCert* (p. 17)

In general an ID certificate is not validated by the keypart held on the certificate.

The ID Certificate of a CA (Certification Authority) is a root certificate and is signed by itself.

CAIdCertC

$$IDCertC$$

$$\begin{array}{l} \exists\, IDCertContents;\ theDigest : DIGESTDATA \bullet \\ \quad \theta IDCertContents = extractIDCert\,\theta RawCertificate \\ \quad \land\ theDigest = digest\ mechanism\ data \\ \quad \land\ (mechanism, theDigest, signature)\ \mathsf{isVerifiedBy}\ subjectPubKC \end{array}$$

▷ See: *IDCertC* (p. 18), *IDCertContents* (p. 16), *DIGESTDATA* (p. 11), *extractIDCert* (p. 17), *RawCertificate* (p. 15), *digest* (p. 15)

We can extract the contents of a privilege certificate from a Priv Certificate.

$$PrivCertC \mathrel{\widehat=} [\, RawCertificate \mid \theta RawCertificate \in \operatorname{dom} extractPrivCert \,]$$

▷ See: *RawCertificate* (p. 15)

We can extract the contents of an I&A certificate from an I&A Certificate.

$$IandACertC \mathrel{\widehat=} [\, RawCertificate \mid \theta RawCertificate \in \operatorname{dom} extractIandACert \,]$$

▷ See: *RawCertificate* (p. 15)

We can extract the contents of an authorisation certificate from an Auth Certificate.

$$AuthCertC \mathrel{\widehat=} [\, RawCertificate \mid \theta RawCertificate \in \operatorname{dom} extractAuthCert \,]$$

▷ See: *RawCertificate* (p. 15)

2.7.2 Tokens

FD.Types.Tokens
FS.Types.Tokens

Each Token has an ID. Token IDs are different for every token from a given smartcard supplier (issuer). Tokens from different issuers may, however, share Token IDs.

$$TOKENIDC == \mathbb{N}$$

A *Token* contains a number of certificates. The authorisation certificate is optional while the others must be present.

$$
\begin{array}{|l}
\textit{TokenC} \\
\hline
\textit{tokenIDC} : \textit{TOKENIDC} \\[1ex]
\textit{idCertC} : \textit{RawCertificate} \\
\textit{privCertC} : \textit{RawCertificate} \\
\textit{iandACertC} : \textit{RawCertificate} \\
\textit{authCertC} : \mathrm{optional}\ \textit{RawCertificate} \\
\hline
\end{array}
$$

▷ See: *TOKENIDC* (p. 18), *RawCertificate* (p. 15)

A *Token* is valid if all of the certificates on it are well-formed, each certificate correctly cross-references to the ID Certificate, and the ID Certificate correctly cross-references to the *Token* ID.

A token need not contain a valid Auth certificate to be considered valid.

$$
\begin{array}{|l}
\textit{ValidTokenC} \\
\hline
\textit{TokenC} \\
\hline
\textit{idCertC} \in \{\textit{IDCertC}\} \\
\textit{privCertC} \in \{\textit{PrivCertC}\} \\
\textit{iandACertC} \in \{\textit{IandACertC}\} \\[1ex]
(\textit{extractPrivCert}\ \textit{privCertC}).\textit{baseCertIdC} = (\textit{extractIDCert}\ \textit{idCertC}).\textit{idC} \\
(\textit{extractIandACert}\ \textit{iandACertC}).\textit{baseCertIdC} = (\textit{extractIDCert}\ \textit{idCertC}).\textit{idC} \\[1ex]
(\textit{extractIDCert}\ \textit{idCertC}).\textit{idC}.\textit{serialNumber} = \textit{tokenIDC} \\
\hline
\end{array}
$$

▷ See: *TokenC* (p. 19), *IDCertC* (p. 18), *PrivCertC* (p. 18), *IandACertC* (p. 18), *extractIDCert* (p. 17)

If the Auth certificate is present it will only be used if it is valid, in that it correctly cross-references to the *Token* ID and the ID certificate.

$$
\begin{array}{|l}
\textit{TokenWithValidAuthC} \\
\hline
\textit{TokenC} \\
\hline
\textit{idCertC} \in \{\textit{IDCertC}\} \\[1ex]
(\textit{extractIDCert}\ \textit{idCertC}).\textit{idC}.\textit{serialNumber} = \textit{tokenIDC} \\[1ex]
\textit{authCertC} \neq \textit{nil} \\
\quad \land\ \textit{the}\ \textit{authCertC} \in \{\textit{AuthCertC}\} \\
\quad \land\ (\textit{extractAuthCert}\ (\textit{the}\ \textit{authCertC})).\textit{baseCertIdC}.\textit{serialNumber} = \textit{tokenIDC} \\
\quad \land\ (\textit{extractAuthCert}\ (\textit{the}\ \textit{authCertC})).\textit{baseCertIdC} = (\textit{extractIDCert}\ \textit{idCertC}).\textit{idC} \\
\hline
\end{array}
$$

▷ See: *TokenC* (p. 19), *IDCertC* (p. 18), *extractIDCert* (p. 17), *AuthCertC* (p. 18)

A *Token* is current if all of the Certificates are current, or if only the Auth Cert is non-current. Currency needs a time, which is included in the schema, and will need to be tied to the relevent time when this schema is used.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad CurrentTokenC \quad}} \\
ValidTokenC \\
nowC : TIME \\
\hline
(\exists IDCertContents \bullet \theta IDCertContents = extractIDCert\ idCertC \\
\quad \land nowC \in certificateValidity\ \theta CertificateContents) \\
(\exists PrivCertContents \bullet \theta PrivCertContents = extractPrivCert\ idCertC \\
\quad \land nowC \in certificateValidity\ \theta CertificateContents) \\
(\exists IandACertContents \bullet \theta IandACertContents = extractIandACert\ idCertC \\
\quad \land nowC \in certificateValidity\ \theta CertificateContents) \\
\hline
\end{array}
$$

▷ See: *ValidTokenC* (p. 19), *TIME* (p. 11), *IDCertContents* (p. 16), *extractIDCert* (p. 17), *certificateValidity* (p. 16), *CertificateContents* (p. 16), *PrivCertContents* (p. 17), *IandACertContents* (p. 17)

2.7.3 Enrolment Data

FD.Types.Enrolment
FS.Types.Enrolment

Enrolment data is the information the ID station needs in order to know how to authenticate tokens presented to it, and to produce its own authentication certificates such that they can be authenticated by workstations in the enclave.

Enrolment data consists of a number of ID certificates:

- this ID Station's ID Certificate, which will be signed by a CA.
- A number of other Issuers' ID Certificates. These will belong to
  - CAs, who authenticate AAs (Attribute Authorities) and ID Stations. These will be self signed.
  - AAs, who authenticate privilege and I&A certificates.

The ID Station's certificate is just one of the issuer certificates, although we will want to be able to identify it as belonging to this ID station.

The certificates are ordered within the enrolment data.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad EnrolC \quad}} \\
idStationCertC : IDCertC \\
issuerCertsC : \mathrm{iseq}\ IDCertC \\
\hline
idStationCertC \in \mathrm{ran}\ issuerCertsC \\
\hline
\end{array}
$$

▷ See: *IDCertC* (p. 18)

For the Enrolment data to be considered valid each certificate must be signed correctly and the Issuer's certificate must be present for it to be possible to check that the signatures are correct. Note that CA ID certificates are self signed but AA and IDStation certificates are signed by an CA.

For each certificate that is not self signed the signing CA will appear earlier in the sequence of issuers.

The ID station certificate is the second certificate in the enrolment data and must be preceded by the certificate for the issuing CA.

$$
\begin{array}{l}
\textit{ValidEnrolC} \\
\hline
\textit{EnrolC} \\
\hline
\operatorname{ran} \textit{issuerCertsC} \cap \{\textit{CAIdCertC}\} \neq \varnothing \\[4pt]
\forall \textit{cert} : \textit{IDCertC}; \textit{earlierCerts} : \operatorname{seq} \textit{IDCertC} \mid \textit{earlierCerts} \frown \langle \textit{cert} \rangle \mathsf{\ prefix\ } \textit{issuerCertsC} \bullet \\
\quad \exists \textit{issuerCert} : \textit{IDCertC}; \\
\quad \textit{certContent}, \textit{issuerContent} : \textit{IDCertContents}; \textit{theDigest} : \textit{DIGESTDATA} \bullet \\
\qquad \textit{issuerCert} \in \operatorname{ran}(\textit{earlierCerts} \frown \langle \textit{cert} \rangle) \\
\qquad \land \textit{certContent} = \textit{extractIDCert}\ \textit{cert} \land \textit{issuerContent} = \textit{extractIDCert}\ \textit{issuerCert} \\
\qquad \land \textit{certContent}.\textit{idC}.\textit{issuerC} = \textit{issuerContent}.\textit{subjectC} \\
\qquad \land \textit{theDigest} = \textit{digest}\ \textit{certContent}.\textit{mechanism}\ \textit{cert}.\textit{data} \\
\qquad \land (\textit{certContent}.\textit{mechanism}, \textit{theDigest}, \textit{cert}.\textit{signature}) \mathsf{\ isVerifiedBy\ } \textit{issuerContent}.\textit{subjectPubKC} \\[4pt]
\textit{issuerCertsC}^{\sim} \textit{idStationCertC} = 2
\end{array}
$$

▷ See: *EnrolC* (p. 20), *CAIdCertC* (p. 18), *IDCertC* (p. 18), *IDCertContents* (p. 16), *DIGESTDATA* (p. 11), *extractIDCert* (p. 17), *digest* (p. 15)

▷ There must be an ID certificate for at least one CA.

▷ For each certificate the enrolment data must include the ID certificate for the issuer of the certificate, the certificate must be validated by the issuer's key and the issuer of the certificate must be a CA.

▷ For each certificate the ID certificate of the issuer of the certificate must apear earlier in the enrolment data.

▷ The certificate for the ID Station is the second certificate.

2.8 World outside the ID Station

We choose to model the real world (or at least the real peripherals) as being outside the ID Station. When the user inserts a token, they are providing input to the ID Station. It is up to the ID Station to then respond by reading the real world input into its own, internal representation. The ID Station receives stimulus from the real world and itself changes the real world. All real world entities are modelled as components of the *RealWorld*.

We will distingush between real world entities that we use (eg. *finger*), we control (eg. *alarm*) and we may change (eg. *userToken* or *floppy*).

2.8.1 Real World types

FD.Types.RealWorld
FS.Types.RealWorld

There are several types associated with the real world. The door, latch and alarm all have two possible states.

$$\textit{DOOR} ::= \textit{open} \mid \textit{closed}$$

$$\textit{LATCH} ::= \textit{unlocked} \mid \textit{locked}$$

$$\textit{ALARM} ::= \textit{silent} \mid \textit{alarming}$$

▷ Definitions repeated unchanged from Formal Specification [4].

Display messages are the short messages presented to the user on the small display outside the enclave.

$$\mathit{DISPLAYMESSAGE} ::= \mathit{blank} \mid \mathit{welcome} \mid \mathit{insertFinger} \mid \mathit{openDoor} \mid \mathit{wait} \mid \\ \mathit{removeToken} \mid \mathit{tokenUpdateFailed} \mid \mathit{doorUnlocked}$$

▷ Definitions repeated unchanged from Formal Specification [4].

The messages that appear on the display are presented in the table 2.1.

	Displayed text	
Message	**Top line**	**Bottom line**
blank	SYSTEM NOT OPERATIONAL	
welcome	WELCOME TO TIS	ENTER TOKEN
insertFinger	AUTHENTICATING USER	INSERT FINGER
wait	AUTHENTICATING USER	PLEASE WAIT
openDoor		REMOVE TOKEN AND ENTER
removeToken	ENTRY DENIED	REMOVE TOKEN
tokenUpdateFailed		TOKEN UPDATE FAILED
doorUnlocked		ENTER ENCLAVE

Table 2.1: Display Messages

Because it is possible to be trying to read a token that is not inserted, or a fingerprint when no finger is inserted, or an invalid token or fingerprint, we introduce free types to capture the absence or poor quality of these.

The values *badFP* and *badT* represent all possible error codes that occur when trying to capture this data. The system will behave the same way in all failure cases with only the audit log capturing the different error codes that actually occur.

$$\mathit{FINGERPRINTTRY} ::= \mathit{noFP} \mid \mathit{badFP} \mid \mathit{goodFP}\langle\!\langle \mathit{FINGERPRINT} \rangle\!\rangle$$

▷ Definition repeated unchanged from Formal Specification [4].

$$\mathit{TOKENTRYC} ::= \mathit{noTC} \mid \mathit{badTC} \mid \mathit{goodTC}\langle\!\langle \mathit{TokenC} \rangle\!\rangle$$

▷ See: *TokenC* (p. 19)

When modelling data supplied on a floppy disk we model the possibility of the disk not being present, being empty or being corrupt as well as containing valid data. We make the assumption that each floppy disk will only contain one data type, either enrolment data, configuration data or audit data.

$$\mathit{FLOPPYC} ::= \mathit{noFloppyC} \mid \mathit{emptyFloppyC} \mid \mathit{badFloppyC} \mid \mathit{enrolmentFileC}\langle\!\langle \mathit{EnrolC} \rangle\!\rangle \mid \\ \mathit{auditFileC}\langle\!\langle \mathbb{F}\, \mathit{AuditC} \rangle\!\rangle \mid \mathit{configFileC}\langle\!\langle \mathit{ConfigC} \rangle\!\rangle$$

▷ See: *EnrolC* (p. 20)

Inputs may be supplied by an administrator at the keyboard. We model input values representing no data, invalid data or a valid request to perform an adminstrator operation.

$$KEYBOARD ::= noKB \mid badKB \mid keyedOps\langle\langle ADMINOP \rangle\rangle$$

▷ Definitions repeated unchanged from Formal Specification [4].

There are a number of messages that may appear on the TIS screen within the enclave. Some of these are simple messages, the text of these is supplied in the Table 2.2. Others involve more complex presentation of data, such as configuration data or system statistics, the details of this presentation is left to design.

$$\begin{aligned} SCREENTEXTC ::= {} & clearC \mid welcomeAdminC \mid busyC \mid removeAdminTokenC \mid closeDoorC \mid \\ & requestAdminOpC \mid doingOpC \mid invalidRequestC \mid invalidDataC \mid \\ & insertEnrolmentDataC \mid validatingEnrolmentDataC \mid enrolmentFailedC \mid \\ & archiveFailedC \mid insertBlankFloppyC \mid insertConfigDataC \mid \\ & displayStatsC\langle\langle StatsC \rangle\rangle \mid displayConfigDataC\langle\langle ConfigC \rangle\rangle \mid \\ & displayAlarm\langle\langle ALARM \rangle\rangle \end{aligned}$$

▷ See: *ALARM* (p. 21)

Message	**Displayed text**
clearC	
welcomeAdminC	`WELCOME TO TIS`
busyC	`SYSTEM BUSY PLEASE WAIT`
removeAdminTokenC	`REMOVE TOKEN`
closeDoorC	`CLOSE ENCLAVE DOOR`
requestAdminOpC	`ENTER REQUIRED OPERATION`
doingOpC	`PERFORMING OPERATION PLEASE WAIT`
invalidRequestC	`INVALID REQUEST: PLEASE ENTER NEW OPERATION`
invalidDataC	`INVALID DATA: PLEASE ENTER NEW OPERATION`
archiveFailedC	`ARCHIVE FAILED: PLEASE ENTER NEW OPERATION`
insertEnrolmentDataC	`PLEASE INSERT ENROLMENT DATA FLOPPY`
validatingEnrolmentDataC	`VALIDATING ENROLMENT DATA PLEASE WAIT`
enrolmentFailedC	`INVALID ENROLMENT DATA`
insertBlankFloppyC	`INSERT BLANK FLOPPY`
insertConfigDataC	`INSERT CONFIGURATION DATA FLOPPY`

Table 2.2: Short Screen Messages

2.8.2 The Real World

FD.ControlledRealWorld.State

The real world entities that are controlled by TIS are as follows:

- the latch on the door into the enclave.
- the audible alarm.

- the display that resides outside the enclave.
- the screen on the ID Station within the enclave with which the administrator interacts.

TISControlledRealWorldC

latchC : *LATCH*
alarmC : *ALARM*
displayC : *DISPLAYMESSAGE*
screenC : *ScreenC*

▷ See: *LATCH* (p. 21), *ALARM* (p. 21), *DISPLAYMESSAGE* (p. 22)

FD.MonitoredRealWorld.State

The real world entities that are used by TIS are as follows:

- the real world has a concept of time.
- the door into the enclave that is monitored by the ID Station.
- fingerprints are read, via the biometric reader, into the ID Station for comparison with fingerprint templates.
- a user, trying to enter the enclave will supply their token to the ID station via the token reader that resides outside the enclave.
- a user within the enclave who has administrator privileges will supply their token to the ID station via the token reader that resides inside the enclave.
- the ID Station accepts enrolment data and configuration data on a floppy disk. The disk drive resides in the enclave.
- the ID Station has a keyboard within the enclave which the administrator uses to control TIS.

TISMonitoredRealWorldC

nowC : *TIME*
doorC : *DOOR*
fingerC : *FINGERPRINTTRY*
userTokenC, *adminTokenC* : *TOKENTRYC*
floppyC : *FLOPPYC*
keyboardC : *KEYBOARD*

▷ See: *TIME* (p. 11), *DOOR* (p. 21), *FINGERPRINTTRY* (p. 22), *TOKENTRYC* (p. 22), *FLOPPYC* (p. 22), *KEYBOARD* (p. 23)

In addition TIS may change some of the entities that it uses from the real world.

- The ID station may need to update the *userToken* token (with an Authentication Certificate).
- The ID Station archives the Audit Log to floppy disk so may write to *floppy*.
- The ID Station flushes fingerprint information from the biometric reader after validating the data.

The Whole real world is given by:

$$RealWorldC \mathrel{\widehat{=}} TISControlledRealWorldC \land TISMonitoredRealWorldC$$

▷ See: *TISControlledRealWorldC* (p. 24), *TISMonitoredRealWorldC* (p. 24)

3 THE TOKEN ID STATION

TIS maintains various state components, these are described and elaborated within this section.

3.1 Configuration Data

FD.ConfigData.State
FS.ConfigData.State

ConfigData will be a structure with all the configuration data. Configuration data can only be modified by an administrator. This data includes:

- Durations for internal timeouts, these effect
  - how long the system waits before raising an audible (door) alarm;
  - how long the system leaves the door unlocked for;
  - how long the system waits for a token to be removed before unloading the door; and
  - how long the system attempts to capture a matching fingerprint.
- The security classification of the enclave. For this implementation only the *CLASS* is considered.
- A definition of the current working hours, this is in terms of the start and end of the working day. All days are considered working days, so there is no special treatment of weekends.
- A definition of the current maximum authorisation period applied to an authorisation certificate if "all hours" access is given.
- The access policy used to determine the entry conditions and the authorisation period.
  - The access policy is either "working hours only" or "all hours".
  - When the access policy is "working hours only" the authorisation period will be from the current time to the end of the current working day. This may be empty if the current time is after the end of the working day. The user will only be admitted to the enclave if the current time is within working hours.
  - When the access policy is "all hours" the authorisation period will be from the current time for the maximum authorisation duration. The user will always be allowed into the enclave if all identification checks are satisified.
- The lowest security classification a user must hold to gain entry to the enclave. If this condition is not met then entry will be denied.
- *minPreservedLogSizeC* gives the minimum size of audit log that must be supported without truncation. A slightly smaller value, *alarmThresholdSizeC*, sets the number of audit entries at which an alarm is raised, with the intension that the audit log will be archived and cleared before the maximum size is reached.
- *minEntryClass* must be no higher class than *enclaveClearanceC*. This ensures that any authorisation certificate issued with this configuration data will also permit entry.
- *systemMaxFAR* gives the system minimum acceptable false accept rate. This will override the FAR provided within a template where the template FAR exceeds this system limit.

$ACCESS\_POLICY ::= workingHours \mid allHours$

ConfigData

$alarmSilentDurationC, latchUnlockDurationC : TIME$
$tokenRemovalDurationC : TIME$
$fingerWaitDuration : TIME$
$enclaveClearanceC : CLASS$

$workingHoursStart : DAYTIME$
$workingHoursEnd : DAYTIME$
$maxAuthDuration : DAYTIME$
$accessPolicy : ACCESS\_POLICY$
$minEntryClass : CLASS$

$minPreservedLogSizeC : \mathbb{N}$
$alarmThresholdSizeC : \mathbb{N}$

$systemMaxFar : INTEGER32$

$alarmThresholdSizeC < minPreservedLogSizeC$
$minPreservedLogSizeC \leq maxSupportedLogSize$
$minEntryClass = minClass\{minEntryClass, enclaveClearanceC\}$

▷ See: *TIME* (p. 11), *CLASS* (p. 12), *DAYTIME* (p. 11), *ACCESS_POLICY* (p. 27), *INTEGER*32 (p. 11)

▷ The upper bound on the *minPreservedLogSizeC* ensures that the system can support the selected value for this.

Notice that the concrete configuration data is simplified so that authorisation periods and entry criteria do not depend on the user's privilege. This is a design decision to simplify these.

The authorisation period is always a contiguous range of times. This is necessary due to the way that the authorisation period is encoded in the authorisation certificate.

The entry period is the same for each day.

ConfigData defines the data that must be provided in order to perform a configuration. *ConfigC* contains extra components which are derived from *ConfigData*.

$$
\begin{array}{l}
\textit{ConfigC} \\
\hline
\textit{ConfigData} \\
\\
\textit{authPeriodC} : \textit{TIME} \longrightarrow \mathbb{P}\,\textit{TIME} \\
\textit{entryPeriodC} : \textit{CLASS} \longrightarrow \mathbb{P}\,\textit{TIME} \\
\textit{authPeriodIsEmpty} : \mathbb{P}\,\textit{TIME} \\
\textit{getAuthPeriod} : \textit{TIME} \nrightarrow \textit{TIME} \times \textit{TIME} \\
\textit{alarmThresholdEntries} : \mathbb{N} \\
\hline
\textit{accessPolicy} = \textit{allHours} \\
\quad \land\ \textit{authPeriodC} = \{t : \textit{TIME} \bullet t \mapsto t \mathrel{..} \textit{max}\,\{0, t + \textit{maxAuthDuration} - 1\}\} \\
\quad \land\ \textit{entryPeriodC} = \{c : \textit{CLASS} \mid \textit{maxClass}\{c, \textit{minEntryClass}\} = c \bullet c \mapsto \textit{TIME}\} \\
\qquad \cup \{c : \textit{CLASS} \mid \textit{maxClass}\{c, \textit{minEntryClass}\} \neq c \bullet c \mapsto \varnothing\} \\
\lor \\
\textit{accessPolicy} = \textit{workingHours} \\
\quad \land\ \textit{authPeriodC} = \{t : \textit{TIME} \bullet t \mapsto (t \operatorname{div} \textit{dayLength}) * \textit{dayLength} + \textit{workingHoursStart} \mathrel{..} \\
\qquad\qquad (t \operatorname{div} \textit{dayLength}) * \textit{dayLength} + \textit{workingHoursEnd}\} \\
\quad \land\ \textit{entryPeriodC} = \\
\qquad \{c : \textit{CLASS} \mid \textit{maxClass}\{c, \textit{minEntryClass}\} = c \\
\qquad\quad \bullet c \mapsto \{t : \textit{TIME} \mid t \bmod \textit{dayLength} \in \textit{workingHoursStart} \mathrel{..} \textit{workingHoursEnd}\}\} \\
\qquad \cup \{c : \textit{CLASS} \mid \textit{maxClass}\{c, \textit{minEntryClass}\} \neq c \bullet c \mapsto \varnothing\} \\
\\
\textit{authPeriodIsEmpty} = \{t : \textit{TIME} \mid \textit{authPeriodC}\ t = \varnothing\} \\
\textit{getAuthPeriod} = \{t : \textit{TIME} \mid \textit{authPeriodC}\ t \neq \varnothing \bullet t \mapsto (\textit{min}\,(\textit{authPeriodC}\ t), \textit{max}\,(\textit{authPeriodC}\ t))\} \\
\\
(\textit{alarmThresholdEntries} - 1) * \textit{sizeAuditElement} < \textit{alarmThresholdSizeC} \\
\textit{alarmThresholdEntries} * \textit{sizeAuditElement} \geq \textit{alarmThresholdSizeC} \\
\hline
\end{array}
$$

▷ See: *ConfigData* (p. 27), *TIME* (p. 11), *CLASS* (p. 12), *allHours* (p. 27), *workingHours* (p. 27), *dayLength* (p. 11)

▷ Invarients on *authPeriodC* and *entryPeriodC* define these functions in terms of the other configuration items. These values will not be supplied as part of configuration data.

▷ Invarients on *alarmThresholdEntries* define this values in terms of other configuration items. *alarmThresholdEntries* is the number of elements in the log after which the audit alarm will be raised.

▷ *getAuthPeriod* and *authPeriodIsEmpty* are completely determined by invarients relating these entities to other configuration items.

3.2 Audit Log

FD.AuditLog.State	
FS.AuditLog.State	*FAU_GEN.1.2*
FAU_GEN.1.1	

TIS maintains an audit log. This is a log of all auditable events and actions performed or monitored by TIS. The audit log will be used to analyse the interactions with the TIS.

Audit will be a structure for each audit record, recording at least time of event, type of event, user if known, the user is identified from the ID Certificate on the token and a free text description. The free text may contain additional information relating to the specific type of event.

$$
\begin{array}{l}
\mathit{AUDIT\_ELEMENT} ::= \\
\quad \mathit{startUnenrolledTISElement} \mid \mathit{startEnrolledTISElement} \mid \mathit{enrolmentCompleteElement} \mid \mathit{enrolmentFailedElement} \\
\quad \mid \mathit{displayChangedElement} \mid \mathit{screenChangedElement} \mid \mathit{doorClosedElement} \mid \mathit{doorOpenedElement} \\
\quad \mid \mathit{latchLockedElement} \mid \mathit{latchUnlockedElement} \mid \mathit{alarmRaisedElement} \mid \mathit{alarmSilencedElement} \\
\quad \mid \mathit{truncateLogElement} \mid \mathit{auditAlarmRaisedElement} \mid \mathit{auditAlarmSilencedElement} \\
\quad \mid \mathit{userTokenRemovedElement} \mid \mathit{userTokenPresentElement} \mid \mathit{userTokenInvalidElement} \\
\quad \mid \mathit{authCertValidElement} \mid \mathit{authCertInvalidElement} \\
\quad \mid \mathit{fingerDetectedElement} \mid \mathit{fingerTimeoutElement} \mid \mathit{fingerMatchedElement} \mid \mathit{fingerNotMatchedElement} \\
\quad \mid \mathit{authCertWrittenElement} \mid \mathit{authCertWriteFailedElement} \\
\quad \mid \mathit{entryPermittedElement} \mid \mathit{entryTimeoutElement} \mid \mathit{entryDeniedElement} \\
\quad \mid \mathit{adminTokenPresentElement} \mid \mathit{adminTokenValidElement} \mid \mathit{adminTokenInvalidElement} \\
\quad \mid \mathit{adminTokenExpiredElement} \mid \mathit{adminTokenRemovedElement} \\
\quad \mid \mathit{invalidOpRequestElement} \mid \mathit{operationStartElement} \\
\quad \mid \mathit{archiveLogElement} \mid \mathit{archiveCompleteElement} \mid \mathit{archiveCheckFailedElement} \mid \mathit{updatedConfigDataElement} \\
\quad \mid \mathit{invalidConfigDataElement} \mid \mathit{shutdownElement} \mid \mathit{overrideLockElement} \mid \mathit{systemFaultElement}
\end{array}
$$

$$\mathit{AUDIT\_SEVERITY} ::= \mathit{information} \mid \mathit{warning} \mid \mathit{critical}$$

$$
\begin{array}{l}
\mathit{USER\_INDEPENDENT\_ELEMENTS} == \{\mathit{systemFaultElement}, \mathit{displayChangedElement}, \mathit{screenChangedElement}, \\
\quad \mathit{doorClosedElement}, \mathit{doorOpenedElement}, \mathit{latchLockedElement}, \mathit{latchUnlockedElement}, \\
\quad \mathit{alarmRaisedElement}, \mathit{alarmSilencedElement}, \mathit{auditAlarmRaisedElement}, \mathit{auditAlarmSilencedElement}, \mathit{truncateLogElement}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{USER\_ENTRY\_ELEMENTS} == \{\mathit{userTokenRemovedElement}, \mathit{userTokenPresentElement}, \\
\quad \mathit{userTokenInvalidElement}, \mathit{authCertValidElement}, \mathit{authCertInvalidElement}, \mathit{fingerDetectedElement}, \\
\quad \mathit{fingerTimeoutElement}, \mathit{fingerMatchedElement}, \mathit{fingerNotMatchedElement}, \\
\quad \mathit{authCertWrittenElement}, \mathit{authCertWriteFailedElement}, \\
\quad \mathit{entryPermittedElement}, \mathit{entryTimeoutElement}, \mathit{entryDeniedElement}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{ADMIN\_ELEMENTS} == \{\mathit{adminTokenPresentElement}, \mathit{adminTokenValidElement}, \\
\quad \mathit{adminTokenInvalidElement}, \mathit{adminTokenExpiredElement}, \mathit{adminTokenRemovedElement}, \\
\quad \mathit{invalidOpRequestElement}, \mathit{operationStartElement}, \\
\quad \mathit{archiveLogElement}, \mathit{archiveCompleteElement}, \mathit{archiveCheckFailedElement}, \mathit{updatedConfigDataElement}, \\
\quad \mathit{invalidConfigDataElement}, \mathit{shutdownElement}, \mathit{overrideLockElement}\}
\end{array}
$$

$$\mathit{ENROL\_ELEMENTS} == \{\mathit{enrolmentCompleteElement}, \mathit{enrolmentFailedElement}\}$$

$$\mathit{STARTUP\_ELEMENTS} == \{\mathit{startUnenrolledTISElement}, \mathit{startEnrolledTISElement}\}$$

$$
\begin{array}{l}
\mathit{INFO\_ELEMENTS} == \{\mathit{startUnenrolledTISElement}, \mathit{startEnrolledTISElement}, \mathit{enrolmentCompleteElement}, \\
\quad \mathit{displayChangedElement}, \mathit{screenChangedElement}, \mathit{doorClosedElement}, \mathit{doorOpenedElement}, \\
\quad \mathit{latchLockedElement}, \mathit{latchUnlockedElement}, \mathit{alarmSilencedElement}, \mathit{auditAlarmSilencedElement}, \\
\quad \mathit{userTokenRemovedElement}, \mathit{userTokenPresentElement}, \mathit{authCertValidElement}, \\
\quad \mathit{authCertInvalidElement}, \mathit{fingerDetectedElement}, \mathit{fingerMatchedElement}, \mathit{fingerNotMatchedElement}, \\
\quad \mathit{authCertWrittenElement}, \mathit{entryPermittedElement}, \mathit{adminTokenPresentElement}, \mathit{adminTokenValidElement}, \\
\quad \mathit{adminTokenRemovedElement}, \mathit{operationStartElement}, \mathit{archiveLogElement}, \mathit{archiveCompleteElement}, \\
\quad \mathit{updatedConfigDataElement}, \mathit{shutdownElement}, \mathit{overrideLockElement}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{WARNING\_ELEMENTS} == \{\mathit{enrolmentFailedElement}, \mathit{auditAlarmRaisedElement}, \mathit{userTokenRemovedElement}, \\
\quad \mathit{userTokenInvalidElement}, \mathit{fingerTimeoutElement}, \mathit{authCertWriteFailedElement}, \mathit{entryDeniedElement}, \\
\quad \mathit{entryTimeoutElement}, \mathit{adminTokenInvalidElement}, \\
\quad \mathit{adminTokenExpiredElement}, \mathit{adminTokenRemovedElement}, \mathit{invalidOpRequestElement}, \\
\quad \mathit{archiveCheckFailedElement}, \mathit{invalidConfigDataElement}, \mathit{systemFaultElement}\}
\end{array}
$$

$$\mathit{CRITICAL\_ELEMENTS} == \{\mathit{alarmRaisedElement}, \mathit{truncateLogElement}, \mathit{systemFaultElement}\}$$

▷ See: *systemFaultElement* (p. 28), *displayChangedElement* (p. 28), *screenChangedElement* (p. 28), *doorClosedElement* (p. 28), *doorOpenedElement* (p. 28), *latchLockedElement* (p. 28), *latchUnlockedElement* (p. 28), *alarmRaisedElement* (p. 28), *alarmSilencedElement* (p. 28), *auditAlarmRaisedElement* (p. 28), *auditAlarmSilencedElement* (p. 28), *truncateLogElement* (p. 28), *userTokenRemovedElement* (p. 28), *userTokenPresentElement* (p. 28), *userTokenInvalidElement* (p. 28), *authCertValidElement* (p. 28), *authCertInvalidElement* (p. 28), *fingerDetectedElement* (p. 28), *fingerTimeoutElement* (p. 28), *fingerMatchedElement* (p. 28), *fingerNotMatchedElement* (p. 28), *authCertWrittenElement* (p. 28), *authCertWriteFailedElement* (p. 28), *entryPermittedElement* (p. 28), *entryTimeoutElement* (p. 28), *entryDeniedElement* (p. 28), *adminTokenPresentElement* (p. 28), *adminTokenValidElement* (p. 28), *adminTokenInvalidElement* (p. 28), *adminTokenExpiredElement* (p. 28), *adminTokenRemovedElement* (p. 28), *invalidOpRequestElement* (p. 28), *operationStartElement* (p. 28), *archiveLogElement* (p. 28), *archiveCompleteElement* (p. 28), *archiveCheckFailedElement* (p. 28), *updatedConfigDataElement* (p. 28), *invalidConfigDataElement* (p. 28), *shutdownElement* (p. 28), *overrideLockElement* (p. 28), *enrolmentCompleteElement* (p. 28), *enrolmentFailedElement* (p. 28), *startEnrolledTISElement* (p. 28)

FD.AuditLog.ExtractUser

Each audit element has an associated user, if the user is not relevant or not available then the *noUser* value is used.

$$USERTEXT ::= noUser \mid thisUser \langle\langle CertificateIdC \rangle\rangle$$

▷ See: *CertificateIdC* (p. 16)

There is an extraction function which obtains the user from the current token. This will extract the CertificateIdC from any token sufficiently valid to contain one or return *noUser*.

$$extractUser : TOKENTRYC \longrightarrow USERTEXT$$

▷ See: *TOKENTRYC* (p. 22), *USERTEXT* (p. 30)

Each audit element has a free text field. This is an informal description of the entry and may contain no text.

$$[TEXT]$$

$$noDescription : TEXT$$

```
AuditC
  logTime : TIME
  elementId : AUDIT_ELEMENT
  severity : AUDIT_SEVERITY
  user : USERTEXT
  description : TEXT
```

▷ See: *TIME* (p. 11), *AUDIT_ELEMENT* (p. 28), *AUDIT_SEVERITY* (p. 28), *USERTEXT* (p. 30)

Most audit elements have a user associated with them, where this can be determined it will be supplied.

Some audit elements have different severities depending on their context. A token removal is erroneous during an operation but not at the end of an operation for instance.

We define a function that gives the set of *AUDIT_ELEMENT*s captured within a set of Audit elements.

$$
\begin{array}{|l}
auditType : AuditC \longrightarrow AUDIT\_ELEMENT \\
auditTypes : \mathbb{F}\, AuditC \longrightarrow \mathbb{F}\, AUDIT\_ELEMENT \\
\hline
auditType = (\lambda\, AuditC \bullet elementId) \\
auditTypes = \{A : \mathbb{F}\, AuditC \bullet A \mapsto auditType(\!| A |\!)\}
\end{array}
$$

▷ See: *AuditC* (p. 30), *AUDIT_ELEMENT* (p. 28)

In this implementation the size of each audit element is fixed, we also note that the capacity of a floppy is fixed.

$$
\begin{array}{|l}
sizeAuditElement : \mathbb{N} \\
floppyCapacity : \mathbb{N}
\end{array}
$$

The Audit log consists of a number of *Audit* elements. An audit error alarm will be raised if the audit log becomes full and needs to be archived and cleared.

The Audit log will be implemented as a number of files with a fixed maximum capacity. The intention is to distribute the log across the these files, this should enable truncation to be implemented simply. There will be an internal upper limit to the number of files supported. The size of each file is fixed in terms of the number of audit elements it holds.

$$
\begin{array}{|l}
maxNumberLogFiles : \mathbb{N} \\
maxLogFileEntries : \mathbb{N} \\
maxNumberArchivableFiles : \mathbb{N} \\
\hline
maxNumberLogFiles > 2 \\
maxLogFileEntries \geq 100 \\
maxSupportedLogSize \leq sizeAuditElement * (maxNumberLogFiles - 1) * maxLogFileEntries \\
maxNumberArchivableFiles > 1 \\
maxNumberArchivableFiles * maxLogFileEntries * sizeAuditElement \leq floppyCapacity
\end{array}
$$

▷ See: *sizeAuditElement* (p. 31)

▷ The system supports at least three files.

▷ Each file can hold at least 100 elements.

▷ The files have sufficient capacity to support the maximum supported log size defined in the specification, even when one file is empty. This will ensure that truncation preserves the conditions within the specification.

▷ At least one file can be archived onto a floppy.

In order to simplify the implementation we make a number of assumptions about the internal implementation of the log file.

- When data is archived only full logFiles are removed.

$$LOGFILEINDEX == 1 \mathrel{..} maxNumberLogFiles$$

▷ See: *maxNumberLogFiles* (p. 31)

All audit elements have associated with them a timestamp so it is possible to determine the times of the newest and oldest entries in the log.

$$
\begin{array}{l}
oldestLogTimeC : \mathbb{F}_1\, AuditC \longrightarrow TIME \\
newestLogTimeC : \mathbb{F}_1\, AuditC \longrightarrow TIME \\
\hline
\forall A, B : \mathbb{F}\, AuditC \bullet \\
\quad newestLogTimeC(A \cup B) \geq newestLogTimeC\, A \\
\quad \land oldestLogTimeC(A \cup B) \leq oldestLogTimeC\, A \\
\quad \land A \neq \emptyset \Rightarrow oldestLogTimeC\, A = min\,\{audit : A \bullet audit.logTime\} \\
\quad \land A \neq \emptyset \Rightarrow newestLogTimeC\, A = max\,\{audit : A \bullet audit.logTime\}
\end{array}
$$

▷ See: *AuditC* (p. 30), *TIME* (p. 11)

▷ Both these functions are monotonic.

At any time each log file will either be empty and *free* for use, *used* (or in use) or *archived*, in that an attempt has been made to archive the data.

$$LOGFILESTATUS ::= free \mid archived \mid used$$

As all log entries are time stamped there is no requirement to impose an ordering on the entries in an audit file, however we do insist that the log files can be ordered such that the all the elements in the oldest file are older than all the elements in the other files.

$$
\begin{array}{l}
\textbf{AuditLogC} \\
\hline
logFiles : LOGFILEINDEX \longrightarrow \mathbb{F}\, AuditC \\
currentLogFile : LOGFILEINDEX \\
usedLogFiles : \mathrm{iseq}\, LOGFILEINDEX \\
freeLogFiles : \mathbb{P}\, LOGFILEINDEX \\
logFilesStatus : LOGFILEINDEX \longrightarrow LOGFILESTATUS \\
numberLogEntries : \mathbb{N} \\
auditAlarmC : ALARM \\
\hline
freeLogFiles = \mathrm{dom}(logFiles \rhd \{\emptyset\}) \\
freeLogFiles = \mathrm{dom}(logFilesStatus \rhd \{free\}) \\
\mathrm{ran}\, usedLogFiles = \mathrm{dom}(logFilesStatus \rhd \{archived, used\}) \\
\forall file1, file2 : \mathrm{ran}\, usedLogFiles \bullet \\
\quad usedLogFiles^{\sim} file1 < usedLogFiles^{\sim} file2 \Rightarrow \\
\quad\quad newestLogTimeC\,(logFiles\, file1) \leq oldestLogTimeC\,(logFiles\, file2) \\
usedLogFiles \neq \langle\rangle \\
\quad \Rightarrow (\forall file : LOGFILEINDEX \mid file \in \mathrm{ran}(front\, usedLogFiles) \bullet \#(logFiles\, file) = maxLogFileEntries) \\
usedLogFiles \neq \langle\rangle \\
\quad\quad \land currentLogFile = last\, usedLogFiles \\
\quad\quad \land numberLogEntries = (\#usedLogFiles - 1) * maxLogFileEntries + \#(logFiles\, currentLogFile) \\
\quad \lor \\
usedLogFiles = \langle\rangle \land numberLogEntries = 0
\end{array}
$$

▷ See: *LOGFILEINDEX* (p. 31), *AuditC* (p. 30), *LOGFILESTATUS* (p. 32), *ALARM* (p. 21), *free* (p. 32), *archived* (p. 32), *used* (p. 32), *oldestLogTimeC* (p. 32)

▷ The *freeLogFiles* are exactly those which are empty.

- The *usedLogFiles* is a sequence of log files which are non-empty.
- The *usedLogFiles* are ordered such that the oldest entries appear in the first log file in the sequence.
- All but the last *usedLogFiles* are filled to their maximum capacity.
- The *numberLogEntries* is completely derived and is maintained for convenience.

3.3 Key Store

FD.KeyStore.State
FS.KeyStore.State

TIS maintains a key store, this is managed by the Crypto Library. It contains all Issuer keys relevant to its function. This will include known CAs, AAs and its own key.

The only private key part held will be for TIS's own key. Having a private key within the set of keys indicates that the TIS knows who it is.

TIS will generate its key at the first start-up and request an Id certificate from a CA. This activity is not modelled here and will not be implemented. We model the private part of the TIS key as *theTISKey*. This will be inserted into the keystore at enrolment. The private part of the TIS key is used subsequently to sign authorisation certificates.

Only one public key is held for each Issuer.

KeyStoreC

$$
\begin{array}{l}
keys : \mathbb{F}\ KeyPart \\
theTISKey : KEYPART \\[4pt]
keyMatchingIssuer : USERID \longrightarrow \mathrm{optional}\ KEYPART \\
privateKey : \mathrm{optional}\ KeyPart \\
\hline
\{key : keys \mid key.keyType = private \bullet key.keyData\} \subseteq \{theTISKey\} \\[4pt]
privateKey \neq nil \Rightarrow \\
\quad (\exists\, ownPub : keys \bullet ownPub.keyType = public \\
\quad\quad \land\ ownPub.keyOwner = (the\ privateKey).keyOwner) \\[4pt]
\#\{key : keys \mid key.keyType = public\} = \#\{key : keys \mid key.keyType = public \bullet key.keyOwner\} \\[4pt]
keyMatchingIssuer = (USERID \times \{\varnothing\}) \oplus \{key : keys \mid key.keyType = public \bullet key.keyOwner.id \mapsto \{key.keyData\}\} \\
privateKey = \{key : keys \mid key.keyType = private\}
\end{array}
$$

- See: *KeyPart* (p. 14), *private* (p. 14), *public* (p. 14)

- The Crypto Library provides facilities to query information, these are modelled by *keysMatchingIssuer* and *privateKeys*.
- *keysMatchingIssuer* and *privateKeys* are completely defined by invarients.

3.4 Certificate Store

FD.CertificateStore.State

TIS issues certificates, these certificates have a unique identifier, which is composed of the unique *USERID* identification given to TIS and a serial number.

TIS must maintain knowlege of the serial numbers already issued to ensure that new certificates are issued with a unique serial number.

CertificateStore
$nextSerialNumber : \mathbb{N}$

3.5 System Statistics

FD.Stats.State
FS.Stats.State

The system statistics recorded are as defined in the formal specification [4].

TIS keeps track of the number of times that a entry to the enclave has been attempted (and denied) and the number of times it has succeeded. It also records the number of times that a biometric comparison has been made (and failed) and the number of times it succeeded.

By retaining these statistics it is possible for the performance of the system to be monitored.

StatsC
$successEntryC : \mathbb{N}$
$failEntryC : \mathbb{N}$
$successBioC : \mathbb{N}$
$failBioC : \mathbb{N}$

3.6 Administration

FD.Admin.State
FS.Admin.State

This component of TIS is not refined from the specification.

In addition to its role of authorising entry to the enclave, TIS supports a number of administrative operations.

- ArchiveLog - writes the archive log to floppy and truncates the internally held archive log.
- UpdateConfiguration - accepts new configuration data from a floppy.
- OverrideDoorLock - unlocks the enclave door.
- Shutdown - stops TIS, leaving the protected entry to the enclave secure.

$$ADMINOP ::= archiveLog \mid updateConfigData \mid overrideLock \mid shutdownOp$$

▷ Definition repeated from Formal Specification [4]

Only users with administrator privileges can make use of the TIS to perform administrative functions. There are a number of different administrator privileges that may be held.

$$ADMINPRIVILEGE == \{guard, auditManager, securityOfficer\}$$

▷ See: *guard* (p. 12), *auditManager* (p. 12), *securityOfficer* (p. 12)

▷ Definition repeated from Formal Specification [4]

The role held by the administrator will determine the operations available to the administrator. For security reasons an administrator can only hold one role.

AdminC

$$
\begin{array}{l}
rolePresentC : \mathrm{optional}\, ADMINPRIVILEGE \\
availableOpsC : \mathbb{P}\, ADMINOP \\
currentAdminOpC : \mathrm{optional}\, ADMINOP \\
\hline
rolePresentC = nil \Rightarrow availableOpsC = \varnothing \\
(rolePresentC \neq nil \land the\, rolePresentC = guard) \Rightarrow availableOpsC = \{overrideLock\} \\
(rolePresentC \neq nil \land the\, rolePresentC = auditManager) \Rightarrow availableOpsC = \{archiveLog\} \\
(rolePresentC \neq nil \land the\, rolePresentC = securityOfficer) \Rightarrow availableOpsC = \{updateConfigData, shutdownOp\} \\
currentAdminOpC \neq nil \Rightarrow \\
\quad (the\, currentAdminOpC \in availableOpsC \land rolePresentC \neq nil)
\end{array}
$$

▷ See: *ADMINPRIVILEGE* (p. 34), *ADMINOP* (p. 34), *guard* (p. 12), *overrideLock* (p. 34), *auditManager* (p. 12), *archiveLog* (p. 34), *securityOfficer* (p. 12), *updateConfigData* (p. 34), *shutdownOp* (p. 34)

▷ The *availableOpsC* are completely determined by the roles present and will be implemented using a constant table.

In order to perform an administrative operation an administrator must be present. Presence will be determined by an appropriate token being present in the administrator's card reader.

3.7 Real World Entities

FD.RealWorld.State
FS.RealWorld.State

The latch is allowed to be in two states: *locked* and *unlocked*. When the latch is unlocked, *latchTimeoutC* will be set to the time at which the lock must again be *locked*.

The alarm is similar to the latch, in that it has a *silent*, and *alarming*, with an *alarmTimeoutC*. Once the door and latch move into a potentially insecure state (door *open* and latch *locked*) then the *alarmTimeoutC* is set to the time at which the alarm will sound.

Within the implementation the *currentLatchC* and *doorAlarmC* will be explicitly calculated athough they can be entirely derived.

DoorLatchAlarmC

$$
\begin{array}{l}
currentTimeC : TIME \\
currentDoorC : DOOR \\
currentLatchC : LATCH \\
doorAlarmC : ALARM \\
latchTimeoutC : TIME \\
alarmTimeoutC : TIME
\end{array}
$$

▷ See: *TIME* (p. 11), *DOOR* (p. 21), *LATCH* (p. 21), *ALARM* (p. 21)

The ID Station holds internal representations of all of the Real World, plus its own data. It holds separate indications of the presence of input in the real world peripherals of the User Token, Admin Token, Fingerprint reader, and Floppy disk. This is so that once the input has been read, and the card, finger or disk removed, the ID Station can continue to know what the value was, even if it later detects that the real world entity has been removed.

UserTokenC
$currentUserTokenC : TOKENTRYC$
$userTokenPresenceC : PRESENCE$

▷ See: *TOKENTRYC* (p. 22), *PRESENCE* (p. 11)

AdminTokenC
$currentAdminTokenC : TOKENTRYC$
$adminTokenPresenceC : PRESENCE$

▷ See: *TOKENTRYC* (p. 22), *PRESENCE* (p. 11)

The core TIS does not need to know what the current fingerprint is since it is always read directly from the real world by the Biometrics Library.

FingerC
$fingerPresenceC : PRESENCE$

▷ See: *PRESENCE* (p. 11)

The core TIS does not need to preserve the value of the current keyed data since it is always read directly from the real world and the information does not need to be persistent.

KeyboardC
$keyedDataPresenceC : PRESENCE$

▷ See: *PRESENCE* (p. 11)

We need to retain an internal view of the last data written to the floppy as well as the current data on the floppy, this is because we need to check that writing to floppy works when we archive the log.

FloppyC
$currentFloppyC : FLOPPYC$
$writtenFloppyC : FLOPPYC$
$floppyPresenceC : PRESENCE$

▷ See: *FLOPPYC* (p. 22), *PRESENCE* (p. 11)

The ID Station screen within the enclave may display many pieces of information. The majority of this data will be determined by state invarients. In addition to those identified in the specification we now identify the alarm states as being necessary display elements.

ScreenC
screenStatsC : *SCREENTEXTC*
screenMsgC : *SCREENTEXTC*
screenConfigC : *SCREENTEXTC*
screenDoorAlarm : *SCREENTEXTC*
screenLogAlarm : *SCREENTEXTC*

▷ See: *SCREENTEXTC* (p. 23)

3.8 Internal State

FD.Internal.State
FD.Internal.State

STATUS and *ENCLAVESTATUS* are a purely internal records of the progress through processing. *STATUS* tracks progress through user entry, while *ENCLAVESTATUS* tracks progress through all activities performed within the enclave.

STATUS ::= *quiescent* |
gotUserToken | *waitingFinger* | *gotFinger* | *waitingUpdateToken* | *waitingEntry* |
waitingRemoveTokenSuccess | *waitingRemoveTokenFail*
ENCLAVESTATUS ::= *notEnrolled* | *waitingEnrol* | *waitingEndEnrol* |
enclaveQuiescent |
gotAdminToken | *waitingRemoveAdminTokenFail* | *waitingStartAdminOp* | *waitingFinishAdminOp* |
shutdown

▷ Definitions repeated from Formal Specification [4]

The states *quiescent* and *enclaveQuiescent* represent the enclave interface and the user entry interface being quiescent.

The states *gotUserToken*, .. *waitingRemoveTokenFail* are all associated with the process of user authentication and entry. These are described futher in Section 6.

The states *notEnrolled*, .. *waitingEnrolEnd* reflect enrolment activity that must be performed before TIS can offer any of its normal operations. Once the TIS is successfully enrolled it becomes *quiescent*.

The states *gotAdminToken*, .. *waitingFinishAdminOp* reflect activity at the TIS console relating to administrator use of TIS.

The state *shutdown* models the system when it is shutdown.

There are two timeouts held internally, one of these controls the system wait for the user to remove their token before opening the door in a successful user entry scenario. The other controls the system wait for the user to provide a good fingerprint for verification before giving up on this part of the authentication process.

InternalC
statusC : *STATUS*
enclaveStatusC : *ENCLAVESTATUS*
tokenRemovalTimeoutC : *TIME*
fingerTimeout : *TIME*

▷ See: *STATUS* (p. 37), *ENCLAVESTATUS* (p. 37), *TIME* (p. 11)

3.9 The whole Token ID Station

FD.TIS.State
FS.TIS.State

The whole Token ID Station is constructed from combining the described state components.

In addition there is a display outside the enclave and and screen within the enclave. The ID Station screen within the enclave may display many pieces of information. The majority of this data will be determined by state invariants.

If the authentication protocol has moved on to requesting a fingerprint, then the User Token will have passed its validation checks.

Similarly if the system considers there to be an administrator present then the Admin Token will have passed its validation checks.

Once the ID station has been enrolled it has a private key, its own key.

TIS is only ever in the two states *waitingStartAdminOp* or *waitingFinishAdminOp* when then there is a current admin operation in progress. For single phase operations the state *waitingFinishAdminOp* is not used.

TIS will only read the Admin Token to log on an administrator if there is not an administrator role currently present.

$$
\begin{array}{l}
\textit{IDStationC} \\
\hline
\textit{UserTokenC} \\
\textit{AdminTokenC} \\
\textit{FingerC} \\
\textit{DoorLatchAlarmC} \\
\textit{FloppyC} \\
\textit{KeyboardC} \\
\textit{ConfigC} \\
\textit{StatsC} \\
\textit{KeyStoreC} \\
\textit{CertificateStore} \\
\textit{AdminC} \\
\textit{AuditLogC} \\
\textit{InternalC} \\
\\
\textit{currentDisplayC} : \textit{DISPLAYMESSAGE} \\
\textit{currentScreenC} : \textit{ScreenC} \\
\hline
\textit{statusC} \in \{ \textit{gotFinger}, \textit{waitingFinger}, \textit{waitingUpdateToken}, \textit{waitingEntry} \} \Rightarrow \\
\quad ((\exists \textit{ValidTokenC} \bullet \textit{goodTC}(\theta \textit{ValidTokenC}) = \textit{currentUserTokenC}) \\
\qquad \lor (\exists \textit{TokenWithValidAuthC} \bullet \textit{goodTC}(\theta \textit{TokenWithValidAuthC}) = \textit{currentUserTokenC})) \\
\\
\textit{rolePresentC} \neq \textit{nil} \Rightarrow \\
\quad (\exists \textit{TokenWithValidAuthC} \bullet \textit{goodTC}(\theta \textit{TokenWithValidAuthC}) = \textit{currentAdminTokenC}) \\
\\
\textit{enclaveStatusC} \notin \{ \textit{notEnrolled}, \textit{waitingEnrol}, \textit{waitingEndEnrol} \} \Rightarrow \\
\quad \#\{\textit{key} : \textit{keys} \mid \textit{key}.\textit{keyType} = \textit{private}\} = 1 \\
\\
\textit{enclaveStatusC} \in \{ \textit{waitingStartAdminOp}, \textit{waitingFinishAdminOp} \} \Leftrightarrow \textit{currentAdminOpC} \neq \textit{nil} \\
\\
(\textit{currentAdminOpC} \neq \textit{nil} \land \textit{the}\ \textit{currentAdminOpC} \in \{ \textit{shutdownOp}, \textit{overrideLock} \}) \\
\qquad \Rightarrow \textit{enclaveStatusC} = \textit{waitingStartAdminOp} \\
\\
\textit{enclaveStatusC} = \textit{gotAdminToken} \Rightarrow \textit{rolePresentC} = \textit{nil} \\
\\
\textit{currentScreenC}.\textit{screenStatsC} = \textit{displayStatsC}(\theta \textit{StatsC}) \\
\textit{currentScreenC}.\textit{screenConfigC} = \textit{displayConfigDataC}(\theta \textit{ConfigC}) \\
\textit{currentScreenC}.\textit{screenDoorAlarm} = \textit{displayAlarm}\ \textit{doorAlarmC} \\
\textit{currentScreenC}.\textit{screenLogAlarm} = \textit{displayAlarm}\ \textit{auditAlarmC} \\
\hline
\end{array}
$$

- ▷ See: *UserTokenC* (p. 36), *AdminTokenC* (p. 36), *FingerC* (p. 36), *DoorLatchAlarmC* (p. 35), *FloppyC* (p. 36), *KeyboardC* (p. 36), *ConfigC* (p. 27), *StatsC* (p. 34), *KeyStoreC* (p. 33), *CertificateStore* (p. 34), *AdminC* (p. 35), *AuditLogC* (p. 32), *InternalC* (p. 37), *DISPLAYMESSAGE* (p. 22), *ScreenC* (p. 36), *gotFinger* (p. 37), *waitingFinger* (p. 37), *waitingUpdateToken* (p. 37), *waitingEntry* (p. 37), *ValidTokenC* (p. 19), *goodTC* (p. 22), *TokenWithValidAuthC* (p. 19), *notEnrolled* (p. 37), *waitingEnrol* (p. 37), *waitingEndEnrol* (p. 37), *private* (p. 14), *waitingStartAdminOp* (p. 37), *waitingFinishAdminOp* (p. 37), *shutdownOp* (p. 34), *overrideLock* (p. 34), *displayStatsC* (p. 23), *displayConfigDataC* (p. 23)

- ▷ Note that the token may not still be current since time will have moved on since the checks were performed.
- ▷ Operations that can be performed in a single phase do not result in TIS entering the state *waitingFinishAdminOp* as they are finished when they are started.
- ▷ TIS only enters the state *gotAdminToken* when there is no administrator present.
- ▷ Invarients define many of the screen components.

4 OPERATIONS INTERFACING TO THE ID STATION

4.1 Real World Changes

The monitored components of the real world can change at any time. The only assumption we make of the real world is that time increases.

RealWorldTimeChanges

$nowC, nowC' : TIME$

$nowC' \geq nowC$

▷ See: *TIME* (p. 11)

$RealWorldChangesC \mathrel{\widehat=} RealWorldTimeChanges \land \Delta RealWorldC$

▷ See: *RealWorldTimeChanges* (p. 40), *RealWorldC* (p. 25)

RealWorldChanges

$\Delta RealWorld$

$now' \geq now$

4.2 Obtaining inputs from the real world

Most data is polled from the real world on a periodic basis. Some items are however only read when the system is in a state to receive data. This includes reading the contents of Tokens, the floppy disk and the keyboard.

4.2.1 Polling the real world

FD.Interface.TISPoll
FS.Interface.TISPoll

We poll all of the real world entities. For those entities that take time to read we simply check for the presence of data.

FD.Interface.PollTime
FD.Interface.TISPoll

PollTimeC

$\Delta DoorLatchAlarmC$
$RealWorldC$

$currentTimeC' = nowC$

▷ See: *DoorLatchAlarmC* (p. 35), *RealWorldC* (p. 25)

FD.Interface.PollDoor
FD.Interface.TISPoll

When polling the door, we do not change the alarm timeout or latch timeout.

$$
\begin{array}{l}
\textit{PollDoorC} \\
\hline
\Delta \textit{DoorLatchAlarmC} \\
\textit{RealWorldC} \\
\hline
\textit{currentDoorC}' = \textit{doorC} \\
\textit{latchTimeoutC}' = \textit{latchTimeoutC} \\
\textit{alarmTimeoutC}' = \textit{alarmTimeoutC} \\
\textit{doorAlarmC}' = \textit{doorAlarmC} \\
\textit{currentLatchC}' = \textit{currentLatchC}
\end{array}
$$

▷ See: *DoorLatchAlarmC* (p. 35), *RealWorldC* (p. 25)

The internal representation of the latch or the alarm may need to be updated as a result of changes to the attributes that influence their values.

$$\textit{PollTimeAndDoor} \mathrel{\widehat{=}} (\textit{PollTimeC} \land \textit{PollDoorC}) \mathbin{⨾} \textit{UpdateInternalLatch} \mathbin{⨾} \textit{UpdateInternalAlarm}$$

▷ See: *PollTimeC* (p. 40), *PollDoorC* (p. 41)

The system only polls for the presence of the tokens, finger, floppy and keyboard data. This is a refinement from the Formal Specification [4], which made the assumption that all inputs could be read sufficiently fast to perform the read regularly, see Section 2.2.3. These entities are either required infrequently or are time consuming to read so within the design they are only read when data is present and the system requires the information.

FD.Interface.PollUserToken
FD.Interface.TISPoll

$$
\begin{array}{l}
\textit{PollUserTokenC} \\
\hline
\Delta \textit{UserTokenC} \\
\textit{RealWorldC} \\
\hline
\textit{userTokenPresenceC}' = \textit{present} \Leftrightarrow \textit{userTokenC} \neq \textit{noTC} \\
\textit{currentUserTokenC}' = \textit{currentUserTokenC}
\end{array}
$$

▷ See: *UserTokenC* (p. 36), *RealWorldC* (p. 25), *present* (p. 11), *noTC* (p. 22)

FD.Interface.PollAdminToken
FD.Interface.TISPoll

$$
\begin{array}{l}
\textit{PollAdminTokenC} \\
\hline
\Delta \textit{AdminTokenC} \\
\textit{RealWorldC} \\
\hline
\textit{adminTokenPresenceC}' = \textit{present} \Leftrightarrow \textit{adminTokenC} \neq \textit{noTC} \\
\textit{currentAdminTokenC}' = \textit{currentAdminTokenC}
\end{array}
$$

▷ See: *AdminTokenC* (p. 36), *RealWorldC* (p. 25), *present* (p. 11), *noTC* (p. 22)

FD.Interface.PollFinger
FD.Interface.TISPoll

$$\begin{array}{l} \textit{PollFingerC} \\ \Delta \textit{FingerC} \\ \textit{RealWorldC} \\ \hline \textit{fingerPresenceC}' = \textit{present} \Leftrightarrow \textit{fingerC} \neq \textit{noFP} \end{array}$$

▷ See: *FingerC* (p. 36), *RealWorldC* (p. 25), *present* (p. 11), *noFP* (p. 22)

FD.Interface.PollFloppy
FD.Interface.TISPoll

$$\begin{array}{l} \textit{PollFloppyC} \\ \Delta \textit{FloppyC} \\ \textit{RealWorldC} \\ \hline \textit{floppyPresenceC}' = \textit{present} \Leftrightarrow \textit{floppyC} \neq \textit{noFloppyC} \\ \textit{currentFloppyC}' = \textit{currentFloppyC} \\ \textit{writtenFloppyC}' = \textit{writtenFloppyC} \end{array}$$

▷ See: *FloppyC* (p. 36), *RealWorldC* (p. 25), *present* (p. 11), *noFloppyC* (p. 22)

FD.Interface.PollKeyboard
FD.Interface.TISPoll

$$\begin{array}{l} \textit{PollKeyboardC} \\ \Delta \textit{KeyboardC} \\ \textit{RealWorldC} \\ \hline \textit{keyedDataPresenceC} = \textit{present} \Leftrightarrow \textit{keyboardC} \neq \textit{noKB} \end{array}$$

▷ See: *KeyboardC* (p. 36), *RealWorldC* (p. 25), *present* (p. 11), *noKB* (p. 23)

So the overall poll operation is obtained by combining all the individual polling actions.

FD.Interface.DisplayPollUpdate
FD.Interface.TISPoll

If the user is currently being invited to enter the enclave on the display and the door becomes latched then the display will change to indicate that the system is no longer offering entry.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{DisplayPollUpdate}}} \\
\Delta IDStationC \\
\hline
currentLatchC' = locked \\
\quad \land (currentDisplayC = doorUnlocked \\
\qquad \land (statusC \neq waitingRemoveTokenFail \land currentDisplayC' = welcome \\
\qquad\quad \lor statusC = waitingRemoveTokenFail \land currentDisplayC' = removeToken) \\
\quad \lor currentDisplayC \neq doorUnlocked \\
\qquad \land currentDisplayC' = currentDisplayC) \\
\lor currentLatchC' \neq locked \\
\quad \land currentDisplayC' = currentDisplayC \\
\hline
\end{array}
$$

▷ See: *IDStationC* (p. 38), *locked* (p. 21), *doorUnlocked* (p. 22), *waitingRemoveTokenFail* (p. 37), *welcome* (p. 22)

We assume that while polling occurs the *RealWorld* does not change. This is a reasonable assumption since all information polled is easy and quick to obtain.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{PollC}}} \\
\Delta IDStationC \\
\Xi RealWorldC \\
\\
PollTimeAndDoor \\
PollUserTokenC \\
PollAdminTokenC \\
PollFingerC \\
PollFloppyC \\
PollKeyboardC \\
DisplayPollUpdate \\
\\
\Xi ConfigC \\
\Xi KeyStoreC \\
\Xi CertificateStore \\
\Xi AdminC \\
\Xi StatsC \\
\Xi InternalC \\
\hline
currentScreenC' = currentScreenC \\
\hline
\end{array}
$$

▷ See: *IDStationC* (p. 38), *RealWorldC* (p. 25), *PollTimeAndDoor* (p. 41), *PollUserTokenC* (p. 41), *PollAdminTokenC* (p. 41), *PollFingerC* (p. 42), *PollFloppyC* (p. 42), *PollKeyboardC* (p. 42), *DisplayPollUpdate* (p. 42), *ConfigC* (p. 27), *KeyStoreC* (p. 33), *CertificateStore* (p. 34), *AdminC* (p. 35), *StatsC* (p. 34), *InternalC* (p. 37)

Polling the real world may result in changes which need to be audited. The only events that will appear in the audit log during polling are the user independent elements.

$$
\begin{array}{l}
TISPollC \mathrel{\widehat=} PollC \land LogChangeC \\
\qquad \land [AddElementsToLogC \mid auditTypes\ newElements? \subseteq USER\_INDEPENDENT\_ELEMENTS] \setminus (newElements?)
\end{array}
$$

▷ See: *PollC* (p. 43), *USER_INDEPENDENT_ELEMENTS* (p. 29)

4.2.2 Reading Real World Values

Those entities that are read on demand are the tokens and floppy.

ReadUserTokenC
$\Delta UserTokenC$
$RealWorldC$

$userTokenPresenceC' = userTokenPresenceC$
$currentUserTokenC' = userTokenC$

▷ See: *UserTokenC* (p. 36), *RealWorldC* (p. 25)

ReadAdminTokenC
$\Delta AdminTokenC$
$RealWorldC$

$adminTokenPresenceC' = adminTokenPresenceC$
$currentAdminTokenC' = adminTokenC$

▷ See: *AdminTokenC* (p. 36), *RealWorldC* (p. 25)

ReadFloppyC
$\Delta FloppyC$
$RealWorldC$

$floppyPresenceC' = floppyPresenceC$
$currentFloppyC' = floppyC$
$writtenFloppyC' = writtenFloppyC$

▷ See: *FloppyC* (p. 36), *RealWorldC* (p. 25)

4.3 The ID Station changes the world

4.3.1 Periodic Updates

We consider the process of updating the real world with the current internal representation, one variable at a time.

FD.Interface.UpdateLatch	
FD.Interface.TISUpdates	*FD.Interface.TISEarlyUpdates*

UpdateLatchC
$\Xi DoorLatchAlarmC$
$RealWorldChangesC$

$latchC' = currentLatchC$

▷ See: *DoorLatchAlarmC* (p. 35), *RealWorldChangesC* (p. 40)

FD.Interface.UpdateAlarm	
FD.Interface.TISUpdates	*FD.Interface.TISEarlyUpdates*

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{UpdateAlarmC}} \\
\hline
\Xi DoorLatchAlarmC \\
AuditLogC \\
RealWorldChangesC \\
\hline
alarmC' = alarming \Leftrightarrow doorAlarmC = alarming \lor auditAlarmC = alarming \\
\hline
\end{array}
$$

▷ See: *DoorLatchAlarmC* (p. 35), *AuditLogC* (p. 32), *RealWorldChangesC* (p. 40), *alarming* (p. 21)

FD.Interface.UpdateDisplay
FD.Interface.TISUpdates

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{UpdateDisplayC}} \\
\hline
\Delta IDStationC \\
RealWorldChangesC \\
\hline
displayC' = currentDisplayC \\
currentDisplayC' = currentDisplayC \\
\hline
\end{array}
$$

▷ See: *IDStationC* (p. 38), *RealWorldChangesC* (p. 40)

Configuration Data is only displayed if the security officer is present. System statistics are only displayed if an administrator is logged on.

FD.Interface.UpdateScreen
FD.Interface.TISUpdates

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{UpdateScreenC}} \\
\hline
\Xi IDStationC \\
\Xi AdminC \\
RealWorldChangesC \\
\hline
screenC'.screenMsgC = currentScreenC.screenMsgC \\
screenC'.screenConfigC = \mathbf{if}\ the\ rolePresentC = securityOfficer\ \mathbf{then}\ displayConfigDataC(\theta ConfigC)\ \mathbf{else}\ clearC \\
screenC'.screenStatsC = \mathbf{if}\ rolePresentC \neq nil\ \mathbf{then}\ displayStatsC(\theta StatsC)\ \mathbf{else}\ clearC \\
screenC'.screenDoorAlarm = displayAlarm\ doorAlarmC \\
screenC'.screenLogAlarm = displayAlarm\ auditAlarmC \\
\hline
\end{array}
$$

▷ See: *IDStationC* (p. 38), *AdminC* (p. 35), *RealWorldChangesC* (p. 40), *securityOfficer* (p. 12), *displayConfigDataC* (p. 23), *ConfigC* (p. 27), *clearC* (p. 23), *displayStatsC* (p. 23), *StatsC* (p. 34)

All these can be combined, along with no change in the remaining real world variables, to represent the regular updating of the world.

When updates to the real world occur it is possible that interfacing with external devices will result in a system fault that is audited. Not other aspects of TIS will change during updates of the real world.

FD.Interface.TISEarlyUpdates
FS.Interface.TISEarlyUpdates

The alarm and the door latch will need to be updated as soon as possible after polling the real world, this ensures that the system is kept secure.

$$
\begin{aligned}
\mathit{TISEarlyUpdateC} \mathrel{\widehat=}\ & \mathit{UpdateLatchC} \land \mathit{UpdateAlarmC} \\
& \land [\, \mathit{RealWorldChangesC} \mid \mathit{screenC}' = \mathit{screenC} \land \mathit{displayC}' = \mathit{displayC} \,] \\
& \land \Xi \mathit{UserTokenC} \land \Xi \mathit{AdminTokenC} \land \Xi \mathit{FingerC} \land \Xi \mathit{FloppyC} \land \\
& \Xi \mathit{ScreenC} \land \Xi \mathit{KeyboardC} \land \Xi \mathit{ConfigC} \land \Xi \mathit{StatsC} \\
& \land \Xi \mathit{KeyStoreC} \land \Xi \mathit{AdminC} \land \Xi \mathit{InternalC} \\
& \land [\mathit{AddElementsToLogC} \mid \mathit{auditTypes}\ \mathit{newElements}? \subseteq \{\mathit{systemFaultElement}\}] \setminus (\mathit{newElements}?)
\end{aligned}
$$

▷ See: *UpdateLatchC* (p. 44), *UpdateAlarmC* (p. 44), *RealWorldChangesC* (p. 40), *UserTokenC* (p. 36), *AdminTokenC* (p. 36), *FingerC* (p. 36), *FloppyC* (p. 36), *ScreenC* (p. 36), *KeyboardC* (p. 36), *ConfigC* (p. 27), *StatsC* (p. 34), *KeyStoreC* (p. 33), *AdminC* (p. 35), *InternalC* (p. 37), *systemFaultElement* (p. 28)

FD.Interface.TISUpdates
FS.Interface.TISUpdates

The alarm, door latch, display and TIS screen will be updated after performing any calculations.

$$
\begin{aligned}
\mathit{TISUpdateC} \mathrel{\widehat=}\ & \mathit{UpdateLatchC} \land \mathit{UpdateAlarmC} \land \mathit{UpdateDisplayC} \land \mathit{UpdateScreenC} \\
& \land \Xi \mathit{UserTokenC} \land \Xi \mathit{AdminTokenC} \land \Xi \mathit{FingerC} \land \Xi \mathit{FloppyC} \land \\
& \Xi \mathit{KeyboardC} \land \Xi \mathit{ConfigC} \land \Xi \mathit{StatsC} \\
& \land \Xi \mathit{KeyStoreC} \land \Xi \mathit{AdminC} \land \Xi \mathit{InternalC} \\
& \land [\mathit{AddElementsToLogC} \mid \mathit{auditTypes}\ \mathit{newElements}? \subseteq \{\mathit{systemFaultElement}\}] \setminus (\mathit{newElements}?)
\end{aligned}
$$

▷ See: *UpdateLatchC* (p. 44), *UpdateAlarmC* (p. 44), *UpdateDisplayC* (p. 45), *UpdateScreenC* (p. 45), *UserTokenC* (p. 36), *AdminTokenC* (p. 36), *FingerC* (p. 36), *FloppyC* (p. 36), *KeyboardC* (p. 36), *ConfigC* (p. 27), *StatsC* (p. 34), *KeyStoreC* (p. 33), *AdminC* (p. 35), *InternalC* (p. 37), *systemFaultElement* (p. 28)

4.3.2 Updating the user Token

FD.Interface.UpdateToken
FS.Interface.UpdateToken

We have a further operation, which writes to the User Token only. We treat this separately because we expect to update the other devices regularly and frequently, but we will only be updating the User Token when we have something to write.

UpdateUserTokenC

$$
\begin{array}{|l}
\Delta \mathit{IDStationC} \\
\mathit{RealWorldChangesC} \\
\\
\Xi \mathit{TISControlledRealWorldC} \\
\hline
\mathit{userTokenC}' = \mathit{currentUserTokenC}
\end{array}
$$

▷ See: *IDStationC* (p. 38), *RealWorldChangesC* (p. 40), *TISControlledRealWorldC* (p. 24)

4.3.3 Updating the Floppy

FD.Interface.UpdateFloppy
FS.Interface.UpdateFloppy

We have an operation which writes to the Floppy only. We will only be updating the Floppy disk when we have something to write.

UpdateFloppyC

$\Delta IDStationC$
$RealWorldChangesC$

$\Xi UserTokenC$
$\Xi AdminTokenC$
$\Xi FingerC$
$\Xi DoorLatchAlarmC$
$\Xi KeyboardC$
$\Xi ConfigC$
$\Xi StatsC$
$\Xi KeyStoreC$
$\Xi AdminC$
$\Xi AuditLogC$
$\Xi InternalC$

$\Xi TISControlledRealWorldC$

$floppyC' = writtenFloppyC$
$currentFloppyC' = badFloppyC$

$floppyPresenceC' = floppyPresenceC$
$currentDisplayC' = currentDisplayC$
$currentScreenC' = currentScreenC$

▷ See: *IDStationC* (p. 38), *RealWorldChangesC* (p. 40), *UserTokenC* (p. 36), *AdminTokenC* (p. 36), *FingerC* (p. 36), *DoorLatchAlarmC* (p. 35), *KeyboardC* (p. 36), *ConfigC* (p. 27), *StatsC* (p. 34), *KeyStoreC* (p. 33), *AdminC* (p. 35), *AuditLogC* (p. 32), *InternalC* (p. 37), *TISControlledRealWorldC* (p. 24), *badFloppyC* (p. 22)

▷ Having written the floppy we can assume nothing about the *currentFloppy* until we next poll. We do not know what data is on the floppy as it may have been corrupted during the write. This ensures that the readback we do is forced to be effective.

4.4 Clearing Biometric Data

FD.Interface.FlushFingerData
FDP_RIP.2.1

The biometric device must be cleared of stale data after a fingerprint has been verified and before an attempt is made to capture fingerprint data. This will force the biometric device to capture fresh data.

FlushFingerDataC

$fingerC, fingerC' : FINGERPRINTTRY$

$fingerC' = noFP$

▷ See: *FINGERPRINTTRY* (p. 22), *noFP* (p. 22)

5 INTERNAL OPERATIONS

In this section we present a number of operations performed internally by the TIS. These operations are combined to create the operations available to the user.

The majority of these operations only update elements in a single schema, although they may read values from other schemas to influence new values, these may be viewed as imports to the operations.

5.1 Updating the Audit Log

5.1.1 Adding elements to the Log

FD.AuditLog.AddElementsToLog
FS.AuditLog.AddElementsToLog

When we add a set of entries to the log, either there is sufficient room in the log for the new entries, in which case the log will not need to be truncated, or there is insufficient room in the log, in which case the log will be truncated losing the oldest data.

The implementation uses several files to hold the log. If the current file has sufficient room to take the log then the new entries will be added to the current file, otherwise a new file will need to be found to contain the remaining data.

If the log is truncated or close to its maximum size an alarm raised to notify the administrator that the log is full.

Assuming that the set of elements being added to the file contains fewer elements than the maximum file capacity we can define operations for adding sets of elements to the current log file. The new elements being added refer to the new elements that are generated during a single interation through the main loop. We have already assumed that a file must be able to contain at least 100 elements, so it is a reasonable assumption that the number of new elements added at any one time is less than the capacity of a file.

$$
\begin{array}{l}
\textbf{schema } \mathit{ValidNewElements} \\
\quad \mathit{RealWorldTimeChanges} \\[4pt]
\quad \mathit{AuditLogC} \\
\quad \mathit{newElements}? : \mathbb{F}\ \mathit{AuditC} \\
\hline
\quad \#\mathit{newElements}? < \mathit{maxLogFileEntries} \\
\quad \mathit{newElements}? \neq \emptyset \\[4pt]
\quad \mathit{oldestLogTimeC}\ \mathit{newElements}? \geq \mathit{nowC} \\
\quad \mathit{newestLogTimeC}\ \mathit{newElements}? \leq \mathit{nowC}' \\
\quad \forall\, i : \mathit{LOGFILEINDEX} \mid i \notin \mathit{freeLogFiles} \bullet \mathit{nowC} \geq \mathit{newestLogTimeC}\ (\mathit{logFiles}\ i)
\end{array}
$$

▷ See: *RealWorldTimeChanges* (p. 40), *AuditLogC* (p. 32), *AuditC* (p. 30), *oldestLogTimeC* (p. 32), *LOGFILEINDEX* (p. 31)

If the set of *newElements* is empty then no change occurs to the log.

$$
\begin{array}{|l}
\textit{AddNoElementsToLog} \\
\hline
\Xi \mathit{AuditLogC} \\
\mathit{newElements?} : \mathbb{F}\ \mathit{AuditC} \\
\hline
\mathit{newElements?} = \varnothing \\
\hline
\end{array}
$$

▷ See: *AuditLogC* (p. 32), *AuditC* (p. 30)

If there is space for the *newElements* in the current file then these elements are added to the current file, and a check is made to determine whether the *auditAlarm* should be raised.

$$
\begin{array}{|l}
\textit{AddElementsToCurrentFile} \\
\hline
\Delta \mathit{AuditLogC} \\
\\
\mathit{ConfigC} \\
\mathit{ValidNewElements} \\
\hline
\#\mathit{newElements?} + \#(\mathit{logFiles}\ \mathit{currentLogFile}) \leq \mathit{maxLogFileEntries} \\
\\
\mathit{numberLogEntries}' = \mathit{numberLogEntries} + \#\mathit{newElements?} \\
\mathit{logFiles}' = \mathit{logFiles} \oplus \{\mathit{currentLogFile} \mapsto \mathit{logFiles}\ \mathit{currentLogFile} \cup \mathit{newElements?}\} \\
\mathit{currentLogFile}' = \mathit{currentLogFile} \\
\mathit{usedLogFiles}' = \mathit{usedLogFiles} \\
\mathit{freeLogFiles}' = \mathit{freeLogFiles} \\
\mathit{logFilesStatus}' = \mathit{logFilesStatus} \\
\\
(\mathit{numberLogEntries}' \geq \mathit{alarmThresholdEntries} \land \mathit{auditAlarmC}' = \mathit{alarming} \\
\qquad \lor \mathit{numberLogEntries}' < \mathit{alarmThresholdEntries} \land \mathit{auditAlarmC}' = \mathit{auditAlarmC}) \\
\hline
\end{array}
$$

▷ See: *AuditLogC* (p. 32), *ConfigC* (p. 27), *ValidNewElements* (p. 48), *alarming* (p. 21)

▷ The value of *alarmThresholdEntries* is imported from *ConfigC*.

If there is insufficient space for the *newElements* in the current log file and there is a free file still available then the current log file is filled using the oldest elements from the set of *newElements*, the remaining *newElements* are added to one of the previously empty files, which becomes the new current file.

AddElementsToNextFileNoTruncate

$\Delta AuditLogC$

$ConfigC$
$ValidNewElements$

$$freeLogFiles \neq \varnothing$$
$$\#newElements? + \#(logFiles\ currentLogFile) > maxLogFileEntries$$

$$numberLogEntries' = numberLogEntries + \#newElements?$$
$$\begin{aligned}&\exists\, elementsInCurrentFile, elementsInNextFile : \mathbb{F}\, AuditC \bullet elementsInCurrentFile \subseteq newElements?\\ &\quad \land \#elementsInCurrentFile + \#(logFiles\ currentLogFile) = maxLogFileEntries\\ &\quad \land elementsInNextFile = newElements? \setminus elementsInCurrentFile\\ &\quad \land oldestLogTimeC\ elementsInNextFile \geq newestLogTimeC\ elementsInCurrentFile\\ &\quad \land logFiles' = logFiles \oplus \{currentLogFile \mapsto logFiles\ currentLogFile \cup elementsInCurrentFile\ ,\\ &\qquad\qquad currentLogFile' \mapsto elementsInNextFile\}\end{aligned}$$

$$currentLogFile' = min\ freeLogFiles$$
$$usedLogFiles' = usedLogFiles \frown \langle currentLogFile' \rangle$$
$$freeLogFiles' = freeLogFiles \setminus \{currentLogFile'\}$$
$$logFilesStatus' = logFilesStatus \oplus \{currentLogFile' \mapsto used\}$$

$$\begin{aligned}&(numberLogEntries' \geq alarmThresholdEntries \land auditAlarmC' = alarming\\ &\quad \lor numberLogEntries' < alarmThresholdEntries \land auditAlarmC' = auditAlarmC)\end{aligned}$$

- ▷ See: *AuditLogC* (p. 32), *ConfigC* (p. 27), *ValidNewElements* (p. 48), *AuditC* (p. 30), *oldestLogTimeC* (p. 32), *used* (p. 32), *alarming* (p. 21)

- ▷ *elementsInCurrentFile* is the subset of *newElements* that will fill the current file.
- ▷ *elementsInNextFile* is the subset of *newElements* that will be written to a new file.
- ▷ The value of *alarmThresholdEntries* is imported from *ConfigC*.

If there is insufficient space for the *newElements* in the current log file and there is not a free file available then the log will require truncation before all the data can be added. The current log file is filled using the oldest elements from the set of *newElements*. The oldest file is emptied and an audit entry recording the truncation is added to this file followed by the remaining *newElements*. The file that was the oldest now becomes the new current file.

$$
\begin{array}{l}
\textit{AddElementsToNextFileWithTruncate} \\
\Delta \textit{AuditLogC} \\
\textit{ConfigC} \\
\textit{ValidNewElements} \\
\hline
\textit{freeLogFiles} = \emptyset \\
\# \textit{newElements}? + \#(\textit{logFiles}\ \textit{currentLogFile}) \geq \textit{maxLogFileEntries} \\
\textit{numberLogEntries}' = \textit{numberLogEntries} + \#\textit{newElements}? - \textit{maxLogFileEntries} + 1 \\
\exists\, \textit{truncElement} : \textit{AuditC};\ \textit{elementsInCurrentFile}, \textit{elementsInNextFile} : \mathbb{F}\ \textit{AuditC} \bullet \\
\quad \textit{truncElement}.\textit{logTime} \in \textit{nowC} \mathrel{..} \textit{nowC}' \\
\quad \land\ \textit{truncElement}.\textit{elementId} = \textit{truncateLogElement} \\
\quad \land\ \textit{truncElement}.\textit{severity} = \textit{critical} \\
\quad \land\ \textit{truncElement}.\textit{user} = \textit{noUser} \\
\quad \land\ \textit{elementsInCurrentFile} \subseteq \textit{newElements}? \\
\quad \land\ \#(\textit{logFiles}\ \textit{currentLogFile}) + \#\textit{elementsInCurrentFile} = \textit{maxLogFileEntries} \\
\quad \land\ \textit{elementsInNextFile} = \textit{newElements}? \setminus \textit{elementsInCurrentFile} \\
\quad \land\ \textit{oldestLogTimeC}\ \textit{elementsInNextFile} \geq \textit{truncElement}.\textit{logTime} \\
\quad \land\ \textit{truncElement}.\textit{logTime} \geq \textit{newestLogTimeC}\ \textit{elementsInCurrentFile} \\
\quad \land\ \textit{logFiles}' = \textit{logFiles} \oplus \{\textit{currentLogFile} \mapsto \textit{logFiles}\ \textit{currentLogFile} \cup \textit{elementsInCurrentFile}, \\
\qquad\qquad \textit{currentLogFile}' \mapsto \textit{elementsInNextFile} \cup \{\textit{truncElement}\}\ \} \\
\textit{currentLogFile}' = \textit{head}\ \textit{usedLogFiles} \\
\textit{usedLogFiles}' = \textit{tail}\ \textit{usedLogFiles} \frown \langle \textit{currentLogFile}' \rangle \\
\textit{freeLogFiles}' = \textit{freeLogFiles} \\
\textit{logFilesStatus}' = \textit{logFilesStatus} \oplus \{\textit{currentLogFile}' \mapsto \textit{used}\} \\
\textit{auditAlarmC}' = \textit{alarming}
\end{array}
$$

▷ See: *AuditLogC* (p. 32), *ConfigC* (p. 27), *ValidNewElements* (p. 48), *AuditC* (p. 30), *truncateLogElement* (p. 28), *critical* (p. 28), *noUser* (p. 30), *oldestLogTimeC* (p. 32), *used* (p. 32), *alarming* (p. 21)

▷ The status of the $\textit{currentLogFile}'$ may change from *archived* to *used* during this operation.

▷ *elementsInCurrentFile* is the subset of *newElements* that will fill the current file.

▷ *elementsInNextFile* is the subset of *newElements* that will be written to a new file.

▷ *truncElement* is the audit element recording the truncation of the log file.

▷ The *truncElement.description* should contain the time range of data truncated from the log. This is not formally stated.

▷ The value of *alarmThresholdEntries* is imported from *ConfigC*.

Combining these gives us the operation of adding a number of elements to the log file.

$$
\begin{array}{ll}
\textit{AddElementsToLogC} \mathrel{\widehat=} & \textit{AddNoElementsToLog} \\
& \lor\ \textit{AddElementsToCurrentFile} \lor \textit{AddElementsToNextFileNoTruncate} \\
& \lor\ \textit{AddElementsToNextFileWithTruncate}
\end{array}
$$

▷ See: *AddNoElementsToLog* (p. 48), *AddElementsToCurrentFile* (p. 49), *AddElementsToNextFileNoTruncate* (p. 49), *AddElementsToNextFileWithTruncate* (p. 50)

5.1.2 Implementation Notes

It should be noted that for implementation purposes only a single element will be added to the log at a time, the following operations are those that are required to be implemented. These deal with truncation and addition separately and then these two problems are brought together to define the full operation.

FD.AuditLog.AddElementToLogFile
FD.AuditLog.AddElementsToLog

An element is only valid for addition into the log if it occured between the last and the next time indicated by the real world. This can be guaranteed by using the current time (from a trusted source) for each element added to the log.

ValidNewElement

$$
\begin{array}{l}
RealWorldTimeChanges \\
\\
AuditLogC \\
newElement? : AuditC \\
\hline
newElement?.logTime \in nowC \,..\, nowC' \\
\forall\, i : LOGFILEINDEX \mid i \notin freeLogFiles \bullet nowC \geq newestLogTimeC\,(logFiles\; i)
\end{array}
$$

▷ See: *RealWorldTimeChanges* (p. 40), *AuditLogC* (p. 32), *AuditC* (p. 30), *LOGFILEINDEX* (p. 31)

If there is room in the current file the new element is added to this.

AddElementToCurrentLogFile

$$
\begin{array}{l}
\Delta AuditLogC \\
\\
ConfigC \\
ValidNewElement \\
\hline
\#(logFiles\; currentLogFile) < maxLogFileEntries \\
\\
numberLogEntries' = numberLogEntries + 1 \\
logFiles' = logFiles \oplus \{currentLogFile \mapsto logFiles\; currentLogFile \cup \{newElement?\}\,\} \\
currentLogFile' = currentLogFile \\
usedLogFiles' = usedLogFiles \\
freeLogFiles' = freeLogFiles \\
logFilesStatus' = logFilesStatus \\
\\
(numberLogEntries' \geq alarmThresholdEntries \land auditAlarmC' = alarming \\
\quad \lor numberLogEntries' < alarmThresholdEntries \land auditAlarmC' = auditAlarmC)
\end{array}
$$

▷ See: *AuditLogC* (p. 32), *ConfigC* (p. 27), *ValidNewElement* (p. 52), *alarming* (p. 21)

▷ The value of *alarmThresholdEntries* is imported from *ConfigC*.

If there is no room in the current file then there must be a free file and this becomes the current file.

$$
\begin{array}{l}
\textbf{schema } \mathit{AddElementToNextLogFile} \\
\Delta \mathit{AuditLogC} \\
\mathit{ConfigC} \\
\mathit{ValidNewElement} \\
\hline
\mathit{freeLogFiles} \neq \emptyset \\
\#(\mathit{logFiles}\ \mathit{currentLogFile}) = \mathit{maxLogFileEntries} \\
\mathit{numberLogEntries}' = \mathit{numberLogEntries} + 1 \\
\mathit{logFiles}' = \mathit{logFiles} \oplus \{\mathit{currentLogFile}' \mapsto \{\mathit{newElement}?\}\ \} \\
\mathit{currentLogFile}' = \mathit{min}\ \mathit{freeLogFiles} \\
\mathit{usedLogFiles}' = \mathit{usedLogFiles} \frown \langle \mathit{currentLogFile}' \rangle \\
\mathit{freeLogFiles}' = \mathit{freeLogFiles} \setminus \{\mathit{currentLogFile}'\} \\
\mathit{logFilesStatus}' = \mathit{logFilesStatus} \oplus \{\mathit{currentLogFile}' \mapsto \mathit{used}\} \\
(\mathit{numberLogEntries}' \geq \mathit{alarmThresholdEntries} \wedge \mathit{auditAlarmC}' = \mathit{alarming} \\
\quad \vee\ \mathit{numberLogEntries}' < \mathit{alarmThresholdEntries} \wedge \mathit{auditAlarmC}' = \mathit{auditAlarmC})
\end{array}
$$

▷ See: *AuditLogC* (p. 32), *ConfigC* (p. 27), *ValidNewElement* (p. 52), *used* (p. 32), *alarming* (p. 21)

▷ The value of *alarmThresholdEntries* is imported from *ConfigC*.

$$\mathit{AddElementToLogFile} \mathrel{\widehat=} \mathit{AddElementToCurrentLogFile} \vee \mathit{AddElementToNextLogFile}$$

▷ See: *AddElementToCurrentLogFile* (p. 52), *AddElementToNextLogFile* (p. 52)

FD.AuditLog.TruncateLog
FD.AuditLog.AddElementsToLog

The log files are truncated by deleting the oldest log file, as there are at least two files, this is not the current file.

$$
\begin{array}{l}
\textbf{schema } \mathit{TruncateLog} \\
\Delta \mathit{AuditLogC} \\
\mathit{RealWorldTimeChanges} \\
\mathit{truncElement}! : \mathit{AuditC} \\
\hline
\mathit{freeLogFiles} = \emptyset \\
\#(\mathit{logFiles}\ \mathit{currentLogFile}) = \mathit{maxLogFileEntries} \\
\mathit{numberLogEntries}' = \mathit{numberLogEntries} - \mathit{maxLogFileEntries} \\
\mathit{logFiles}' = \mathit{logFiles} \oplus \{\mathit{head}\ \mathit{usedLogFiles} \mapsto \emptyset\} \\
\mathit{currentLogFile}' = \mathit{currentLogFile} \\
\mathit{usedLogFiles}' = \mathit{tail}\ \mathit{usedLogFiles} \\
\mathit{freeLogFiles}' = \mathit{freeLogFiles} \cup \{\mathit{head}\ \mathit{usedLogFiles}\} \\
\mathit{logFilesStatus}' = \mathit{logFilesStatus} \oplus \{\mathit{head}\ \mathit{usedLogFiles} \mapsto \mathit{free}\} \\
\mathit{auditAlarmC}' = \mathit{alarming} \\
\mathit{truncElement}!.\mathit{logTime} \in \mathit{nowC} \mathrel{..} \mathit{nowC}' \\
\mathit{truncElement}!.\mathit{elementId} = \mathit{truncateLogElement} \\
\mathit{truncElement}!.\mathit{severity} = \mathit{critical} \\
\mathit{truncElement}!.\mathit{user} = \mathit{noUser}
\end{array}
$$

▷ See: *AuditLogC* (p. 32), *RealWorldTimeChanges* (p. 40), *AuditC* (p. 30), *free* (p. 32), *alarming* (p. 21), *truncateLogElement* (p. 28), *critical* (p. 28), *noUser* (p. 30)

▷ The $truncElement!.description$ should contain the time range of data truncated from the log. This is not formally stated.

▷ $truncElement!$ is the audit element recording this truncation.

$$
\begin{array}{|l}
\textit{TruncateLogNotRequired} \\
\hline
\Xi AuditLogC \\
\hline
freeLogFiles \neq \varnothing \\
\lor \#(logFiles\ currentLogFile) < maxLogFileEntries \\
\hline
\end{array}
$$

▷ See: *AuditLogC* (p. 32)

FD.AuditLog.AddElementToLog
FD.AuditLog.AddElementsToLog

$$
\begin{aligned}
AddElementToLogC \mathrel{\widehat{=}} & ((TruncateLog[theElement'/truncElement!] \mathbin{⨾} AddElementToLogFile[theElement/newElement?]) \\
& \quad \lor TruncateLogNotRequired) \\
& \qquad \mathbin{⨾} AddElementToLogFile
\end{aligned}
$$

▷ See: *TruncateLog* (p. 53), *AddElementToLogFile* (p. 53), *TruncateLogNotRequired* (p. 54)

We claim that adding new entries for the log one by one has the same effect as adding them as a set. All that is required is that the elements are added in chronological order. This is stated formally as follows:

$$
\begin{aligned}
& [AddElementsToLogC \mid newElements? \neq \varnothing] \equiv \\
& \qquad [ValidNewElements; \Delta AuditLogC; ConfigC \mid \\
& \qquad\quad \exists\, newElement? : AuditC; remainingElements? : \mathbb{F}\, AuditC \bullet \\
& \qquad\quad newElement? \in newElements? \land newElement?.logTime = oldestLogTimeC\ newElements? \\
& \qquad\quad \land remainingElements? = newElements? \setminus \{newElement?\} \\
& \qquad\quad \land AddElementToLogC \mathbin{⨾} AddElementsToLogC[remainingElements?/newElements?]\,]
\end{aligned}
$$

▷ $newElement?$ is the oldest element in $newElements?$, while $remainingElements?$ are the elements that are left in $newElements?$ once $newElement?$ is removed.

▷ The above states that adding $newElement?$ using the operation *AddElementToLogC* followed by adding $remainingElements?$ using the operation *AddElementsToLogC* is equivalent to adding the set $newElements?$ using the operation *AddElementsToLogC*.

5.1.3 Archiving the Log

FD.AuditLog.ArchiveLog
FS.AuditLog.ArchiveLog

When we archive the log an audit element is added to the log and an archive is generated which can be written to floppy.

We only archive complete log files, upto the maximum capacity of the archive media.

This activity does not clear the log since a check will be made to ensure the archive was successful before clearing the log. It marks all files that are archived so that they can be recognised for clearing if the export of the archive log succeeds.

$$
\begin{array}{l}
\textbf{schema } \mathit{DetermineArchiveLog} \\
\Delta \mathit{AuditLogC} \\
\\
\mathit{RealWorldTimeChanges} \\
\mathit{AdminTokenC} \\
\mathit{ConfigC} \\
\\
\mathit{archive}! : \mathbb{F}\ \mathit{AuditC} \\
\mathit{archiveElement}! : \mathit{AuditC} \\
\hline
\exists \mathit{archivedFiles} : \mathbb{F}\ \mathit{LOGFILEINDEX} \bullet \\
\quad \mathit{archivedFiles} = \{ i : \mathit{LOGFILEINDEX} \mid i \in \operatorname{ran}((1 \mathrel{..} \mathit{maxNumberArchivableFiles}) \lhd \mathit{usedLogFiles}) \\
\qquad\qquad \land \#(\mathit{logFiles}\ i) = \mathit{maxLogFileEntries} \} \\
\quad \land \mathit{archive}! = \bigcup \{ i : \mathit{archivedFiles} \bullet \mathit{logFiles}\ i \} \\
\\
\quad \land \mathit{logFilesStatus}' = \mathit{logFilesStatus} \oplus \{ i : \mathit{archivedFiles} \bullet i \mapsto \mathit{archived} \} \\
\\
\mathit{usedLogFiles}' = \mathit{usedLogFiles} \\
\mathit{freeLogFiles}' = \mathit{freeLogFiles} \\
\mathit{logFiles}' = \mathit{logFiles} \\
\mathit{numberLogEntries}' = \mathit{numberLogEntries} \\
\\
\mathit{archiveElement}!.\mathit{logTime} \in \mathit{nowC} \mathrel{..} \mathit{nowC}' \\
\mathit{archiveElement}!.\mathit{elementId} = \mathit{archiveLogElement} \\
\mathit{archiveElement}!.\mathit{severity} = \mathit{information} \\
\mathit{archiveElement}!.\mathit{user} = \mathit{extractUser}\ \mathit{currentAdminTokenC} \\
\\
\mathit{auditAlarmC}' = \mathit{auditAlarmC}
\end{array}
$$

- ▷ See: *AuditLogC* (p. 32), *RealWorldTimeChanges* (p. 40), *AdminTokenC* (p. 36), *ConfigC* (p. 27), *AuditC* (p. 30), *LOGFILEINDEX* (p. 31), *archived* (p. 32), *archiveLogElement* (p. 28), *information* (p. 28), *extractUser* (p. 30)

- ▷ *archivedFiles* is the set of files that will be archived, these are all full files from the front of the list of *usedLogFiles*.
- ▷ The *archiveElement*! is the audit entry recording the construction of an archive.
- ▷ The *archiveElement*!.*description* should contain the time range of data selected for archive from the log. This is not formally stated.
- ▷ The Id of the current administrator is imported from *AdminTokenC*.
- ▷ The *alarmThresholdEntries* is imported from *ConfigC*.

Other elements may be added to the log once the archive has been determined.

$$
\begin{array}{l}
\mathit{ArchiveLogC} \mathrel{\widehat=} (\mathit{DetermineArchiveLog} \mathbin{⨾} \\
\qquad\qquad [\mathit{AddElementsToLogC}; \mathit{archiveElement}! : \mathit{AuditC} \mid \mathit{archiveElement}! \in \mathit{newElements}?]) \\
\qquad\qquad\quad \setminus (\mathit{archiveElement}!)
\end{array}
$$

- ▷ See: *DetermineArchiveLog* (p. 55), *AddElementsToLogC* (p. 51), *AuditC* (p. 30)

5.1.4 Clearing the Log

FD.AuditLog.ClearLog
FS.AuditLog.ClearLog

The log should only be cleared if it can be verified that an archive has been created of the data that is about to be cleared.

The action of clearing the log will replace all files marked as archived by empty files.

$$
\begin{array}{l}
\textbf{schema } \mathit{ClearLogEntries} \\
\quad \mathit{RealWorldTimeChanges} \\[4pt]
\quad \mathit{ConfigC} \\
\quad \mathit{AdminTokenC} \\
\quad \Delta \mathit{AuditLogC} \\[4pt]
\quad \mathit{archiveCompleteElement}! : \mathit{AuditC} \\
\hline
\quad \exists\, \mathit{archivedFiles} : \mathbb{F}\, \mathit{LOGFILEINDEX} \bullet \\
\qquad \mathit{archivedFiles} = \mathrm{dom}(\mathit{logFilesStatus} \rhd \{\mathit{archived}\}) \\
\qquad \land\ \mathit{logFilesStatus}' = \mathit{logFilesStatus} \oplus (\mathit{archivedFiles} \times \{\mathit{free}\}) \\[4pt]
\qquad \land\ \mathit{usedLogFiles}' = \mathit{usedLogFiles} \upharpoonright (\mathit{LOGFILEINDEX} \setminus \mathit{archivedFiles}) \\
\qquad \land\ \mathit{freeLogFiles}' = \mathit{freeLogFiles} \cup \mathit{archivedFiles} \\
\qquad \land\ \mathit{logFiles}' = \mathit{logFiles} \oplus (\mathit{archivedFiles} \times \{\varnothing\}) \\
\qquad \land\ \mathit{numberLogEntries}' = \mathit{numberLogEntries} - \mathit{maxLogFileEntries} * \#\mathit{archivedFiles} \\[4pt]
\quad \mathit{archiveCompleteElement}!.\mathit{logTime} \in \mathit{nowC} \mathrel{..} \mathit{nowC}' \\
\quad \mathit{archiveCompleteElement}!.\mathit{elementId} = \mathit{archiveCompleteElement} \\
\quad \mathit{archiveCompleteElement}!.\mathit{severity} = \mathit{information} \\
\quad \mathit{archiveCompleteElement}!.\mathit{description} = \mathit{noDescription} \\
\quad \mathit{archiveCompleteElement}!.\mathit{user} = \mathit{extractUser}\ \mathit{currentAdminTokenC} \\[4pt]
\quad (\mathit{numberLogEntries}' < \mathit{alarmThresholdEntries} \land \mathit{auditAlarmC}' = \mathit{silent} \\
\qquad \lor\ \mathit{numberLogEntries} \geq \mathit{alarmThresholdEntries} \land \mathit{auditAlarmC}' = \mathit{alarming})
\end{array}
$$

▷ See: *RealWorldTimeChanges* (p. 40), *ConfigC* (p. 27), *AdminTokenC* (p. 36), *AuditLogC* (p. 32), *archiveCompleteElement* (p. 28), *AuditC* (p. 30), *LOGFILEINDEX* (p. 31), *archived* (p. 32), *free* (p. 32), *information* (p. 28), *noDescription* (p. 30), *extractUser* (p. 30), *silent* (p. 21), *alarming* (p. 21)

Other entries may be added to the log following clearing of the archived entries.

$$
\begin{array}{l}
\mathit{ClearLogC} \mathrel{\widehat{=}} (\mathit{ClearLogEntries} \mathbin{;} \\
\qquad [\mathit{AddElementsToLogC};\ \mathit{archiveCompleteElement}! : \mathit{AuditC} \mid \\
\qquad\qquad \mathit{archiveCompleteElement}! \in \mathit{newElements}?]) \\
\qquad\quad \setminus (\mathit{archiveCompleteElement}!)
\end{array}
$$

▷ See: *ClearLogEntries* (p. 56), *AddElementsToLogC* (p. 51), *archiveCompleteElement* (p. 28), *AuditC* (p. 30)

FD.AuditLog.CancelArchive

If the archive fails then all record of the archive should be removed from the status of the log files.

$$
\begin{array}{l}
\textit{CancelArchiveIndication} \\
\hline
\textit{RealWorldTimeChanges} \\[1ex]
\textit{ConfigC} \\
\textit{AdminTokenC} \\
\textit{RealWorldChangesC} \\
\Delta \textit{AuditLogC} \\
\hline
\exists \textit{archivedFiles} : \mathbb{F}\ \textit{LOGFILEINDEX} \bullet \\
\quad \textit{archivedFiles} = \mathrm{dom}(\textit{logFilesStatus} \rhd \{\textit{archived}\}) \\
\quad \land\ \textit{logFilesStatus}' = \textit{logFilesStatus} \oplus (\textit{archivedFiles} \times \{\textit{used}\}) \\[1ex]
\textit{usedLogFiles}' = \textit{usedLogFiles} \\
\textit{freeLogFiles}' = \textit{freeLogFiles} \\
\textit{logFiles}' = \textit{logFiles} \\
\textit{numberLogEntries}' = \textit{numberLogEntries}
\end{array}
$$

▷ See: *RealWorldTimeChanges* (p. 40), *ConfigC* (p. 27), *AdminTokenC* (p. 36), *RealWorldChangesC* (p. 40), *AuditLogC* (p. 32), *LOGFILEINDEX* (p. 31), *archived* (p. 32), *used* (p. 32)

▷ The log entry associated with this is created at the system level as it may incorporate the reason for failure

Other elements may be added to the log following cancellation of the archive indication.

$$\textit{CancelArchive} \mathrel{\widehat{=}} (\textit{CancelArchiveIndication} \mathbin{⨾} \textit{AddElementsToLogC})$$

▷ See: *CancelArchiveIndication* (p. 56), *AddElementsToLogC* (p. 51)

5.1.5 Auditing Changes

FD.AuditLog.LogChange
FS.AuditLog.LogChange

TIS adds audit entries whenever any of the following changes occurs:

- The door is opened or closed.
- The door is latched or unlatched.
- The alarm starts alarming or becomes silenced.
- The audit alarm starts alarming or becomes silenced.
- The text displayed on the display changes.
- The message text displayed on the screen changes.
- The log is truncated (this has already been covered in Section 5.1).

$$\textit{DOORCHANGE\_ELEMENTS} == \{\textit{doorOpenedElement}, \textit{doorClosedElement}\}$$

▷ See: *doorOpenedElement* (p. 28), *doorClosedElement* (p. 28)

▷ The *doorOpenedElement* and *doorClosedElement* are the audit entries recording that the door has been opened and closed respectively.

Audit entries associated with changes to the door do not specify a user, nor do they include additional details.

$$
\begin{array}{l}
\textit{AuditDoorC} \\
\hline
\Delta \textit{DoorLatchAlarmC} \\
\\
\textit{RealWorldTimeChanges} \\
\textit{newElements}? : \mathbb{F}\, \textit{AuditC} \\
\hline
\forall \textit{newElement} : \textit{AuditC} \mid \\
\quad \textit{newElement} \in \textit{newElements}? \land \textit{newElement}.\textit{elementId} \in \textit{DOORCHANGE\_ELEMENTS} \bullet \\
\qquad \textit{newElement}.\textit{logTime} \in \textit{nowC} \mathrel{..} \textit{nowC}' \\
\qquad \land\ \textit{newElement}.\textit{user} = \textit{noUser} \\
\qquad \land\ \textit{newElement}.\textit{severity} = \textit{information} \\
\qquad \land\ \textit{newElement}.\textit{description} = \textit{noDescription} \\
\\
(\textit{currentDoorC} \neq \textit{currentDoorC}' \land \textit{currentDoorC}' = \textit{open} \\
\quad \Leftrightarrow (\exists_1\, \textit{element} : \textit{AuditC} \bullet \textit{element} \in \textit{newElements}? \land \textit{element}.\textit{elementId} = \textit{doorOpenedElement} \\
\qquad \land\ \textit{auditTypes}\ \textit{newElements}? \cap \textit{DOORCHANGE\_ELEMENTS} = \{\textit{doorOpenedElement}\})) \\
(\textit{currentDoorC}' \neq \textit{currentDoorC} \land \textit{currentDoorC}' = \textit{closed} \\
\quad \Leftrightarrow (\exists_1\, \textit{element} : \textit{AuditC} \bullet \textit{element} \in \textit{newElements}? \land \textit{element}.\textit{elementId} = \textit{doorClosedElement} \\
\qquad \land\ \textit{auditTypes}\ \textit{newElements}? \cap \textit{DOORCHANGE\_ELEMENTS} = \{\textit{doorClosedElement}\}))
\end{array}
$$

▷ See: *DoorLatchAlarmC* (p. 35), *RealWorldTimeChanges* (p. 40), *AuditC* (p. 30), *DOORCHANGE_ELEMENTS* (p. 57), *noUser* (p. 30), *information* (p. 28), *noDescription* (p. 30), *open* (p. 21), *doorOpenedElement* (p. 28), *closed* (p. 21), *doorClosedElement* (p. 28)

$$\textit{LATCHCHANGE\_ELEMENTS} == \{\textit{latchLockedElement}, \textit{latchUnlockedElement}\}$$

▷ See: *latchLockedElement* (p. 28), *latchUnlockedElement* (p. 28)

▷ The *latchLockedElement* and *latchUnlockedElement* are the audit entries recording that the latch has been locked and unlocked respectively.

Audit entries associated with changes to the latch do not specify a user, nor do they include additional details.

$$
\begin{array}{l}
\textit{AuditLatchC} \\
\hline
\Delta \textit{DoorLatchAlarmC} \\
\\
\textit{RealWorldTimeChanges} \\
\textit{newElements}? : \mathbb{F}\, \textit{AuditC} \\
\hline
\forall \textit{newElement} : \textit{AuditC} \mid \\
\quad \textit{newElement} \in \textit{newElements}? \land \textit{newElement}.\textit{elementId} \in \textit{LATCHCHANGE\_ELEMENTS} \bullet \\
\qquad \textit{newElement}.\textit{logTime} \in \textit{nowC} \mathrel{..} \textit{nowC}' \\
\qquad \land\ \textit{newElement}.\textit{user} = \textit{noUser} \\
\qquad \land\ \textit{newElement}.\textit{severity} = \textit{information} \\
\qquad \land\ \textit{newElement}.\textit{description} = \textit{noDescription} \\
\\
(\textit{currentLatchC}' \neq \textit{currentLatchC} \land \textit{currentLatchC}' = \textit{locked} \\
\quad \Leftrightarrow (\exists_1\, \textit{element} : \textit{AuditC} \bullet \textit{element} \in \textit{newElements}? \land \textit{element}.\textit{elementId} = \textit{latchLockedElement} \\
\qquad \land\ \textit{auditTypes}\ \textit{newElements}? \cap \textit{LATCHCHANGE\_ELEMENTS} = \{\textit{latchLockedElement}\})) \\
(\textit{currentLatchC}' \neq \textit{currentLatchC} \land \textit{currentLatchC}' = \textit{unlocked} \\
\quad \Leftrightarrow (\exists_1\, \textit{element} : \textit{AuditC} \bullet \textit{element} \in \textit{newElements}? \land \textit{element}.\textit{elementId} = \textit{latchUnlockedElement} \\
\qquad \land\ \textit{auditTypes}\ \textit{newElements}? \cap \textit{LATCHCHANGE\_ELEMENTS} = \{\textit{latchUnlockedElement}\}))
\end{array}
$$

▷ See: *DoorLatchAlarmC* (p. 35), *RealWorldTimeChanges* (p. 40), *AuditC* (p. 30), *LATCHCHANGE_ELEMENTS* (p. 58), *noUser* (p. 30), *information* (p. 28), *noDescription* (p. 30), *locked* (p. 21), *latchLockedElement* (p. 28), *unlocked* (p. 21), *latchUnlockedElement* (p. 28)

$$ALARMCHANGE\_ELEMENTS == \{alarmSilencedElement, alarmRaisedElement\}$$

▷ See: *alarmSilencedElement* (p. 28), *alarmRaisedElement* (p. 28)

▷ The *alarmSilencedElement* and *alarmRaisedElement* are the audit entries recording that the alarm has been silenced and raised respectively.

Audit entries associated with changes to the alarm do not specify a user, nor do they include additional details.

AuditAlarmC

$$\begin{array}{l}
\Delta DoorLatchAlarmC \\
RealWorldTimeChanges \\
newElements? : \mathbb{F}\ AuditC \\
\hline
\forall newElement : AuditC \mid \\
\quad newElement \in newElements? \land newElement.elementId \in ALARMCHANGE_ELEMENTS \bullet \\
\quad\quad newElement.logTime \in nowC \mathrel{..} nowC' \\
\quad\quad \land newElement.user = noUser \\
\quad\quad \land newElement.description = noDescription \\
(doorAlarmC \neq doorAlarmC' \land doorAlarmC' = alarming \\
\quad \Leftrightarrow (\exists_1 element : AuditC \bullet element \in newElements? \land element.elementId = alarmRaisedElement \\
\quad\quad \land element.severity = critical \\
\quad\quad \land auditTypes\ newElements? \cap ALARMCHANGE_ELEMENTS = \{alarmRaisedElement\})) \\
(doorAlarmC \neq doorAlarmC' \land doorAlarmC' = silent \\
\quad \Leftrightarrow (\exists_1 element : AuditC \bullet element \in newElements? \land element.elementId = alarmSilencedElement \\
\quad\quad \land element.severity = information \\
\quad\quad \land auditTypes\ newElements? \cap ALARMCHANGE_ELEMENTS = \{alarmSilencedElement\}))
\end{array}$$

▷ See: *DoorLatchAlarmC* (p. 35), *RealWorldTimeChanges* (p. 40), *AuditC* (p. 30), *ALARMCHANGE_ELEMENTS* (p. 59), *noUser* (p. 30), *noDescription* (p. 30), *alarming* (p. 21), *alarmRaisedElement* (p. 28), *critical* (p. 28), *silent* (p. 21), *alarmSilencedElement* (p. 28), *information* (p. 28)

$$AUDITALARMCHANGE\_ELEMENTS == \{auditAlarmSilencedElement, auditAlarmRaisedElement\}$$

▷ See: *auditAlarmSilencedElement* (p. 28), *auditAlarmRaisedElement* (p. 28)

▷ The *auditAlarmSilencedElement* and *auditAlarmRaisedElement* are the audit entries recording that the audit log overflow warning alarm has been silenced and raised respectively.

Audit entries associated with changes to the alarm do not specify a user, nor do they include additional details.

$$
\begin{array}{l}
\textbf{AuditLogAlarmC} \\
\hline
\Delta AuditLogC \\[4pt]
RealWorldTimeChanges \\
newElements? : \mathbb{F}\, AuditC \\
\hline
\forall newElement : AuditC \mid \\
\quad newElement \in newElements? \land newElement.elementId \in AUDITALARMCHANGE\_ELEMENTS \bullet \\
\quad\quad newElement.logTime \in nowC \mathrel{..} nowC' \\
\quad\quad \land newElement.user = noUser \\
\quad\quad \land newElement.description = noDescription \\[4pt]
(auditAlarmC \neq auditAlarmC' \land auditAlarmC' = alarming \\
\quad \Leftrightarrow (\exists_1 element : AuditC \bullet element \in newElements? \land element.elementId = auditAlarmRaisedElement \\
\quad\quad \land element.severity = warning \\
\quad\quad \land auditTypes\ newElements? \cap AUDITALARMCHANGE\_ELEMENTS = \{auditAlarmRaisedElement\})) \\
(auditAlarmC \neq auditAlarmC' \land auditAlarmC' = silent \\
\quad \Leftrightarrow (\exists_1 element : AuditC \bullet element \in newElements? \land element.elementId = auditAlarmSilencedElement \\
\quad\quad \land element.severity = information \\
\quad\quad \land auditTypes\ newElements? \cap AUDITALARMCHANGE\_ELEMENTS = \{auditAlarmSilencedElement\})) \\
\hline
\end{array}
$$

▷ See: *AuditLogC* (p. 32), *RealWorldTimeChanges* (p. 40), *AuditC* (p. 30), *AUDITALARMCHANGE_ELEMENTS* (p. 59), *noUser* (p. 30), *noDescription* (p. 30), *alarming* (p. 21), *auditAlarmRaisedElement* (p. 28), *warning* (p. 28), *silent* (p. 21), *auditAlarmSilencedElement* (p. 28), *information* (p. 28)

Audit entries recording that the display has changed are of type *displayChangedElement*. Audit entries associated with changes to the display do not specify a user, the additional details will give the new displayed text, this is not stated formally.

$$
\begin{array}{l}
\textbf{AuditDisplayC} \\
\hline
currentDisplayC, currentDisplayC' : DISPLAYMESSAGE \\[4pt]
RealWorldTimeChanges \\
newElements? : \mathbb{F}\, AuditC \\
\hline
\forall newElement : AuditC \mid \\
\quad newElement \in newElements? \land newElement.elementId = displayChangedElement \bullet \\
\quad\quad newElement.logTime \in nowC \mathrel{..} nowC' \\
\quad\quad \land newElement.user = noUser \\
\quad\quad \land newElement.severity = information \\[4pt]
(currentDisplayC' \neq currentDisplayC \\
\quad \Leftrightarrow (\exists_1 element : AuditC \bullet element \in newElements? \land element.elementId = displayChangedElement)) \\
\hline
\end{array}
$$

▷ See: *DISPLAYMESSAGE* (p. 22), *RealWorldTimeChanges* (p. 40), *AuditC* (p. 30), *displayChangedElement* (p. 28), *noUser* (p. 30), *information* (p. 28)

Audit entries recording that the screen message has changed are of type *screenChangedElement*. Audit entries associated with changes to the screen message do not specify a user, the additional details will give the new displayed text, this is not stated formally.

$$
\begin{array}{l}
\textit{AuditScreenC} \\
\hline
\Delta \textit{ScreenC} \\
\\
\textit{RealWorldTimeChanges} \\
\textit{newElements}? : \mathbb{F}\ \textit{AuditC} \\
\hline
\forall\ \textit{newElement} : \textit{AuditC} \mid \\
\quad \textit{newElement} \in \textit{newElements}? \land \textit{newElement}.\textit{elementId} = \textit{screenChangedElement} \bullet \\
\qquad \textit{newElement}.\textit{logTime} \in \textit{nowC} \mathrel{..} \textit{nowC}' \\
\qquad \land\ \textit{newElement}.\textit{user} = \textit{noUser} \\
\qquad \land\ \textit{newElement}.\textit{severity} = \textit{information} \\
\\
(\textit{screenMsgC}' \neq \textit{screenMsgC} \\
\quad \Leftrightarrow (\exists_1\ \textit{element} : \textit{AuditC} \bullet \textit{element} \in \textit{newElements}? \land \textit{element}.\textit{elementId} = \textit{screenChangedElement})) \\
\hline
\end{array}
$$

▷ See: *ScreenC* (p. 36), *RealWorldTimeChanges* (p. 40), *AuditC* (p. 30), *screenChangedElement* (p. 28), *noUser* (p. 30), *information* (p. 28)

▷ The *screenChangedElement* is the audit entry recording that the screen has changed.

$$\textit{LogChangeC} \mathrel{\widehat{=}} \textit{AuditAlarmC} \land \textit{AuditLatchC} \land \textit{AuditDoorC} \land \textit{AuditLogAlarmC} \land \textit{AuditScreenC} \land \textit{AuditDisplayC}$$

▷ See: *AuditAlarmC* (p. 59), *AuditLatchC* (p. 58), *AuditDoorC* (p. 58), *AuditLogAlarmC* (p. 59), *AuditScreenC* (p. 60), *AuditDisplayC* (p. 60)

5.2 Updating System Statistics

FD.Stats.Update
FS.Stats.Update

System statistics are updated as actions that are being monitored for the statistics occur.

We provide operations to increment the count of each of the events being monitored.

$$
\begin{array}{l}
\textit{AddSuccessfulEntryToStatsC} \\
\hline
\Delta \textit{StatsC} \\
\hline
\textit{failEntryC}' = \textit{failEntryC} \\
\textit{successEntryC}' = \textit{successEntryC} + 1 \\
\textit{failBioC}' = \textit{failBioC} \\
\textit{successBioC}' = \textit{successBioC} \\
\hline
\end{array}
$$

▷ See: *StatsC* (p. 34)

$$
\begin{array}{l}
\textit{AddFailedEntryToStatsC} \\
\hline
\Delta \textit{StatsC} \\
\hline
\textit{failEntryC}' = \textit{failEntryC} + 1 \\
\textit{successEntryC}' = \textit{successEntryC} \\
\textit{failBioC}' = \textit{failBioC} \\
\textit{successBioC}' = \textit{successBioC} \\
\hline
\end{array}
$$

▷ See: *StatsC* (p. 34)

AddSuccessfulBioCheckToStatsC

$$\Delta StatsC$$

$$failEntryC' = failEntryC$$
$$successEntryC' = successEntryC$$
$$failBioC' = failBioC$$
$$successBioC' = successBioC + 1$$

▷ See: *StatsC* (p. 34)

AddFailedBioCheckToStatsC

$$\Delta StatsC$$

$$failEntryC' = failEntryC$$
$$successEntryC' = successEntryC$$
$$failBioC' = failBioC + 1$$
$$successBioC' = successBioC$$

▷ See: *StatsC* (p. 34)

5.3 Updating Certificate Store

FD.CertificateStore.Update

The certificate store needs to be updated to increment the next available serial number whenever an authorisation certificate is issued.

UpdateCertificateStore

$$\Delta CertificateStore$$

$$nextSerialNumber' = nextSerialNumber + 1$$

▷ See: *CertificateStore* (p. 34)

5.4 Operating the Door, Latch and Alarm

FD.Latch.UpdateInternalLatch

FD.Door.UnlockDoor *FD.Interface.TISPoll*
FD.Door.LockDoor

The state of the latch depends on whether the latch timout has expired or not.

The latch is locked if the timout has expired.

LatchTimeoutExpired

$$\Delta DoorLatchAlarmC$$

$$currentTimeC \geq latchTimeoutC$$

$$currentLatchC' = locked$$
$$currentTimeC' = currentTimeC$$
$$latchTimeoutC' = latchTimeoutC$$
$$alarmTimeoutC' = alarmTimeoutC$$
$$currentDoorC' = currentDoorC$$
$$doorAlarmC' = doorAlarmC$$

▷ See: *DoorLatchAlarmC* (p. 35), *locked* (p. 21)

The latch is unlocked if the timeout has not expired.

$$
\begin{array}{|l}
\textit{LatchTimeoutNotExpired} \\
\Delta \textit{DoorLatchAlarmC} \\
\hline
\textit{latchTimeoutC} > \textit{currentTimeC} \\
\\
\textit{currentLatchC}' = \textit{unlocked} \\
\textit{currentTimeC}' = \textit{currentTimeC} \\
\textit{latchTimeoutC}' = \textit{latchTimeoutC} \\
\textit{alarmTimeoutC}' = \textit{alarmTimeoutC} \\
\textit{currentDoorC}' = \textit{currentDoorC} \\
\textit{doorAlarmC}' = \textit{doorAlarmC} \\
\hline
\end{array}
$$

▷ See: *DoorLatchAlarmC* (p. 35), *unlocked* (p. 21)

$$\textit{UpdateInternalLatch} \mathrel{\widehat{=}} \textit{LatchTimeoutExpired} \lor \textit{LatchTimeoutNotExpired}$$

▷ See: *LatchTimeoutExpired* (p. 62), *LatchTimeoutNotExpired* (p. 63)

FD.Latch.UpdateInternalAlarm	
FD.Door.UnlockDoor	*FD.Interface.TISPoll*
FD.Door.LockDoor	

The state of the alarm depends on the state of the door, the state of the latch and whether the alarm timout has expired or not.

If the door is open, latch is locked and the alarm timout has expired then the alarm is raised.

$$
\begin{array}{|l}
\textit{RaiseAlarm} \\
\Delta \textit{DoorLatchAlarmC} \\
\hline
\textit{currentDoorC} = \textit{open} \\
\textit{currentLatchC} = \textit{locked} \\
\textit{currentTimeC} \geq \textit{alarmTimeoutC} \\
\\
\textit{doorAlarmC}' = \textit{alarming} \\
\textit{currentLatchC}' = \textit{currentLatchC} \\
\textit{latchTimeoutC}' = \textit{latchTimeoutC} \\
\textit{alarmTimeoutC}' = \textit{alarmTimeoutC} \\
\textit{currentTimeC}' = \textit{currentTimeC} \\
\textit{currentDoorC}' = \textit{currentDoorC} \\
\hline
\end{array}
$$

▷ See: *DoorLatchAlarmC* (p. 35), *open* (p. 21), *locked* (p. 21), *alarming* (p. 21)

If the door closed, or the latch is unlocked or the alarm timout has not expired then the alarm is silenced.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SilenceAlarm}} \\
\Delta DoorLatchAlarmC \\
\hline
currentDoorC = closed \\
\lor currentLatchC = unlocked \\
\lor currentTimeC < alarmTimeoutC \\
\\
doorAlarmC' = alarming \\
currentLatchC' = currentLatchC \\
latchTimeoutC' = latchTimeoutC \\
alarmTimeoutC' = alarmTimeoutC \\
currentTimeC' = currentTimeC \\
currentDoorC' = currentDoorC \\
\hline
\end{array}
$$

▷ See: *DoorLatchAlarmC* (p. 35), *closed* (p. 21), *unlocked* (p. 21), *alarming* (p. 21)

$UpdateInternalAlarm \mathrel{\widehat=} RaiseAlarm \lor SilenceAlarm$

▷ See: *RaiseAlarm* (p. 63), *SilenceAlarm* (p. 63)

FD.Door.UnlockDoor
FS.Door.UnlockDoor

When the door is unlatched the timeouts on the door latch and alarm are set to cause the door to be latched again in the future.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{SetUnlockDoorTimeouts}} \\
\Delta DoorLatchAlarmC \\
ConfigC \\
\hline
currentLatchC' = currentLatchC \\
currentTimeC' = currentTimeC \\
latchTimeoutC' = currentTimeC + latchUnlockDurationC \\
alarmTimeoutC' = currentTimeC + latchUnlockDurationC + alarmSilentDurationC \\
currentDoorC' = currentDoorC \\
doorAlarmC' = doorAlarmC \\
\hline
\end{array}
$$

▷ See: *DoorLatchAlarmC* (p. 35), *ConfigC* (p. 27)

▷ *latchUnlockDurationC* and *alarmSilentDurationC* are imported from *ConfigC*.

Once the timeouts have been reset the latch and alarm must be updated.

$UnlockDoorC \mathrel{\widehat=} SetUnlockDoorTimeouts \fatsemi UpdateInternalLatch \fatsemi UpdateInternalAlarm$

▷ See: *SetUnlockDoorTimeouts* (p. 64), *UpdateInternalLatch* (p. 63), *UpdateInternalAlarm* (p. 64)

FD.Door.LockDoor
FS.Door.LockDoor

Sometimes the door needs to be explicitly latched by TIS, when this occurs the timeouts on the door latch and alarm are reset. Resetting the timeouts to the current time will ensure that the alarm will sound if there is a breach of security, this will occur through checks on the alarm timeout.

$$
\begin{array}{|l}
\textit{SetLockDoorTimeouts} \\
\hline
\Delta \mathit{DoorLatchAlarmC} \\
\hline
\mathit{currentLatchC}' = \mathit{currentLatchC} \\
\mathit{currentTimeC}' = \mathit{currentTimeC} \\
\mathit{latchTimeoutC}' = \mathit{currentTimeC} \\
\mathit{alarmTimeoutC}' = \mathit{currentTimeC} \\
\mathit{currentDoorC}' = \mathit{currentDoorC} \\
\mathit{doorAlarmC}' = \mathit{doorAlarmC} \\
\hline
\end{array}
$$

▷ See: *DoorLatchAlarmC* (p. 35)

Once the timeouts have been reset the latch and alarm must be updated.

$$\mathit{LockDoorC} \mathrel{\widehat=} \mathit{SetLockDoorTimeouts} \mathbin{;} \mathit{UpdateInternalLatch} \mathbin{;} \mathit{UpdateInternalAlarm}$$

▷ See: *SetLockDoorTimeouts* (p. 65), *UpdateInternalLatch* (p. 63), *UpdateInternalAlarm* (p. 64)

5.5 Certificate Operations

5.5.1 Validating Certificates

FD.Certificate.SignedOK
FS.Certificate.Validate

When a certificate is checked in the context of a key store it is only acceptable if the certificate issuer is known to the key store and the signature can be verified by the key store.

A certificate must have been issued by a known issuer.

$$
\begin{array}{|l}
\textit{CertIssuerKnownC} \\
\hline
\mathit{KeyStoreC} \\
\mathit{CertificateContents} \\
\hline
\mathit{keyMatchingIssuer}\ \mathit{idC}.\mathit{issuerC}.\mathit{id} \neq \mathit{nil} \\
\hline
\end{array}
$$

▷ See: *KeyStoreC* (p. 33), *CertificateContents* (p. 16)

A certificate must have been signed by the issuer.

$$
\begin{array}{|l}
\textit{CertOKC} \\
\hline
\mathit{CertIssuerKnownC} \\
\mathit{RawCertificate} \\
\hline
(\mathit{mechanism}, \mathit{digest}\ \mathit{mechanism}\ \mathit{data}, \mathit{signature}) \\
\qquad \mathsf{isVerifiedBy}\ (\mathit{the}\ (\mathit{keyMatchingIssuer}\ \mathit{idC}.\mathit{issuerC}.\mathit{id})) \\
\hline
\end{array}
$$

▷ See: *CertIssuerKnownC* (p. 65), *RawCertificate* (p. 15), *digest* (p. 15)

FD.Certificate.AuthCertSignedOK
FS.Certificate.Validate

In addition the Authorisation certificate must have been issued by this ID station; we make the assumption that a single ID station protects an enclave.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CertIssuerIsThisTISC}} \\
\hline
\textit{KeyStoreC} \\
\textit{CertificateContents} \\
\hline
\textit{privateKey} \neq \textit{nil} \\
\textit{idC}.\textit{issuerC} = (\textit{the}\ \textit{privateKey}).\textit{keyOwner} \\
\hline
\end{array}
$$

▷ See: *KeyStoreC* (p. 33), *CertificateContents* (p. 16)

$$\textit{AuthCertOKC} \mathrel{\widehat=} \textit{CertIssuerIsThisTISC} \land \textit{CertOKC}$$

▷ See: *CertIssuerIsThisTISC* (p. 66), *CertOKC* (p. 65)

5.5.2 Currency of Certificates

FD.Certificate.IsCurrent

A certificate is considered current, within the context of the current time if the current time lies between the not before time and the not after time.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CertIsCurrent}} \\
\hline
\textit{CertificateContents} \\
\textit{currentTimeC} : \textit{TIME} \\
\hline
\textit{currentTimeC} \in \textit{notBefore} \mathrel{..} \textit{notAfter} \\
\hline
\end{array}
$$

▷ See: *CertificateContents* (p. 16), *TIME* (p. 11)

5.5.3 Generating Authorisation Certificates

FD.Certificate.NewAuthCert
FS.Certificate.NewAuthCert

An authorisation certificate contents can be constructed using information from a valid token and the current configuration of TIS. TIS can only generate the authorisation certificate if it has its own key to perform the signing with.

All Authorisation certificates will be signed using RSA the encryption of a SHA-1 digest.

$$
\begin{array}{|l}
\textit{NewAuthCertContents} \\
\hline
\textit{ValidTokenC} \\
\textit{KeyStoreC} \\
\textit{CertificateStore} \\
\textit{ConfigC} \\
\textit{newAuthCertContents}! : \textit{AuthCertContents} \\
\textit{currentTimeC} : \textit{TIME} \\
\hline
\textit{privateKey} \neq \textit{nil} \\
\textit{newAuthCertContents}!.\textit{idC}.\textit{issuerC} = (\textit{the}\ \textit{privateKey}).\textit{keyOwner} \\
\textit{newAuthCertContents}!.\textit{idC}.\textit{serialNumber} = \textit{nextSerialNumber} \\
(\textit{currentTimeC} \in \textit{authPeriodIsEmpty} \\
\quad \wedge \textit{newAuthCertContents}!.\textit{notBefore} = \textit{currentTimeC} \\
\quad \wedge \textit{newAuthCertContents}!.\textit{notAfter} = \textit{zeroTime} \\
\vee \textit{currentTimeC} \notin \textit{authPeriodIsEmpty} \\
\quad \wedge \textit{newAuthCertContents}!.\textit{notBefore} = \textit{first}\ (\textit{getAuthPeriod}\ \textit{currentTimeC}) \\
\quad \wedge \textit{newAuthCertContents}!.\textit{notAfter} = \textit{second}\ (\textit{getAuthPeriod}\ \textit{currentTimeC})) \\
\textit{newAuthCertContents}!.\textit{mechanism} = \textit{rsaWithSha}1 \\
\textit{newAuthCertContents}!.\textit{baseCertIdC} = (\textit{extractIDCert}\ \textit{idCertC}).\textit{idC} \\
\textit{newAuthCertContents}!.\textit{roleC} = (\textit{extractPrivCert}\ \textit{privCertC}).\textit{roleC} \\
\textit{newAuthCertContents}!.\textit{clearanceC}.\textit{class} = \textit{minClass}\{\textit{enclaveClearanceC}, (\textit{extractPrivCert}\ \textit{privCertC}).\textit{clearanceC}.\textit{class}\}
\end{array}
$$

▷ See: *ValidTokenC* (p. 19), *KeyStoreC* (p. 33), *CertificateStore* (p. 34), *ConfigC* (p. 27), *AuthCertContents* (p. 17), *TIME* (p. 11), *zeroTime* (p. 11), *rsaWithSha*1 (p. 15), *extractIDCert* (p. 17), *minClass* (p. 12)

The data for new authorisation certificate is constructed from the contents of the certificate. The signature is obtained by signing the digest of this data.

$$
\begin{array}{|l}
\textit{NewAuthCertC} \\
\hline
\textit{ValidTokenC} \\
\textit{KeyStoreC} \\
\textit{CertificateStore} \\
\textit{ConfigC} \\
\textit{newAuthCert}! : \textit{AuthCertC} \\
\textit{currentTimeC} : \textit{TIME} \\
\hline
\textit{privateKey} \neq \textit{nil} \\
\exists\, \textit{newAuthCertContents}! : \textit{AuthCertContents} \bullet \\
\quad \textit{NewAuthCertContents} \\
\quad \wedge \textit{newAuthCert}!.\textit{data} = \textit{constructAuthCert}\ \textit{newAuthCertContents}! \\
\textit{newAuthCert}!.\textit{signature} = \\
\quad \textit{sign}\ \textit{rsaWithSha}1\ (\textit{the}\ (\textit{keyMatchingIssuer}\ (\textit{the}\ \textit{privateKey}).\textit{keyOwner}.\textit{id}))\ (\textit{digest}\ \textit{rsaWithSha}1\ \textit{newAuthCert}!.\textit{data})
\end{array}
$$

▷ See: *ValidTokenC* (p. 19), *KeyStoreC* (p. 33), *CertificateStore* (p. 34), *ConfigC* (p. 27), *AuthCertC* (p. 18), *TIME* (p. 11), *AuthCertContents* (p. 17), *NewAuthCertContents* (p. 66), *rsaWithSha*1 (p. 15), *digest* (p. 15)

5.5.4 Adding Authorisation Certificates to User Token

FD.UserToken.AddAuthCertToUserToken
FS.UserToken.AddAuthCertToUserToken

If a valid user token is present in the system then an authorisation certificate can be added to it.

$$
\begin{array}{l}
\textit{AddAuthCertToUserTokenC} \\
\hline
\Delta \textit{UserTokenC} \\
\textit{KeyStoreC} \\
\textit{CertificateStore} \\
\textit{ConfigC} \\
\textit{currentTimeC} : \textit{TIME} \\
\hline
\textit{userTokenPresenceC} = \textit{present} \\
\textit{currentUserTokenC} \in \operatorname{ran} \textit{goodTC} \\
\exists \textit{ValidTokenC}; \textit{ValidTokenC}' \bullet \theta \textit{ValidTokenC} = (\textit{goodTC}^{\sim} \textit{currentUserTokenC}) \\
\quad \land \theta \textit{ValidTokenC}' = (\textit{goodTC}^{\sim} \textit{currentUserTokenC}') \\
\quad \land (\exists \textit{newAuthCert!} : \textit{AuthCertC} \bullet \textit{the authCertC}' = \textit{newAuthCert!} \land \textit{NewAuthCertC}) \\
\quad \land \textit{tokenIDC}' = \textit{tokenIDC} \\
\quad \land \textit{idCertC}' = \textit{idCertC} \\
\quad \land \textit{privCertC}' = \textit{privCertC} \\
\quad \land \textit{iandACertC}' = \textit{iandACertC} \\
\textit{userTokenPresenceC}' = \textit{userTokenPresenceC} \\
\hline
\end{array}
$$

▷ See: *UserTokenC* (p. 36), *KeyStoreC* (p. 33), *CertificateStore* (p. 34), *ConfigC* (p. 27), *TIME* (p. 11), *present* (p. 11), *goodTC* (p. 22), *ValidTokenC* (p. 19), *AuthCertC* (p. 18), *NewAuthCertC* (p. 67)

5.6 Updating the Key Store

FD.KeyStore.UpdateKeyStore
FS.KeyStore.UpdateKeyStore

The key store is updated using the supplied enrolment data to add issuers and their public keys.

$$
\begin{array}{l}
\textit{UpdateKeyStoreC} \\
\hline
\Delta \textit{KeyStoreC} \\
\textit{ValidEnrolC} \\
\hline
\textit{keys}' = (\textit{keys} \setminus \{\textit{key} : \textit{keys} \mid \textit{key}.\textit{keyType} = \textit{private}\}) \\
\quad \cup \{c : \operatorname{ran} \textit{issuerCertsC}; \textit{key} : \textit{KeyPart} \mid \\
\qquad\qquad \textit{key}.\textit{keyOwner} = (\textit{extractIDCert}\ c).\textit{subjectC} \\
\qquad\qquad \land \textit{key}.\textit{keyType} = \textit{public} \\
\qquad\qquad \land \textit{key}.\textit{keyData} = (\textit{extractIDCert}\ c).\textit{subjectPubKC} \bullet \textit{key}\} \\
\quad \cup \{\textit{key} : \textit{KeyPart} \mid \\
\qquad\qquad \textit{key}.\textit{keyOwner} = (\textit{extractIDCert}\ \textit{idStationCertC}).\textit{subjectC} \\
\qquad\qquad \land \textit{key}.\textit{keyType} = \textit{private} \\
\qquad\qquad \land \textit{key}.\textit{keyData} = \textit{theTISKey} \bullet \textit{key}\} \\
\textit{theTISKey}' = \textit{theTISKey} \\
\hline
\end{array}
$$

▷ See: *KeyStoreC* (p. 33), *ValidEnrolC* (p. 21), *private* (p. 14), *KeyPart* (p. 14), *extractIDCert* (p. 17), *public* (p. 14)

▷ This operation uses union and override so that it can be used to add issuers as well as initial enrolment.

▷ The enrolment data must include the ID certificate for this TIS. This contains the official name for the TIS and will result in the private TIS key being inserted into the keystore with the name of the TIS. If the private key was already in the keystore it will be replaced.

The enrolment data will always be supplied on a floppy disk.

$$
\begin{array}{|l}
\hline
\textit{UpdateKeyStoreFromFloppyC} \\
\Delta \textit{KeyStoreC} \\
\textit{FloppyC} \\
\hline
\textit{currentFloppyC} \in \operatorname{ran} \textit{enrolmentFileC} \\
(\exists \textit{ValidEnrolC} \bullet \theta \textit{ValidEnrolC} = \textit{enrolmentFileC}^{\sim} \textit{currentFloppyC} \\
\quad \wedge \textit{UpdateKeyStoreC}) \\
\hline
\end{array}
$$

▷ See: *KeyStoreC* (p. 33), *FloppyC* (p. 36), *enrolmentFileC* (p. 22), *ValidEnrolC* (p. 21), *UpdateKeyStoreC* (p. 68)

5.7 Token Validation

FD.Token.Validate

There are a number of validation checks that need to be performed on tokens. Some of these checks are consistency checks, others require the presence of a key store.

The token must contain an ID certificate, which has a serial number matching the tokenID.

$$
\begin{array}{|l}
\hline
\textit{TokenIDCertPresent} \\
\textit{TokenC} \\
\hline
\textit{idCertC} \in \operatorname{dom} \textit{extractIDCert} \\
(\textit{extractIDCert}\ \textit{idCertC}).\textit{idC}.\textit{serialNumber} = \textit{tokenIDC} \\
\hline
\end{array}
$$

▷ See: *TokenC* (p. 19), *extractIDCert* (p. 17)

The ID certificate must be correctly signed by a known issuer.

$$
\begin{array}{|l}
\hline
\textit{TokenIDCertOK} \\
\textit{TokenIDCertPresent} \\
\\
\textit{KeyStoreC} \\
\hline
\exists \textit{IDCertContents}; \textit{RawCertificate} \bullet \\
\quad \theta \textit{IDCertContents} = \textit{extractIDCert}\ \textit{idCertC} \wedge \theta \textit{RawCertificate} = \textit{idCertC} \wedge \textit{CertOKC} \\
\hline
\end{array}
$$

▷ See: *TokenIDCertPresent* (p. 69), *KeyStoreC* (p. 33), *IDCertContents* (p. 16), *RawCertificate* (p. 15), *extractIDCert* (p. 17), *CertOKC* (p. 65)

The ID certificate must be current.

$$
\begin{array}{|l}
\hline
\textit{TokenIDCertCurrent} \\
\textit{TokenIDCertPresent} \\
\\
\textit{currentTimeC} : \textit{TIME} \\
\hline
\exists \textit{IDCertContents} \bullet \theta \textit{IDCertContents} = \textit{extractIDCert}\ \textit{idCertC} \wedge \textit{CertIsCurrent} \\
\hline
\end{array}
$$

▷ See: *TokenIDCertPresent* (p. 69), *TIME* (p. 11), *IDCertContents* (p. 16), *extractIDCert* (p. 17), *CertIsCurrent* (p. 66)

The privilege certificate must be present and the base certificate must match the ID Certificate.

$$
\begin{array}{l}
\textbf{schema } \mathit{TokenPrivCertPresent} \\
\quad \mathit{TokenIDCertPresent} \\
\hline
\quad \mathit{privCertC} \in \operatorname{dom} \mathit{extractPrivCert} \\
\quad (\mathit{extractIDCert}\ \mathit{idCertC}).\mathit{idC} = (\mathit{extractPrivCert}\ \mathit{privCertC}).\mathit{baseCertIdC}
\end{array}
$$

▷ See: *TokenIDCertPresent* (p. 69), *extractIDCert* (p. 17)

The privilege certificate must be correctly signed by a known issuer.

$$
\begin{array}{l}
\textbf{schema } \mathit{TokenPrivCertOK} \\
\quad \mathit{TokenPrivCertPresent} \\
\quad \mathit{KeyStoreC} \\
\hline
\quad \exists\, \mathit{PrivCertContents}; \mathit{RawCertificate} \bullet \\
\qquad \theta \mathit{PrivCertContents} = \mathit{extractPrivCert}\ \mathit{privCertC} \land \theta \mathit{RawCertificate} = \mathit{privCertC} \land \mathit{CertOKC}
\end{array}
$$

▷ See: *TokenPrivCertPresent* (p. 70), *KeyStoreC* (p. 33), *PrivCertContents* (p. 17), *RawCertificate* (p. 15), *CertOKC* (p. 65)

The Priv certificate must be current.

$$
\begin{array}{l}
\textbf{schema } \mathit{TokenPrivCertCurrent} \\
\quad \mathit{TokenPrivCertPresent} \\
\quad \mathit{currentTimeC} : \mathit{TIME} \\
\hline
\quad \exists\, \mathit{PrivCertContents} \bullet \theta \mathit{PrivCertContents} = \mathit{extractPrivCert}\ \mathit{privCertC} \land \mathit{CertIsCurrent}
\end{array}
$$

▷ See: *TokenPrivCertPresent* (p. 70), *TIME* (p. 11), *PrivCertContents* (p. 17), *CertIsCurrent* (p. 66)

The I&A certificate must be present and the base certificate must match the ID Certificate.

$$
\begin{array}{l}
\textbf{schema } \mathit{TokenIandACertPresent} \\
\quad \mathit{TokenIDCertPresent} \\
\hline
\quad \mathit{iandACertC} \in \operatorname{dom} \mathit{extractIandACert} \\
\quad (\mathit{extractIDCert}\ \mathit{idCertC}).\mathit{idC} = (\mathit{extractIandACert}\ \mathit{iandACertC}).\mathit{baseCertIdC}
\end{array}
$$

▷ See: *TokenIDCertPresent* (p. 69), *extractIDCert* (p. 17)

The I&A certificate must be correctly signed by a known issuer.

$$
\begin{array}{l}
\textbf{schema } \mathit{TokenIandACertOK} \\
\quad \mathit{TokenIandACertPresent} \\
\quad \mathit{KeyStoreC} \\
\hline
\quad \exists\, \mathit{IandACertContents}; \mathit{RawCertificate} \bullet \\
\qquad \theta \mathit{IandACertContents} = \mathit{extractIandACert}\ \mathit{iandACertC} \land \theta \mathit{RawCertificate} = \mathit{iandACertC} \land \mathit{CertOKC}
\end{array}
$$

▷ See: *TokenIandACertPresent* (p. 70), *KeyStoreC* (p. 33), *IandACertContents* (p. 17), *RawCertificate* (p. 15), *CertOKC* (p. 65)

The I&A certificate must be current.

$$\begin{array}{|l}
\hline \textit{TokenIandACertCurrent} \\
\textit{TokenIandACertPresent} \\
\textit{currentTimeC} : \textit{TIME} \\
\hline
\exists \textit{IandACertContents} \bullet \theta\textit{IandACertContents} = \textit{extractIandACert}\ \textit{iandACertC} \land \textit{CertIsCurrent} \\
\hline
\end{array}$$

▷ See: *TokenIandACertPresent* (p. 70), *TIME* (p. 11), *IandACertContents* (p. 17), *CertIsCurrent* (p. 66)

$\textit{TokenOKC} \mathrel{\widehat=} \textit{TokenIDCertCurrent} \land \textit{TokenPrivCertCurrent} \land \textit{TokenIandACertCurrent}$

▷ See: *TokenIDCertCurrent* (p. 69), *TokenPrivCertCurrent* (p. 70), *TokenIandACertCurrent* (p. 71)

The Auth certificate must be present and the serial number of the base certificate must match the ID Certificate.

$$\begin{array}{|l}
\hline \textit{TokenAuthCertPresent} \\
\textit{TokenIDCertPresent} \\
\hline
\textit{authCertC} \neq \textit{nil} \\
\textit{the}\ \textit{authCertC} \in \operatorname{dom} \textit{extractAuthCert} \\
(\textit{extractIDCert}\ \textit{idCertC}).\textit{idC} = (\textit{extractAuthCert}\ (\textit{the}\ \textit{authCertC})).\textit{baseCertIdC} \\
\hline
\end{array}$$

▷ See: *TokenIDCertPresent* (p. 69), *extractIDCert* (p. 17)

The Auth certificate must be correctly signed by this TIS.

$$\begin{array}{|l}
\hline \textit{TokenAuthCertOK} \\
\textit{TokenAuthCertPresent} \\
\textit{KeyStoreC} \\
\hline
\exists \textit{AuthCertContents}; \textit{RawCertificate} \bullet \\
\qquad \theta\textit{AuthCertContents} = \textit{extractAuthCert}\ (\textit{the}\ \textit{authCertC}) \land \theta\textit{RawCertificate} = (\textit{the}\ \textit{authCertC}) \land \textit{AuthCertOKC} \\
\hline
\end{array}$$

▷ See: *TokenAuthCertPresent* (p. 71), *KeyStoreC* (p. 33), *AuthCertContents* (p. 17), *RawCertificate* (p. 15), *AuthCertOKC* (p. 66)

The Auth certificate must be current.

$$\begin{array}{|l}
\hline \textit{TokenAuthCertCurrent} \\
\textit{TokenAuthCertPresent} \\
\textit{currentTimeC} : \textit{TIME} \\
\hline
\exists \textit{AuthCertContents} \bullet \theta\textit{AuthCertContents} = \textit{extractAuthCert}\ (\textit{the}\ \textit{authCertC}) \land \textit{CertIsCurrent} \\
\hline
\end{array}$$

▷ See: *TokenAuthCertPresent* (p. 71), *TIME* (p. 11), *AuthCertContents* (p. 17), *CertIsCurrent* (p. 66)

$\textit{TokenWithOKAuthCertC} \mathrel{\widehat=} \textit{TokenAuthCertOK} \land \textit{TokenAuthCertCurrent}$

▷ See: *TokenAuthCertOK* (p. 71), *TokenAuthCertCurrent* (p. 71)

5.8 User Token Operations and Checks

5.8.1 User Token Clear

FD.UserToken.Clear

The user token held internally must be cleared whenever the token is removed. This ensures that no information relating to the user token is retained following token removal.

ClearUserToken

$\Delta UserTokenC$

$userTokenPresenceC' = absent$
$currentUserTokenC' = noTC$

▷ See: *UserTokenC* (p. 36), *absent* (p. 11), *noTC* (p. 22)

5.8.2 User Token Validation

FD.UserToken.UserTokenOK

The user token must be good, in that it must not result in errors being raised when it is read.

UserTokenGood

$UserTokenC$

$currentUserTokenC \in \operatorname{ran} goodTC$

▷ See: *UserTokenC* (p. 36), *goodTC* (p. 22)

UserTokenOKC

$UserTokenGood$
$KeyStoreC$
$currentTimeC : TIME$

$\exists TokenC \bullet goodTC\ \theta TokenC = currentUserTokenC$
$\quad \land TokenIDCertOK \land TokenIDCertCurrent$
$\quad \land TokenPrivCertOK \land TokenPrivCertCurrent$
$\quad \land TokenIandACertOK \land TokenIandACertCurrent$

▷ See: *UserTokenGood* (p. 72), *KeyStoreC* (p. 33), *TIME* (p. 11), *TokenC* (p. 19), *goodTC* (p. 22), *TokenIDCertOK* (p. 69), *TokenIDCertCurrent* (p. 69), *TokenPrivCertOK* (p. 70), *TokenPrivCertCurrent* (p. 70), *TokenIandACertOK* (p. 70), *TokenIandACertCurrent* (p. 71)

FD.UserToken.UserTokenNotOK

If a user token is not OK then the cause of the fault will be captured in the description of the audit entry.

$$tokenBad, idCertBad, idCertNotVerifiable, idCertNotCurrent,$$
$$iandACertBad, iandACertNotVerifiable, iandACertNotCurrent,$$
$$privCertBad, privCertNotVerifiable, privCertNotCurrent : TEXT$$

The formal statement below makes clear there is only one description generated. In the case where the token exhibits many faults the first applicable fault, taking the possible descriptions in the order presented, will be described in the description. This is not captured formally.

UserTokenNotOK

$$UserTokenC$$
$$KeyStoreC$$
$$currentTimeC : TIME$$
$$description! : TEXT$$

$$\neg\ UserTokenGood \land description! = tokenBad \lor (\exists\, TokenC \bullet goodTC\ \theta TokenC = currentUserTokenC$$
$$\land (\neg\ TokenIDCertPresent \land description! = idCertBad$$
$$\lor \neg\ TokenIDCertOK \land description! = idCertNotVerifiable$$
$$\lor \neg\ TokenIDCertCurrent \land description! = idCertNotCurrent$$
$$\lor \neg\ TokenPrivCertPresent \land description! = privCertBad$$
$$\lor \neg\ TokenPrivCertOK \land description! = privCertNotVerifiable$$
$$\lor \neg\ TokenPrivCertCurrent \land description! = privCertNotCurrent$$
$$\lor \neg\ TokenIandACertPresent \land description! = iandACertBad$$
$$\lor \neg\ TokenIandACertOK \land description! = iandACertNotVerifiable$$
$$\lor \neg\ TokenIandACertCurrent \land description! = iandACertNotCurrent))$$

▷ See: *UserTokenC* (p. 36), *KeyStoreC* (p. 33), *TIME* (p. 11), *UserTokenGood* (p. 72), *TokenC* (p. 19), *goodTC* (p. 22), *TokenIDCertPresent* (p. 69), *TokenIDCertOK* (p. 69), *TokenIDCertCurrent* (p. 69), *TokenPrivCertPresent* (p. 70), *TokenPrivCertOK* (p. 70), *privCertNotVerifiable* (p. 72), *TokenPrivCertCurrent* (p. 70), *privCertNotCurrent* (p. 72), *TokenIandACertPresent* (p. 70), *TokenIandACertOK* (p. 70), *TokenIandACertCurrent* (p. 71)

FD.UserToken.UserTokenWithOKAuthCert

We also need to check whether a User token has an acceptable Authorisation Certificate. This requires the Authorisation certificate to be present, correctly reference the TokenID, be verifiable within the context of the Key Store and be current.

UserTokenWithOKAuthCertC

$$UserTokenGood$$
$$KeyStoreC$$
$$currentTimeC : TIME$$

$$\exists\, TokenC \bullet goodTC(\theta TokenC) = currentUserTokenC$$
$$\land TokenIDCertOK$$
$$\land TokenAuthCertOK \land TokenAuthCertCurrent$$

▷ See: *UserTokenGood* (p. 72), *KeyStoreC* (p. 33), *TIME* (p. 11), *TokenC* (p. 19), *goodTC* (p. 22), *TokenIDCertOK* (p. 69), *TokenAuthCertOK* (p. 71), *TokenAuthCertCurrent* (p. 71)

5.9 Admin Token Operations and Checks

5.9.1 Admin Token Clear

FD.AdminToken.Clear

The admin token held internally must be cleared whenever the token is removed. This ensures that no information relating to the Admin token is retained following token removal.

ClearAdminToken

$\Delta AdminTokenC$

$adminTokenPresenceC' = absent$
$currentAdminTokenC' = noTC$

▷ See: *AdminTokenC* (p. 36), *absent* (p. 11), *noTC* (p. 22)

5.9.2 Admin Token Validation

The admin token must be good, in that it must not result in errors being raised when it is read.

AdminTokenGood

$AdminTokenC$

$currentAdminTokenC \in \operatorname{ran} goodTC$

▷ See: *AdminTokenC* (p. 36), *goodTC* (p. 22)

FD.AdminToken.Current

The Authorisation certificate on the admin token must be present, and current:

AdminTokenCurrent

$AdminTokenGood$
$currentTimeC : TIME$

$\exists TokenC \bullet goodTC\ \theta TokenC = currentAdminTokenC$
$\quad \wedge TokenAuthCertCurrent$

▷ See: *AdminTokenGood* (p. 74), *TIME* (p. 11), *TokenC* (p. 19), *goodTC* (p. 22), *TokenAuthCertCurrent* (p. 71)

FD.AdminToken.AdminTokenOK

Additionally it must be validated within the context of the key store and the role must correspond to an administrator.

$$
\begin{array}{l}
\textit{AdminTokenOKC} \\
\hline
\textit{AdminTokenCurrent} \\
\textit{KeyStoreC} \\
\hline
\exists \textit{TokenC} \bullet \textit{goodTC}\ \theta\textit{TokenC} = \textit{currentAdminTokenC} \\
\quad \land \textit{TokenIDCertOK} \\
\quad \land \textit{TokenAuthCertOK} \land \textit{TokenAuthCertCurrent} \\
\quad \land (\textit{extractAuthCert}\ (\textit{the}\ \textit{authCertC})).\textit{roleC} \in \textit{ADMINPRIVILEGE}
\end{array}
$$

▷ See: *AdminTokenCurrent* (p. 74), *KeyStoreC* (p. 33), *TokenC* (p. 19), *goodTC* (p. 22), *TokenIDCertOK* (p. 69), *TokenAuthCertOK* (p. 71), *TokenAuthCertCurrent* (p. 71), *ADMINPRIVILEGE* (p. 34)

FD.AdminToken.AdminTokenNotOK

If an admin token is not OK then the cause of the fault will be captured in the description of the audit entry.

$$
\textit{authCertBad}, \textit{authCertNotVerifiable}, \textit{authCertNotCurrent}, \textit{authCertNotAdmin} : \textit{TEXT}
$$

The formal statement below makes clear there is only one description generated. In the case where the token exhibits many faults the first applicable fault, taking the possible descriptions in the order presented, will be described in the description. This is not captured formally.

$$
\begin{array}{l}
\textit{AdminTokenNotOK} \\
\hline
\textit{AdminTokenC} \\
\textit{KeyStoreC} \\
\textit{currentTimeC} : \textit{TIME} \\
\\
\textit{description}! : \textit{TEXT} \\
\hline
\lnot \textit{AdminTokenGood} \land \textit{description}! = \textit{tokenBad} \lor (\exists \textit{TokenC} \bullet \textit{goodTC}\ \theta\textit{TokenC} = \textit{currentAdminTokenC} \\
\quad \land (\lnot \textit{TokenIDCertPresent} \land \textit{description}! = \textit{idCertBad} \\
\quad \lor \lnot \textit{TokenIDCertOK} \land \textit{description}! = \textit{idCertNotVerifiable} \\
\quad \lor \lnot \textit{TokenAuthCertPresent} \land \textit{description}! = \textit{authCertBad} \\
\quad \lor \lnot \textit{TokenAuthCertOK} \land \textit{description}! = \textit{authCertNotVerifiable} \\
\quad \lor \lnot \textit{TokenAuthCertCurrent} \land \textit{description}! = \textit{authCertNotCurrent} \\
\quad \lor (\textit{extractAuthCert}\ (\textit{the}\ \textit{authCertC})).\textit{roleC} \notin \textit{ADMINPRIVILEGE} \\
\qquad \land \textit{description}! = \textit{authCertNotAdmin}))
\end{array}
$$

▷ See: *AdminTokenC* (p. 36), *KeyStoreC* (p. 33), *TIME* (p. 11), *AdminTokenGood* (p. 74), *TokenC* (p. 19), *goodTC* (p. 22), *TokenIDCertPresent* (p. 69), *TokenIDCertOK* (p. 69), *TokenAuthCertPresent* (p. 71), *TokenAuthCertOK* (p. 71), *TokenAuthCertCurrent* (p. 71), *ADMINPRIVILEGE* (p. 34), *authCertNotAdmin* (p. 75)

5.10 Administrator Operations and Checks

An administrator may log on to the TIS console, logoff, or start an operation. There are also a number of checks that are performed on the Admin state.

5.10.1 Logon Administrator

FD.Admin.AdminLogon
FS.Admin.AdminLogon

An administrator can only log on if there is no-one currently logged on.

$$
\begin{array}{l}
\textit{AdminLogonC} \\
\Delta \textit{AdminC} \\
\textit{requiredRole}? : \textit{ADMINPRIVILEGE} \\
\hline
\textit{rolePresentC} = \textit{nil} \\
\textit{the rolePresentC}' = \textit{requiredRole}? \\
\textit{currentAdminOpC}' = \textit{nil}
\end{array}
$$

▷ See: *AdminC* (p. 35), *ADMINPRIVILEGE* (p. 34)

5.10.2 Logout Administrator

FD.Admin.AdminLogout
FS.Admin.AdminLogout

An administrator can always log off. This will terminate the current operation.

$$
\begin{array}{l}
\textit{AdminLogoutC} \\
\Delta \textit{AdminC} \\
\hline
\textit{rolePresentC}' = \textit{nil} \\
\textit{currentAdminOpC}' = \textit{nil}
\end{array}
$$

▷ See: *AdminC* (p. 35)

5.10.3 Administrator Starts Operation

FD.Admin.AdminStartOp
FS.Admin.AdminStartOp

An administrator, who is currently logged on, can start any of the operations that he is allowed to perform. An operation can only be started if there is no operation currently in progress.

$$
\begin{array}{l}
\textit{AdminStartOpC} \\
\Delta \textit{AdminC} \\
\textit{requestedOp}? : \textit{ADMINOP} \\
\hline
\textit{rolePresentC} \neq \textit{nil} \\
\textit{currentAdminOpC} = \textit{nil} \\
\textit{requestedOp}? \in \textit{availableOpsC} \\
\textit{rolePresentC}' = \textit{rolePresentC} \\
\textit{the currentAdminOpC}' = \textit{requestedOp}?
\end{array}
$$

▷ See: *AdminC* (p. 35), *ADMINOP* (p. 34)

5.10.4 Administrator Finishes Operation

FD.Admin.AdminFinishOp
FD.Admin.AdminFinishOp

An administrator, who is currently logged on, can finish an operation.

$$
\begin{array}{l}
\textit{AdminFinishOpC} \\
\hline
\Delta \textit{AdminC} \\
\hline
\textit{rolePresentC} \neq \textit{nil} \\
\textit{currentAdminOpC} \neq \textit{nil} \\
\\
\textit{rolePresentC}' = \textit{rolePresentC} \\
\textit{currentAdminOpC}' = \textit{nil}
\end{array}
$$

▷ See: *AdminC* (p. 35)

5.10.5 Administrator Checks

FD.Admin.AdminOpIsAvailable

A check can be made to ensure that the requested operation is one that is available.

$$
\begin{array}{l}
\textit{AdminOpIsAvailable} \\
\hline
\textit{AdminC} \\
\\
\textit{request}? : \textit{KEYBOARD} \\
\hline
\textit{request}? \in \textit{keyedOps} \llparenthesis \textit{availableOpsC} \rrparenthesis
\end{array}
$$

▷ See: *AdminC* (p. 35), *KEYBOARD* (p. 23), *keyedOps* (p. 23)

FD.Admin.AdminIsPresent

A check can be made to ensure that an administrator is logged on.

$$
\begin{array}{l}
\textit{AdminIsPresent} \\
\hline
\textit{AdminC} \\
\hline
\textit{rolePresentC} \neq \textit{nil}
\end{array}
$$

▷ See: *AdminC* (p. 35)

FD.Admin.AdminIsDoingOp

A check can be made to ensure that an administrator is performing an operation.

$$
\begin{array}{l}
\textit{AdminIsDoingOp} \\
\hline
\textit{AdminC} \\
\hline
\textit{currentAdminOpC} \neq \textit{nil}
\end{array}
$$

▷ See: *AdminC* (p. 35)

5.11 Prioritisation Checks

There are a number of checks relating to the internal state that determine what action needs to be performed.

5.11.1 Enrolement In Progress

FD.Enclave.EnrolmentInProgress

No other activity can take place until enrolment has completed.

$$
\begin{array}{|l}
\textit{EnrolmentIsInProgress} \\
\hline
\textit{enclaveStatusC} : \textit{ENCLAVESTATUS} \\
\hline
\textit{enclaveStatusC} \in \{\textit{notEnrolled}, \textit{waitingEnrol}, \textit{waitingEndEnrol}\} \\
\hline
\end{array}
$$

▷ See: *ENCLAVESTATUS* (p. 37), *notEnrolled* (p. 37), *waitingEnrol* (p. 37), *waitingEndEnrol* (p. 37)

5.11.2 Administrator Must Logout

FD.Enclave.AdminMustLogout

An administrator must be logged out if there is a role present but the administrator token has been torn. The only exception to this is when TIS is in the process of shutting down, this does not force a loggout if the token is removed.

$$
\begin{array}{|l}
\textit{PresentAdminHasDeparted} \\
\hline
\textit{AdminTokenC} \\
\textit{AdminC} \\
\textit{enclaveStatusC} : \textit{ENCLAVESTATUS} \\
\hline
\textit{AdminIsPresent} \\
\textit{currentAdminOpC} = \textit{nil} \lor \textit{the currentAdminOpC} \neq \textit{shutdownOp} \\
\textit{adminTokenPresenceC} = \textit{absent} \\
\hline
\end{array}
$$

▷ See: *AdminTokenC* (p. 36), *AdminC* (p. 35), *ENCLAVESTATUS* (p. 37), *AdminIsPresent* (p. 77), *shutdownOp* (p. 34), *absent* (p. 11)

A logged on administrator expires their logon if the authorisation certificate is no longer valid. This takes priority over any user entry activity, although expiry checks only take place when there is no administrator activity.

$$
\begin{array}{|l}
\textit{AdminTokenHasExpired} \\
\hline
\textit{AdminTokenC} \\
\textit{AdminC} \\
\\
\textit{enclaveStatusC} : \textit{ENCLAVESTATUS} \\
\textit{currentTimeC} : \textit{TIME} \\
\hline
\textit{AdminIsPresent} \\
\textit{enclaveStatusC} = \textit{enclaveQuiescent} \\
\textit{adminTokenPresenceC} = \textit{present} \\
\neg\ \textit{AdminTokenCurrent} \\
\hline
\end{array}
$$

▷ See: *AdminTokenC* (p. 36), *AdminC* (p. 35), *ENCLAVESTATUS* (p. 37), *TIME* (p. 11), *AdminIsPresent* (p. 77), *present* (p. 11), *AdminTokenCurrent* (p. 74)

$$AdminMustLogout \mathrel{\widehat=} AdminTokenHasExpired \lor PresentAdminHasDeparted$$

▷ See: *AdminTokenHasExpired* (p. 78), *PresentAdminHasDeparted* (p. 78)

5.11.3 User Departed

FD.UserEntry.UserHasDeparted

A user is considered to have just departed from the system if the status is not *quiescent* but the user token has been torn.

UserHasDeparted
UserTokenC
$statusC : STATUS$

$statusC \neq quiescent$
$userTokenPresenceC = absent$

▷ See: *UserTokenC* (p. 36), *STATUS* (p. 37), *quiescent* (p. 37), *absent* (p. 11)

5.11.4 User Entry In Progress

FD.UserEntry.UserEntryInProgress

User entry processing is considered to be in progess while the status is neither *quiescent* nor *waitingRemoveTokenFail*.

UserEntryInProgress
$statusC : STATUS$

$statusC \notin \{quiescent, waitingRemoveTokenFail\}$

▷ See: *STATUS* (p. 37), *quiescent* (p. 37), *waitingRemoveTokenFail* (p. 37)

5.11.5 Current User Entry Activity Possible

FD.UserEntry.CurrentUserEntryActivityPossible

A user entry activity is possible if a user has just departed or there is a user entry in progress.

$$CurrentUserEntryActivityPossible \mathrel{\widehat=} UserHasDeparted \lor UserEntryInProgress$$

▷ See: *UserHasDeparted* (p. 79), *UserEntryInProgress* (p. 79)

5.11.6 Admin Departed

FD.Enclave.AdminHasDeparted

A administrator is considered to have just departed from the system if the enclave status is not *quiescent* but the admin token has been torn.

$$
\begin{array}{l}
\textit{AdminHasDeparted} \\
\hline
\textit{AdminTokenC} \\
\textit{enclaveStatusC} : \textit{ENCLAVESTATUS} \\
\hline
\neg\ \textit{EnrolmentIsInProgress} \\
\\
\textit{enclaveStatusC} \neq \textit{enclaveQuiescent} \\
\textit{adminTokenPresenceC} = \textit{absent} \\
\hline
\end{array}
$$

▷ See: *AdminTokenC* (p. 36), *ENCLAVESTATUS* (p. 37), *EnrolmentIsInProgress* (p. 78), *absent* (p. 11)

5.11.7 Enclave Activities In Progress

FD.Enclave.EnclaveActivityInProgress

There is an administrator activity in progress within the enclave when the *enclaveStatus* is neither *enclaveQuiescent* nor *waitingRemovalAdminTokenFail* and an enrolment is not in progress.

$$
\begin{array}{l}
\textit{AdminActivityInProgress} \\
\hline
\textit{enclaveStatusC} : \textit{ENCLAVESTATUS} \\
\hline
\neg\ \textit{EnrolmentIsInProgress} \\
\textit{enclaveStatusC} \notin \{\textit{enclaveQuiescent}, \textit{waitingRemoveAdminTokenFail}\} \\
\hline
\end{array}
$$

▷ See: *ENCLAVESTATUS* (p. 37), *EnrolmentIsInProgress* (p. 78), *waitingRemoveAdminTokenFail* (p. 37)

5.11.8 Current Enclave Activity Possible

FD.Enclave.CurrentAdminActivityPossible

An enclave activity is possible if the administrator has just departed or there is an enclave activity in progress.

$$\textit{CurrentAdminActivityPossible} \mathrel{\widehat{=}} \textit{AdminHasDeparted} \lor \textit{AdminActivityInProgress}$$

▷ See: *AdminHasDeparted* (p. 80), *AdminActivityInProgress* (p. 80)

5.11.9 User Entry Can Start

FD.UserEntry.UserEntryCanStart

User entry processing can start if a user token is present and the status is *quiescent*.

UserEntryCanStart
UserTokenC
$statusC : STATUS$

$statusC = quiescent$
$userTokenPresenceC = present$

▷ See: *UserTokenC* (p. 36), *STATUS* (p. 37), *quiescent* (p. 37), *present* (p. 11)

5.11.10 Admin Op Can Start

FD.Enclave.AdminOpCanStart

An administrator operation can start if there is an administrator present and enclave is quiescent.

AdminOpCanStart
AdminC
AdminTokenC
$enclaveStatusC : ENCLAVESTATUS$

AdminIsPresent
$enclaveStatusC = enclaveQuiescent$
$adminTokenPresenceC = present$

▷ See: *AdminC* (p. 35), *AdminTokenC* (p. 36), *ENCLAVESTATUS* (p. 37), *AdminIsPresent* (p. 77), *present* (p. 11)

5.11.11 Admin Logon Can Start

FD.Enclave.AdminLoginCanStart

An administrator operation can start the logon procedure if there is no administrator present and enclave is quiescent.

AdminLogonCanStart
AdminTokenC
AdminC
$enclaveStatusC : ENCLAVESTATUS$

$\neg$ *AdminIsPresent*
$enclaveStatusC = enclaveQuiescent$
$adminTokenPresenceC = present$

▷ See: *AdminTokenC* (p. 36), *AdminC* (p. 35), *ENCLAVESTATUS* (p. 37), *AdminIsPresent* (p. 77), *present* (p. 11)

6 THE USER ENTRY OPERATION

FD.External.TISUserEntryOp
FS.External.TISUserEntryOp

This operation is a multi-stage operation and will be presented as a number of operations with preconditions on the internal *state*. The state transition diagram for user authentication and entry is given in Figure 6.1. Before user authentication and entry the system is in the *quiescent* state, on completion of the user authentication and entry the system will return the to *quiescent* state.

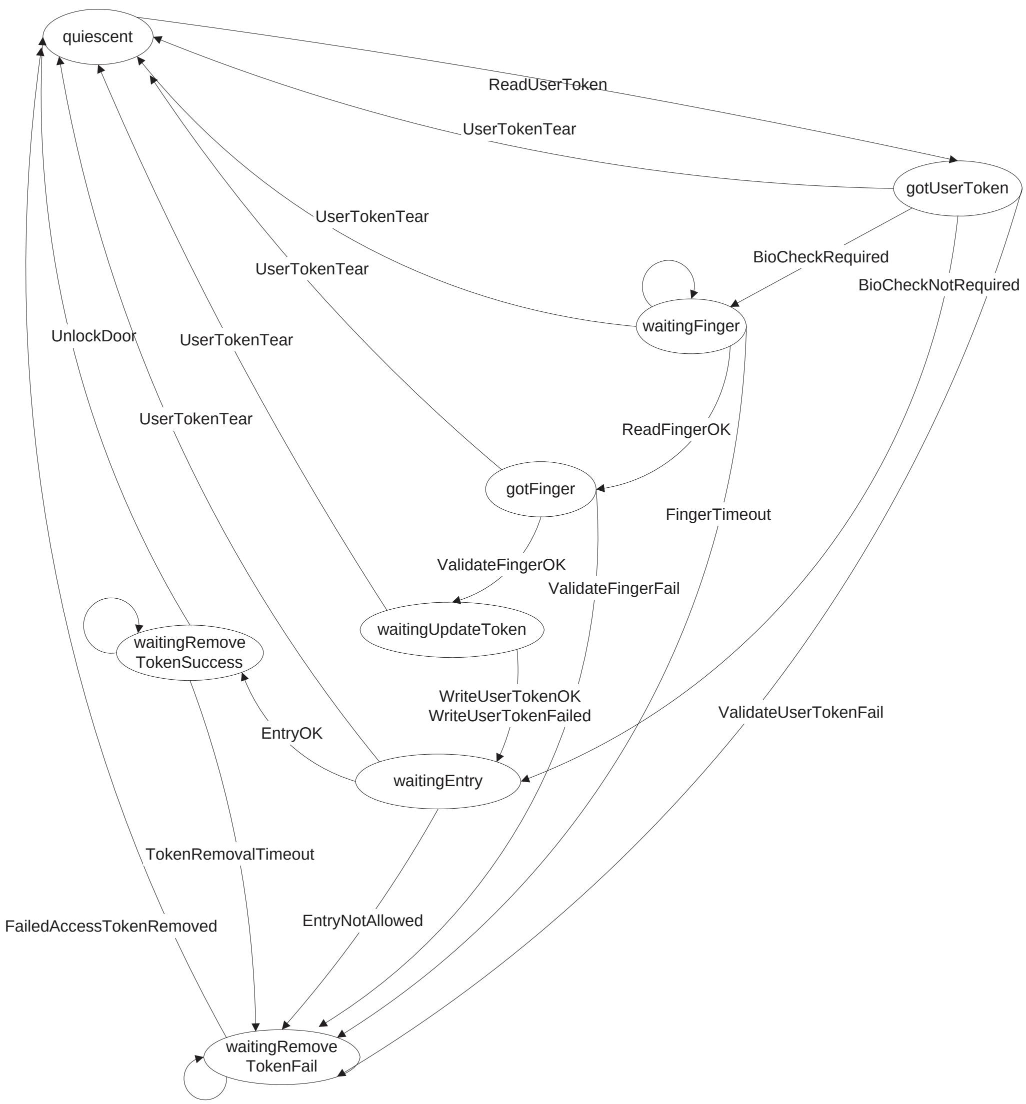

Figure 6.1: User Authentication and Entry state transitions

The process of user authentication and entry follows the following stages:

- Before any user attempts access, the system is *quiescent*.
- Once the token has been inserted and the information read off, the status moves to *gotUserToken*, waiting for the system to validate the token.
- Once the token has been successfully validated the status moves to *waitingFinger*, waiting for the user to give a fingerprint.
- Once the fingerprint has been read, the status moves to *gotFinger*, waiting for the system to validate the fingerprint.
- Once a fingerprint has been successfully validated, the status moves to *waitingUpdateToken*, waiting to write the Auth Cert to the token.
- Once the Auth Cert has been written, the status moves to *waitingEntry*, where it determines whether the role has current entry privileges.
- If the role has current entry privileges the status moves to *waitingTokenRemoveSuccess*, where the system system waits for the token to be removed.
- Once the token has been removed the latch will be unlocked if the role has current access privileges to the enclave and the ID Station will return to *quiescent*.

In the case of a failure in the user validation process the status moves to *waitingRemoveTokenFail*, waiting until the token has been removed before returning to a *quiescent* state.

This specification separates opening the door from having a valid Auth Certificate. It is possible for a role to be entitled to enter the enclave but not use the workstations (for example such clearence might be given to a buildings maintenance engineer). TIS configurations will ensure that having a valid Auth Certificate will guarantee that entry to the enclave is permitted.

FD.Enclave.ResetScreenMessage
FS.Enclave.ResetScreenMessage

The message displayed on the screen will indicate that the system is busy while a user entry is in progress that blocks administrator activity. Once the user entry activity becomes non-blocking then an appropriate message is displayed on the screen.

ResetScreenMessageC

$$
\begin{array}{l}
\Delta InternalC \\
\Delta AdminC \\
currentScreenC, currentScreenC' : ScreenC \\
\hline
UserEntryInProgress' \land currentScreenC'.screenMsgC = busyC \\
\lor \\
\neg\ UserEntryInProgress' \\
\quad \land (enclaveStatusC' = enclaveQuiescent \land rolePresentC' = nil \\
\qquad\qquad \land currentScreenC'.screenMsgC = welcomeAdminC \\
\qquad \lor enclaveStatusC' = enclaveQuiescent \land rolePresentC' \neq nil \\
\qquad\qquad \land currentScreenC'.screenMsgC = requestAdminOpC \\
\qquad \lor enclaveStatusC' = waitingRemoveAdminTokenFail \\
\qquad\qquad \land currentScreenC'.screenMsgC = removeAdminTokenC \\
\qquad \lor enclaveStatusC' \notin \{enclaveQuiescent, waitingRemoveAdminTokenFail\} \\
\qquad\qquad \land currentScreenC'.screenMsgC = currentScreenC.screenMsgC)
\end{array}
$$

▷ See: *InternalC* (p. 37), *AdminC* (p. 35), *ScreenC* (p. 36), *UserEntryInProgress* (p. 79), *busyC* (p. 23), *welcomeAdminC* (p. 23), *waitingRemoveAdminTokenFail* (p. 37), *removeAdminTokenC* (p. 23)

The user entry operation leaves much of the *IDStation* state unchanged. The context of this operation is summarised:

$$
\begin{array}{l}
\textit{UserEntryContextC} \\
\hline
\Delta \textit{IDStationC} \\
\textit{RealWorldChangesC} \\
\\
\Xi \textit{ConfigC} \\
\Xi \textit{AdminTokenC} \\
\Xi \textit{KeyStoreC} \\
\Xi \textit{AdminC} \\
\Xi \textit{KeyboardC} \\
\Xi \textit{FloppyC} \\
\textit{AddElementsToLogC} \\
\textit{LogChangeC} \\
\textit{ResetScreenMessageC} \\
\\
\Xi \textit{TISControlledRealWorldC} \\
\hline
\textit{enclaveStatusC}' = \textit{enclaveStatusC} \\
\textit{statusC} \neq \textit{waitingEntry} \Rightarrow \textit{tokenRemovalTimeoutC}' = \textit{tokenRemovalTimeoutC} \\
\textit{statusC}' \neq \textit{waitingFinger} \Rightarrow \textit{fingerTimeout}' = \textit{fingerTimeout} \\
\\
\textit{auditTypes}\ \textit{newElements}? \subseteq \textit{USER\_ENTRY\_ELEMENTS} \cup \textit{USER\_INDEPENDENT\_ELEMENTS}
\end{array}
$$

▷ See: *IDStationC* (p. 38), *RealWorldChangesC* (p. 40), *ConfigC* (p. 27), *AdminTokenC* (p. 36), *KeyStoreC* (p. 33), *AdminC* (p. 35), *KeyboardC* (p. 36), *FloppyC* (p. 36), *AddElementsToLogC* (p. 51), *LogChangeC* (p. 61), *ResetScreenMessageC* (p. 83), *TISControlledRealWorldC* (p. 24), *waitingEntry* (p. 37), *waitingFinger* (p. 37), *USER_ENTRY_ELEMENTS* (p. 29), *USER_INDEPENDENT_ELEMENTS* (p. 29)

▷ The following state components may change *UserTokenC*, *InternalC DoorLatchAlarmC*, *CertificateStore*, *StatsC* and *AuditLogC*.

▷ The components of the real world controlled by TIS remain unchanged.

▷ The *tokenRemovalTimeoutC* is only updated if the current status is *waitingEntry*.

▷ The *fingerTimeout* may only be updated if the current status becomes *waitingFinger*.

▷ All elements logged during user entry operations either relate to the user entry or are independent of any operation.

▷ Changes are logged and *newElements*? will be added to the Audit Log.

Each of the following sub-operations is performed within the above context.

6.1 User Token Tears

FD.UserEntry.UserTokenTorn
FS.UserEntry.UserTokenTorn

During the operation the user may tear his token from the reader prematurely. There are a number of internal states during which token removal is deamed erroneous.

If the user tears the Token out before the operation is complete then the operation is terminated unsuccessfully.

$$
\begin{array}{l}
\textbf{UserTokenTornC} \\
\hline
UserEntryContextC \\
\\
ClearUserToken \\
\Xi DoorLatchAlarmC \\
AddFailedEntryToStatsC \\
\Xi CertificateStore \\
\hline
UserHasDeparted \\
statusC \in \{gotUserToken, waitingUpdateToken, waitingFinger, gotFinger, waitingEntry\} \\
\\
currentDisplayC' = welcome \\
statusC' = quiescent \\
\\
auditTypes\ newElements? \cap USER\_ENTRY\_ELEMENTS = \{userTokenRemovedElement\} \\
\\
\exists_1\ element : AuditC \bullet element \in newElements? \\
\quad \land element.elementId = userTokenRemovedElement \\
\quad \land element.logTime \in nowC \mathrel{..} nowC' \\
\quad \land element.user = extractUser\ currentUserTokenC \\
\quad \land element.severity = warning \\
\quad \land element.description = noDescription
\end{array}
$$

▷ See: *UserEntryContextC* (p. 84), *ClearUserToken* (p. 72), *DoorLatchAlarmC* (p. 35), *AddFailedEntryToStatsC* (p. 61), *CertificateStore* (p. 34), *UserHasDeparted* (p. 79), *waitingUpdateToken* (p. 37), *waitingFinger* (p. 37), *gotFinger* (p. 37), *waitingEntry* (p. 37), *welcome* (p. 22), *quiescent* (p. 37), *USER_ENTRY_ELEMENTS* (p. 29), *userTokenRemovedElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *warning* (p. 28), *noDescription* (p. 30)

▷ The *userTokenRemovedElement* is the audit entry recording that the token has been removed from the reader outside the enclave.

6.2 Reading the User Token

FD.UserEntry.TISReadUserToken
FS.UserEntry.TISReadUserToken

The User Entry operation is initiated when TIS is in a *quiescent* state and detects the presence of a token in the user token reader (which resides outside the enclave).

A user entry operation may start while the *enclaveStatus* is quiescent (*enclaveQuiescent*) or the enclave is waiting for a failed admin token to be removed.

When the user token is first detected as present, its presence is audited and the internal status changes. It is not until the token has been validated that we can be sure of the user's identity, however the token ATR should provide a token ID which can be used as the user identity.

No other aspects of the system are modified.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{GetPresentUserTokenC}} \\
\hline
\textit{UserEntryContextC} \\[4pt]
\textit{ReadUserTokenC} \\
\Xi \textit{DoorLatchAlarmC} \\
\Xi \textit{StatsC} \\
\Xi \textit{CertificateStore} \\
\hline
\textit{UserEntryCanStart} \\[4pt]
\textit{userTokenPresenceC}' = \textit{userTokenPresenceC} \\
\textit{currentUserTokenC}' = \textit{userTokenC} \\[4pt]
\textit{currentDisplayC}' = \textit{wait} \\
\textit{statusC}' = \textit{gotUserToken} \\[4pt]
\textit{auditTypes}\ \textit{newElements}? \cap \textit{USER\_ENTRY\_ELEMENTS} = \{\textit{userTokenPresentElement}\} \\[4pt]
\exists_1\ \textit{element} : \textit{AuditC} \bullet \textit{element} \in \textit{newElements}? \\
\quad \wedge\ \textit{element.elementId} = \textit{userTokenPresentElement} \\
\quad \wedge\ \textit{element.logTime} \in \textit{nowC} \mathrel{..} \textit{nowC}' \\
\quad \wedge\ \textit{element.user} = \textit{extractUser}\ \textit{currentUserTokenC}' \\
\quad \wedge\ \textit{element.severity} = \textit{information} \\
\quad \wedge\ \textit{element.description} = \textit{noDescription} \\
\hline
\end{array}
$$

▷ See: *UserEntryContextC* (p. 84), *ReadUserTokenC* (p. 43), *DoorLatchAlarmC* (p. 35), *StatsC* (p. 34), *CertificateStore* (p. 34), *UserEntryCanStart* (p. 80), *wait* (p. 22), *USER_ENTRY_ELEMENTS* (p. 29), *userTokenPresentElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *information* (p. 28), *noDescription* (p. 30)

The operation to read the user token is as follows:

$$\textit{TISReadUserTokenC} \mathrel{\widehat=} \textit{GetPresentUserTokenC} \setminus (\textit{newElements}?)$$

▷ See: *GetPresentUserTokenC* (p. 85)

6.3 Validating the User Token

Once TIS has read a user token it must validate the contents of that token.

A user token is valid for entry without biometric checks if the token contains a consistent authorisation certificate which is current.

A user token is valid for entry into the enclave if the token is consistent, current and the ID certificate, Privilege certificate and I&A certificate can be validated.

FD.UserEntry.BioCheckNotRequired
FS.UserEntry.BioCheckNotRequired

In the case where there is a valid Authorisation certificate the biometric checks are bypassed.

BioCheckNotRequiredC

$$
\begin{array}{l}
UserEntryContextC \\
\\
\Xi UserTokenC \\
\Xi DoorLatchAlarmC \\
\Xi StatsC \\
\Xi CertificateStore \\
\hline
\neg\ UserHasDeparted \\
statusC = gotUserToken \\
\\
UserTokenWithOKAuthCertC \\
\\
statusC' = waitingEntry \\
currentDisplayC' = wait \\
\\
auditTypes\ newElements? \cap USER\_ENTRY\_ELEMENTS = \{authCertValidElement\} \\
\\
\exists_1\ element : AuditC \bullet element \in newElements? \\
\quad \land\ element.elementId = authCertValidElement \\
\quad \land\ element.logTime \in nowC \mathinner{..} nowC' \\
\quad \land\ element.user = extractUser\ currentUserTokenC \\
\quad \land\ element.severity = information \\
\quad \land\ element.description = noDescription
\end{array}
$$

▷ See: *UserEntryContextC* (p. 84), *UserTokenC* (p. 36), *DoorLatchAlarmC* (p. 35), *StatsC* (p. 34), *CertificateStore* (p. 34), *UserHasDeparted* (p. 79), *UserTokenWithOKAuthCertC* (p. 73), *waitingEntry* (p. 37), *wait* (p. 22), *USER_ENTRY_ELEMENTS* (p. 29), *authCertValidElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *information* (p. 28), *noDescription* (p. 30)

FD.UserEntry.BioCheckRequired	
FS.UserEntry.BioCheckRequired	*FDP_RIP.2.1*

The biometric checks are only required if the Authorisation Certificate is not present or not valid. In this case the remaining certificates on the card must be checked.

An audit element is logged indicating that the authorisation certificate is not valid. The audit element will reference a user, the owner of the token, there is no additional information in the description.

BioCheckRequiredC

$UserEntryContextC$

$FlushFingerDataC$
$\Xi UserTokenC$
$\Xi DoorLatchAlarmC$
$\Xi StatsC$
$\Xi CertificateStore$

$\neg\ UserHasDeparted$
$statusC = gotUserToken$

$\neg\ UserTokenWithOKAuthCertC \land UserTokenOKC$

$currentDisplayC' = insertFinger$
$statusC' = waitingFinger$
$fingerTimeout' = currentTimeC + fingerWaitDuration$

$auditTypes\ newElements? \cap USER\_ENTRY\_ELEMENTS = \{authCertInvalidElement\}$

$\exists_1\ element : AuditC \bullet element \in newElements?$
$\quad \land\ element.elementId = authCertInvalidElement$
$\quad \land\ element.logTime \in nowC \mathrel{..} nowC'$
$\quad \land\ element.user = extractUser\ currentUserTokenC$
$\quad \land\ element.severity = information$
$\quad \land\ element.description = noDescription$

▷ See: *UserEntryContextC* (p. 84), *FlushFingerDataC* (p. 47), *UserTokenC* (p. 36), *DoorLatchAlarmC* (p. 35), *StatsC* (p. 34), *CertificateStore* (p. 34), *UserHasDeparted* (p. 79), *UserTokenWithOKAuthCertC* (p. 73), *UserTokenOKC* (p. 72), *insertFinger* (p. 22), *waitingFinger* (p. 37), *USER_ENTRY_ELEMENTS* (p. 29), *authCertInvalidElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *information* (p. 28), *noDescription* (p. 30)

FD.UserEntry.ValidateUserTokenFail
FS.UserEntry.ValidateUserTokenFail

If the token cannot be validated then this is logged and the user is required to remove the token. The audit element detailing this failure will contain the user if this can be extracted from the token. The description will indicate the point of failure of the card.

$$
\begin{array}{l}
\textbf{ValidateUserTokenFailC} \\
\hline
UserEntryContextC \\
\\
\Xi UserTokenC \\
\Xi DoorLatchAlarmC \\
\Xi StatsC \\
\Xi CertificateStore \\
\hline
\neg\ UserHasDeparted \\
statusC = gotUserToken \\
\\
\neg\ UserTokenOKC \land \neg\ UserTokenWithOKAuthCertC \\
\\
currentDisplayC' = removeToken \\
statusC' = waitingRemoveTokenFail \\
\\
auditTypes\ newElements? \cap USER\_ENTRY\_ELEMENTS = \{userTokenInvalidElement\} \\
\\
\exists_1\ element : AuditC; description! : TEXT \bullet \\
\quad element \in newElements? \\
\quad \land element.elementId = userTokenInvalidElement \\
\quad \land element.logTime \in nowC \mathrel{..} nowC' \\
\quad \land element.user = extractUser\ currentUserTokenC \\
\quad \land element.severity = warning \\
\quad \land (element.description = description! \land UserTokenNotOK)
\end{array}
$$

▷ See: *UserEntryContextC* (p. 84), *UserTokenC* (p. 36), *DoorLatchAlarmC* (p. 35), *StatsC* (p. 34), *CertificateStore* (p. 34), *UserHasDeparted* (p. 79), *UserTokenOKC* (p. 72), *UserTokenWithOKAuthCertC* (p. 73), *waitingRemoveTokenFail* (p. 37), *USER_ENTRY_ELEMENTS* (p. 29), *userTokenInvalidElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *warning* (p. 28), *UserTokenNotOK* (p. 73)

▷ *UserTokenNotOK* defines the error description.

6.3.1 Determining whether biometric checks are required

$$DetermineBioCheckRequired \mathrel{\widehat=} (BioCheckRequiredC \lor BioCheckNotRequiredC) \setminus (newElements?)$$

▷ See: *BioCheckRequiredC* (p. 87), *BioCheckNotRequiredC* (p. 86)

There are lots of things that may go wrong with validation of the user token. In each case the system will terminate the operation unsuccessfully.

$$
\begin{array}{l}
TISValidateUserTokenC \mathrel{\widehat=} (BioCheckRequiredC \lor BioCheckNotRequiredC \lor ValidateUserTokenFailC \\
\qquad\qquad \lor [\,UserTokenTornC \mid statusC = gotUserToken]) \setminus (newElements?)
\end{array}
$$

▷ See: *BioCheckRequiredC* (p. 87), *BioCheckNotRequiredC* (p. 86), *ValidateUserTokenFailC* (p. 88), *UserTokenTornC* (p. 84)

6.4 Reading a fingerprint

FD.UserEntry.ReadFingerOK
FS.UserEntry.ReadFingerOK

A finger will be read if the system is currently waiting for it (has not waited too long) and the user Token is in place.

$$
\begin{array}{l}
\textit{ReadFingerOKC} \\
\hline
\textit{UserEntryContextC} \\
\\
\Xi \textit{DoorLatchAlarmC} \\
\Xi \textit{UserTokenC} \\
\Xi \textit{StatsC} \\
\hline
\neg\ \textit{UserHasDeparted} \\
\textit{statusC} = \textit{waitingFinger} \\
\textit{fingerPresenceC} = \textit{present} \\
\textit{currentTimeC} \leq \textit{fingerTimeout} \\
\\
\textit{currentDisplayC}' = \textit{wait} \\
\textit{statusC}' = \textit{gotFinger} \\
\\
\textit{auditTypes}\ \textit{newElements}? \cap \textit{USER\_ENTRY\_ELEMENTS} = \{\textit{fingerDetectedElement}\} \\
\\
\exists_1\ \textit{element} : \textit{AuditC} \bullet \textit{element} \in \textit{newElements}? \\
\qquad \wedge\ \textit{element}.\textit{elementId} = \textit{fingerDetectedElement} \\
\qquad \wedge\ \textit{element}.\textit{logTime} \in \textit{nowC} \,..\, \textit{nowC}' \\
\qquad \wedge\ \textit{element}.\textit{user} = \textit{extractUser}\ \textit{currentUserTokenC} \\
\qquad \wedge\ \textit{element}.\textit{severity} = \textit{information} \\
\qquad \wedge\ \textit{element}.\textit{description} = \textit{noDescription}
\end{array}
$$

▷ See: *UserEntryContextC* (p. 84), *DoorLatchAlarmC* (p. 35), *UserTokenC* (p. 36), *StatsC* (p. 34), *UserHasDeparted* (p. 79), *waitingFinger* (p. 37), *present* (p. 11), *wait* (p. 22), *gotFinger* (p. 37), *USER_ENTRY_ELEMENTS* (p. 29), *fingerDetectedElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *information* (p. 28), *noDescription* (p. 30)

FD.UserEntry.NoFinger
FS.UserEntry.NoFinger

If there is no finger present then, if we have not allowed sufficient attempts to get and validate a finger, nothing happens.

$$
\begin{array}{l}
\textit{NoFingerC} \\
\hline
\Xi \textit{IDStationC} \\
\textit{RealWorldChangesC} \\
\\
\textit{UserEntryContextC} \\
\Xi \textit{TISControlledRealWorldC} \\
\hline
\neg\ \textit{UserHasDeparted} \\
\textit{statusC} = \textit{waitingFinger} \\
\textit{currentTimeC} \leq \textit{fingerTimeout} \\
\textit{fingerPresenceC} = \textit{absent}
\end{array}
$$

▷ See: *IDStationC* (p. 38), *RealWorldChangesC* (p. 40), *UserEntryContextC* (p. 84), *TISControlledRealWorldC* (p. 24), *UserHasDeparted* (p. 79), *waitingFinger* (p. 37), *absent* (p. 11)

FD.UserEntry.FingerTimeout
FS.UserEntry.FingerTimeout

Alternatively, TIS may have tried to obtain a valid finger for too long, in which case the user is requested to remove the token and the operation is terminated unsuccessfully.

$$
\begin{array}{l}
\textbf{FingerTimeoutC} \\
\hline
UserEntryContextC \\
\Xi UserTokenC \\
\Xi DoorLatchAlarmC \\
\Xi StatsC \\
\hline
\neg\ UserHasDeparted \\
statusC = waitingFinger \\
currentTimeC > fingerTimeout \\
currentDisplayC' = removeToken \\
statusC' = waitingRemoveTokenFail \\
auditTypes\ newElements? \cap USER\_ENTRY\_ELEMENTS = \{fingerTimeoutElement\} \\
\exists_1\ element : AuditC \bullet element \in newElements? \\
\quad \land element.elementId = fingerTimeoutElement \\
\quad \land element.logTime \in nowC \mathrel{..} nowC' \\
\quad \land element.user = extractUser\ currentUserTokenC \\
\quad \land element.severity = warning \\
\quad \land element.description = noDescription
\end{array}
$$

▷ See: *UserEntryContextC* (p. 84), *UserTokenC* (p. 36), *DoorLatchAlarmC* (p. 35), *StatsC* (p. 34), *UserHasDeparted* (p. 79), *waitingFinger* (p. 37), *waitingRemoveTokenFail* (p. 37), *USER_ENTRY_ELEMENTS* (p. 29), *fingerTimeoutElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *warning* (p. 28), *noDescription* (p. 30)

$$
\begin{array}{l}
TISReadFingerC \mathrel{\widehat=} (ReadFingerOKC \lor FingerTimeoutC \lor NoFingerC \\
\qquad \lor [\ UserTokenTornC \mid statusC = waitingFinger\ ]) \setminus (newElements?)
\end{array}
$$

▷ See: *ReadFingerOKC* (p. 89), *FingerTimeoutC* (p. 90), *NoFingerC* (p. 90), *UserTokenTornC* (p. 84), *waitingFinger* (p. 37)

6.5 Validating a fingerprint

FD.UserEntry.ValidateFingerOK	
FS.UserEntry.ValidateFingerOK	*FDP_RIP.2.1*

A finger must match the template information extracted from the userToken for it to be considered acceptable.

The fingerprint being successfully validated is a prerequisite for generating an authorisation certificate and adding it to the user token. Validating the fingerprint is performed first.

When logging the success or otherwise of the attempt to read the fingerprint the audit element will contain the achieved FAR if available.

$$
\mid\ achievedFarDescription : INTEGER32 \longrightarrow TEXT
$$

▷ See: *INTEGER32* (p. 11)

A fingerprint is considered OK if the *verifyBio* function returns a successful match indication.

Following a successful match the data is flushed from the biometric device.

$$
\begin{array}{l}
\textit{ValidateFingerOKC} \\
\hline
\textit{UserEntryContextC} \\
\\
\textit{FlushFingerDataC} \\
\Xi \textit{DoorLatchAlarmC} \\
\Xi \textit{UserTokenC} \\
\Xi \textit{CertificateStore} \\
\textit{AddSuccessfulBioCheckToStatsC} \\
\hline
\neg\ \textit{UserHasDeparted} \\
\textit{statusC} = \textit{gotFinger} \\
\\
\exists\, \textit{achievedFar}!, \textit{maxFar} : \textit{INTEGER}32 \bullet \\
\quad \textit{maxFar} = \textit{min}\{(\textit{extractIandACert}\ ((\textit{goodTC}^{\sim}\ \textit{currentUserTokenC}).\textit{iandACertC})).\textit{templateC}.\textit{far}, \textit{systemMaxFar}\} \\
\quad \land\ (\textit{match}, \textit{achievedFar}!) = \\
\qquad \textit{verifyBio}\ \textit{maxFar}\ (\textit{extractIandACert}\ ((\textit{goodTC}^{\sim}\ \textit{currentUserTokenC}).\textit{iandACertC})).\textit{templateC}.\textit{templateC}\ \textit{fingerC} \\
\\
\quad \land\ (\exists_1\ \textit{element} : \textit{AuditC} \bullet \textit{element} \in \textit{newElements}? \\
\qquad \land\ \textit{element}.\textit{elementId} = \textit{fingerMatchedElement} \\
\qquad \land\ \textit{element}.\textit{logTime} \in \textit{nowC} \mathinner{..} \textit{nowC}' \\
\qquad \land\ \textit{element}.\textit{user} = \textit{extractUser}\ \textit{currentUserTokenC} \\
\qquad \land\ \textit{element}.\textit{severity} = \textit{information} \\
\qquad \land\ \textit{element}.\textit{description} = \textit{achievedFarDescription}\ \textit{achievedFar}!) \\
\\
\textit{statusC}' = \textit{waitingUpdateToken} \\
\textit{currentDisplayC}' = \textit{wait} \\
\\
\textit{auditTypes}\ \textit{newElements}? \cap \textit{USER\_ENTRY\_ELEMENTS} = \{\textit{fingerMatchedElement}\} \\
\hline
\end{array}
$$

▷ See: *UserEntryContextC* (p. 84), *FlushFingerDataC* (p. 47), *DoorLatchAlarmC* (p. 35), *UserTokenC* (p. 36), *CertificateStore* (p. 34), *AddSuccessfulBioCheckToStatsC* (p. 62), *UserHasDeparted* (p. 79), *gotFinger* (p. 37), *INTEGER32* (p. 11), *goodTC* (p. 22), *match* (p. 14), *verifyBio* (p. 14), *AuditC* (p. 30), *fingerMatchedElement* (p. 28), *extractUser* (p. 30), *information* (p. 28), *achievedFarDescription* (p. 91), *waitingUpdateToken* (p. 37), *wait* (p. 22), *USER_ENTRY_ELEMENTS* (p. 29)

FD.UserEntry.ValidateFingerFail	
FS.UserEntry.ValidateFingerFail	*FDP_RIP.2.1*

If the fingerprint is not successfully validated the user is asked to remove their token and the entry attempt is terminated. The biometric check failure is recorded.

Following an unsuccessful match the data is flushed from the biometric device.

$$
\begin{array}{l}
\textbf{schema } \mathit{ValidateFingerFailC} \\
\quad \mathit{UserEntryContextC} \\
\quad \mathit{FlushFingerDataC} \\
\quad \Xi \mathit{UserTokenC} \\
\quad \Xi \mathit{DoorLatchAlarmC} \\
\quad \Xi \mathit{CertificateStore} \\
\quad \mathit{AddFailedBioCheckToStatsC} \\
\hline
\quad \neg\ \mathit{UserHasDeparted} \\
\quad \mathit{statusC} = \mathit{gotFinger} \\
\quad \exists\, \mathit{achievedFar}!, \mathit{maxFar} : \mathit{INTEGER32} \bullet \\
\qquad \mathit{maxFar} = \mathit{min}\{(\mathit{extractIandACert}\ ((\mathit{goodTC}^{\sim}\ \mathit{currentUserTokenC}).\mathit{iandACertC})).\mathit{templateC}.\mathit{far}, \mathit{systemMaxFar}\} \\
\qquad \wedge\ (\mathit{noMatch}, \mathit{achievedFar}!) = \\
\qquad\qquad \mathit{verifyBio}\ \mathit{maxFar}\ (\mathit{extractIandACert}\ ((\mathit{goodTC}^{\sim}\ \mathit{currentUserTokenC}).\mathit{iandACertC})).\mathit{templateC}.\mathit{templateC}\ \mathit{fingerC} \\
\qquad \wedge\ (\exists_1\ \mathit{element} : \mathit{AuditC} \bullet \mathit{element} \in \mathit{newElements}? \\
\qquad\qquad \wedge\ \mathit{element}.\mathit{elementId} = \mathit{fingerNotMatchedElement} \\
\qquad\qquad \wedge\ \mathit{element}.\mathit{logTime} \in \mathit{nowC} \mathrel{..} \mathit{nowC}' \\
\qquad\qquad \wedge\ \mathit{element}.\mathit{user} = \mathit{extractUser}\ \mathit{currentUserTokenC} \\
\qquad\qquad \wedge\ \mathit{element}.\mathit{severity} = \mathit{warning} \\
\qquad\qquad \wedge\ \mathit{element}.\mathit{description} = \mathit{achievedFarDescription}\ \mathit{achievedFar}!) \\
\quad \mathit{currentDisplayC}' = \mathit{removeToken} \\
\quad \mathit{statusC}' = \mathit{waitingRemoveTokenFail} \\
\quad \mathit{auditTypes}\ \mathit{newElements}? \cap \mathit{USER\_ENTRY\_ELEMENTS} = \{\mathit{fingerNotMatchedElement}\}
\end{array}
$$

▷ See: *UserEntryContextC* (p. 84), *FlushFingerDataC* (p. 47), *UserTokenC* (p. 36), *DoorLatchAlarmC* (p. 35), *CertificateStore* (p. 34), *AddFailedBioCheckToStatsC* (p. 62), *UserHasDeparted* (p. 79), *gotFinger* (p. 37), *INTEGER32* (p. 11), *goodTC* (p. 22), *noMatch* (p. 14), *verifyBio* (p. 14), *AuditC* (p. 30), *fingerNotMatchedElement* (p. 28), *extractUser* (p. 30), *warning* (p. 28), *achievedFarDescription* (p. 91), *waitingRemoveTokenFail* (p. 37), *USER_ENTRY_ELEMENTS* (p. 29)

$$
\begin{array}{l}
\mathit{TISValidateFingerC} \mathrel{\widehat=} (\mathit{ValidateFingerOKC} \vee \mathit{ValidateFingerFailC} \\
\qquad\qquad \vee\ [\,\mathit{UserTokenTornC} \mid \mathit{statusC} = \mathit{gotFinger}\,]) \setminus (\mathit{newElements}?)
\end{array}
$$

▷ See: *ValidateFingerOKC* (p. 91), *ValidateFingerFailC* (p. 92), *UserTokenTornC* (p. 84), *gotFinger* (p. 37)

6.6 Writing the User Token

The user Token will be updated with the new Auth certificate.

We implement a multi-phase design for the activity of writing the user token.

FD.UserEntry.ConstructAuthCert	
FS.UserEntry.WriteUerTokenOK	*FS.UserEntry.WriteUerTokenFail*

First the authorisation certificate is constructed. This certificate is added to the local copy of the user Token. This will not result in any errors since it does not require the use of any peripherals.

$$
\begin{array}{|l}
\textit{ConstructAuthCert} \\
\hline
\textit{UserEntryContextC} \\
\\
\Xi \textit{DoorLatchAlarmC} \\
\textit{AddAuthCertToUserTokenC} \\
\Xi \textit{CertificateStore} \\
\Xi \textit{StatsC} \\
\hline
\neg\ \textit{UserHasDeparted} \\
\textit{statusC} = \textit{waitingUpdateToken} \\
\\
\textit{statusC}' = \textit{statusC} \\
\textit{currentDisplayC}' = \textit{wait} \\
\\
\textit{auditTypes}\ \textit{newElements}? \cap \textit{USER\_ENTRY\_ELEMENTS} = \emptyset \\
\hline
\end{array}
$$

▷ See: *UserEntryContextC* (p. 84), *DoorLatchAlarmC* (p. 35), *AddAuthCertToUserTokenC* (p. 67), *CertificateStore* (p. 34), *StatsC* (p. 34), *UserHasDeparted* (p. 79), *waitingUpdateToken* (p. 37), *wait* (p. 22), *USER_ENTRY_ELEMENTS* (p. 29)

Next the certificate will be updated.

Finally the *CertificateStore* is updated to show the issuing of the certificate. This will only happen if the certificate is written successfully.

FD.UserEntry.WriteUserTokenOK
FS.UserEntry.WriteUserTokenOK

An attempt is made to write this certificate to the token. The write of the authorisation certificate may be successful...

$$
\begin{array}{|l}
\textit{WriteUserTokenOKC} \\
\hline
\textit{UserEntryContextC} \\
\\
\textit{UpdateUserTokenC} \\
\Xi \textit{DoorLatchAlarmC} \\
\Xi \textit{UserTokenC} \\
\textit{UpdateCertificateStore} \\
\Xi \textit{StatsC} \\
\hline
\neg\ \textit{UserHasDeparted} \\
\textit{statusC} = \textit{waitingUpdateToken} \\
\\
\textit{statusC}' = \textit{waitingEntry} \\
\textit{currentDisplayC}' = \textit{wait} \\
\\
\textit{auditTypes}\ \textit{newElements}? \cap \textit{USER\_ENTRY\_ELEMENTS} = \{\textit{authCertWrittenElement}\} \\
\\
\exists_1\ \textit{element} : \textit{AuditC} \bullet \textit{element} \in \textit{newElements}? \\
\quad \land\ \textit{element.elementId} = \textit{authCertWrittenElement} \\
\quad \land\ \textit{element.logTime} \in \textit{nowC} \mathrel{..} \textit{nowC}' \\
\quad \land\ \textit{element.user} = \textit{extractUser}\ \textit{currentUserTokenC} \\
\quad \land\ \textit{element.severity} = \textit{information} \\
\quad \land\ \textit{element.description} = \textit{noDescription} \\
\hline
\end{array}
$$

▷ See: *UserEntryContextC* (p. 84), *UpdateUserTokenC* (p. 46), *DoorLatchAlarmC* (p. 35), *UserTokenC* (p. 36), *UpdateCertificateStore* (p. 62), *StatsC* (p. 34), *UserHasDeparted* (p. 79), *waitingUpdateToken* (p. 37), *waitingEntry* (p. 37), *wait* (p. 22), *USER_ENTRY_ELEMENTS* (p. 29), *authCertWrittenElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *information* (p. 28), *noDescription* (p. 30)

▷ Note that the decision as to whether to update the certificate store or not can only be made once the attempt to write to the token has been completed. Only if this write succeeds should the *CertificateStore* be updated.

FD.UserEntry.WriteUserTokenFail
FS.UserEntry.WriteUserTokenFail

... or may fail. The failure case models circumstances where the TIS can detect the failure, through a write failure for instance, or a failure to generate the certificate. As there is no read back of the authorisation certificate we cannot guarantee that the audit log indicating a successful write means that the token contains the authorisation certificate. The user will still subsequently be admitted to the enclave if the conditions are correct.

Whether the authorisation certificate is successfully written or not is non-deterministic in this design since failure conditions on signing data and writing the certificate are not modelled.

$$
\begin{array}{l}
\textbf{schema } \mathit{WriteUserTokenFailC} \\
\quad \mathit{UserEntryContextC} \\
\quad \mathit{UpdateUserTokenC} \\
\quad \Xi \mathit{DoorLatchAlarmC} \\
\quad \Xi \mathit{UserTokenC} \\
\quad \Xi \mathit{StatsC} \\
\quad \Xi \mathit{CertificateStore} \\
\hline
\quad \neg\ \mathit{UserHasDeparted} \\
\quad \mathit{statusC} = \mathit{waitingUpdateToken} \\
\quad \mathit{statusC}' = \mathit{waitingEntry} \\
\quad \mathit{currentDisplayC}' = \mathit{tokenUpdateFailed} \\
\quad \mathit{auditTypes}\ \mathit{newElements}? \cap \mathit{USER\_ENTRY\_ELEMENTS} = \{\mathit{authCertWriteFailedElement}\} \\
\quad \exists_1\ \mathit{element} : \mathit{AuditC} \bullet \mathit{element} \in \mathit{newElements}? \\
\qquad \wedge\ \mathit{element}.\mathit{elementId} = \mathit{authCertWriteFailedElement} \\
\qquad \wedge\ \mathit{element}.\mathit{logTime} \in \mathit{nowC} \mathrel{..} \mathit{nowC}' \\
\qquad \wedge\ \mathit{element}.\mathit{user} = \mathit{extractUser}\ \mathit{currentUserTokenC} \\
\qquad \wedge\ \mathit{element}.\mathit{severity} = \mathit{warning} \\
\qquad \wedge\ \mathit{element}.\mathit{description} = \mathit{noDescription}
\end{array}
$$

▷ See: *UserEntryContextC* (p. 84), *UpdateUserTokenC* (p. 46), *DoorLatchAlarmC* (p. 35), *UserTokenC* (p. 36), *StatsC* (p. 34), *CertificateStore* (p. 34), *UserHasDeparted* (p. 79), *waitingUpdateToken* (p. 37), *waitingEntry* (p. 37), *tokenUpdateFailed* (p. 22), *USER_ENTRY_ELEMENTS* (p. 29), *authCertWriteFailedElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *warning* (p. 28), *noDescription* (p. 30)

$$\mathit{WriteUserTokenC} \mathrel{\widehat=} \mathit{WriteUserTokenOKC} \vee \mathit{WriteUserTokenFailC}$$

▷ See: *WriteUserTokenOKC* (p. 94), *WriteUserTokenFailC* (p. 95)

$$
\begin{array}{l}
\mathit{TISWriteUserTokenC} \mathrel{\widehat=} (((\mathit{ConstructAuthCert} \setminus (\mathit{newElements}?)) \mathbin{⨾} \mathit{WriteUserTokenC}) \\
\qquad \vee [\, \mathit{UserTokenTornC} \mid \mathit{statusC} = \mathit{waitingUpdateToken} \,]) \setminus (\mathit{newElements}?)
\end{array}
$$

▷ See: *ConstructAuthCert* (p. 93), *WriteUserTokenC* (p. 95), *UserTokenTornC* (p. 84), *waitingUpdateToken* (p. 37)

6.7 Validating Entry

The door will only be unlocked if the current TIS configuration allows the user to enter the enclave at this time. It is likely that TIS configurations will ensure that having a valid Auth Certificate will guarantee that entry to the enclave is permitted, but such a constraint is not specified here.

TIS checks to ensure that the current configuration allows the user to enter the enclave:

$$
\begin{array}{l}
\textbf{schema } \mathit{UserAllowedEntryC} \\
\quad \mathit{UserTokenC} \\
\quad \mathit{ConfigC} \\
\quad \mathit{currentTimeC} : \mathit{TIME} \\
\hline
\quad \exists\, \mathit{ValidTokenC} \bullet \\
\qquad \mathit{goodTC}(\theta \mathit{ValidTokenC}) = \mathit{currentUserTokenC} \\
\qquad \land\ \mathit{authCertC} \neq \mathit{nil} \\
\qquad \land\ \mathit{currentTimeC} \in \mathit{entryPeriodC}\ (\mathit{extractAuthCert}\ (\mathit{the}\ \mathit{authCertC})).\mathit{clearanceC}.\mathit{class}
\end{array}
$$

▷ See: *UserTokenC* (p. 36), *ConfigC* (p. 27), *TIME* (p. 11), *ValidTokenC* (p. 19), *goodTC* (p. 22)

FD.UserEntry.EntryOK
FS.UserEntry.EntryOK

Only if entry is permitted at the current time will the user be admitted to the enclave.

Note that if this stage of the processing is reached the internal representation of the token will always contain a valid authorisation certificate.

$$
\begin{array}{l}
\textbf{schema } \mathit{EntryOKC} \\
\quad \mathit{UserEntryContextC} \\
\\
\quad \Xi \mathit{DoorLatchAlarmC} \\
\quad \Xi \mathit{UserTokenC} \\
\quad \Xi \mathit{StatsC} \\
\quad \Xi \mathit{CertificateStore} \\
\hline
\quad \neg\ \mathit{UserHasDeparted} \\
\quad \mathit{statusC} = \mathit{waitingEntry} \\
\\
\quad \mathit{UserAllowedEntryC} \\
\\
\quad \mathit{currentDisplayC}' = \mathit{openDoor} \\
\quad \mathit{statusC}' = \mathit{waitingRemoveTokenSuccess} \\
\quad \mathit{tokenRemovalTimeoutC}' = \mathit{currentTimeC} + \mathit{tokenRemovalDurationC} \\
\\
\quad \mathit{auditTypes}\ \mathit{newElements}? \cap \mathit{USER\_ENTRY\_ELEMENTS} = \{\mathit{entryPermittedElement}\} \\
\\
\quad \exists_1\, \mathit{element} : \mathit{AuditC} \bullet \mathit{element} \in \mathit{newElements}? \\
\qquad \land\ \mathit{element}.\mathit{elementId} = \mathit{entryPermittedElement} \\
\qquad \land\ \mathit{element}.\mathit{logTime} \in \mathit{nowC} \mathinner{..} \mathit{nowC}' \\
\qquad \land\ \mathit{element}.\mathit{user} = \mathit{extractUser}\ \mathit{currentUserTokenC} \\
\qquad \land\ \mathit{element}.\mathit{severity} = \mathit{information} \\
\qquad \land\ \mathit{element}.\mathit{description} = \mathit{noDescription}
\end{array}
$$

▷ See: *UserEntryContextC* (p. 84), *DoorLatchAlarmC* (p. 35), *UserTokenC* (p. 36), *StatsC* (p. 34), *CertificateStore* (p. 34), *UserHasDeparted* (p. 79), *waitingEntry* (p. 37), *UserAllowedEntryC* (p. 96), *openDoor* (p. 22), *USER_ENTRY_ELEMENTS* (p. 29), *entryPermittedElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *information* (p. 28), *noDescription* (p. 30)

If the user is not allowed entry at this time they will be requested to remove their token.

$$
\begin{array}{|l}
\textit{EntryNotAllowedC} \\
\hline
\textit{UserEntryContextC} \\[4pt]
\Xi \textit{DoorLatchAlarmC} \\
\Xi \textit{UserTokenC} \\
\Xi \textit{StatsC} \\
\Xi \textit{CertificateStore} \\
\hline
\neg\ \textit{UserHasDeparted} \\
\textit{statusC} = \textit{waitingEntry} \\[4pt]
\neg\ \textit{UserAllowedEntryC} \\[4pt]
\textit{currentDisplayC}' = \textit{removeToken} \\
\textit{statusC}' = \textit{waitingRemoveTokenFail} \\
\textit{tokenRemovalTimeoutC}' = \textit{tokenRemovalTimeoutC} \\[4pt]
\textit{auditTypes}\ \textit{newElements}? \cap \textit{USER\_ENTRY\_ELEMENTS} = \{\textit{entryDeniedElement}\} \\[4pt]
\exists_1\ \textit{element} : \textit{AuditC} \bullet \textit{element} \in \textit{newElements}? \\
\quad \land\ \textit{element}.\textit{elementId} = \textit{entryDeniedElement} \\
\quad \land\ \textit{element}.\textit{logTime} \in \textit{nowC} \mathinner{..} \textit{nowC}' \\
\quad \land\ \textit{element}.\textit{user} = \textit{extractUser}\ \textit{currentUserTokenC} \\
\quad \land\ \textit{element}.\textit{severity} = \textit{warning} \\
\quad \land\ \textit{element}.\textit{description} = \textit{noDescription} \\
\hline
\end{array}
$$

▷ See: *UserEntryContextC* (p. 84), *DoorLatchAlarmC* (p. 35), *UserTokenC* (p. 36), *StatsC* (p. 34), *CertificateStore* (p. 34), *UserHasDeparted* (p. 79), *waitingEntry* (p. 37), *UserAllowedEntryC* (p. 96), *waitingRemoveTokenFail* (p. 37), *USER_ENTRY_ELEMENTS* (p. 29), *entryDeniedElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *warning* (p. 28), *noDescription* (p. 30)

$$
\begin{aligned}
\textit{TISValidateEntryC} \mathrel{\widehat=} (&\textit{EntryOKC} \\
&\lor \textit{EntryNotAllowedC} \\
&\lor [\,\textit{UserTokenTornC} \mid \textit{statusC} = \textit{waitingEntry}\,]) \setminus (\textit{newElements}?)
\end{aligned}
$$

▷ See: *EntryOKC* (p. 96), *EntryNotAllowedC* (p. 97), *UserTokenTornC* (p. 84), *waitingEntry* (p. 37)

6.8 Unlocking the Door

FD.UserEntry.UnlockDoorOK
FS.UserEntry.UnlockDoorOK

The door will only be unlocked if the current TIS configuration allows the user to enter the enclave at this time. It is likely that TIS configurations will ensure that having a valid Auth Certificate will guarantee that entry to the enclave is permitted.

The door will only be unlocked once the user has removed their token, this helps remind the user to take their token with them.

```
┌─ UnlockDoorOKC ─────────────────────────────
│ UserEntryContextC
│
│ UnlockDoorC
│ ClearUserToken
│ AddSuccessfulEntryToStatsC
│ ΞCertificateStore
├──────────
│ UserHasDeparted
│ statusC = waitingRemoveTokenSuccess
│
│ currentDisplayC′ = doorUnlocked
│ statusC′ = quiescent
└─────────────────────────────────────────────
```

▷ See: *UserEntryContextC* (p. 84), *UnlockDoorC* (p. 64), *ClearUserToken* (p. 72), *AddSuccessfulEntryToStatsC* (p. 61), *CertificateStore* (p. 34), *UserHasDeparted* (p. 79), *doorUnlocked* (p. 22), *quiescent* (p. 37)

FD.UserEntry.WaitingTokenRemoval
FS.UserEntry.WaitingTokenRemoval

The system will wait indefinitely for a token to be removed, however the system will deny entry to a user who takes too long to extract their token.

```
┌─ WaitingTokenRemovalC ──────────────────────
│ ΞIDStationC
│ RealWorldChangesC
│
│ ΞTISControlledRealWorldC
├──────────
│ ¬ UserHasDeparted
│ statusC = waitingRemoveTokenSuccess
│ currentTimeC ≤ tokenRemovalTimeoutC
└─────────────────────────────────────────────
```

▷ See: *IDStationC* (p. 38), *RealWorldChangesC* (p. 40), *TISControlledRealWorldC* (p. 24), *UserHasDeparted* (p. 79)

▷ The constraints on this schema have been tightened as idling while waiting for a failed token to be removed is considered part of the TIS system idle rather than the user entry operation.

FD.UserEntry.TokenRemovalTimeout
FS.UserEntry.TokenRemovalTimeout

If the user waits too long to remove their token then this is logged and the system continues to wait for the token to be removed but will no longer allow access to the enclave.

$$
\begin{array}{l}
\textbf{TokenRemovalTimeoutC} \\
\hline
UserEntryContextC \\[4pt]
\Xi DoorLatchAlarmC \\
\Xi UserTokenC \\
\Xi StatsC \\
\Xi CertificateStore \\
\hline
\neg\ UserHasDeparted \\
statusC = waitingRemoveTokenSuccess \\
currentTimeC > tokenRemovalTimeoutC \\[4pt]
statusC' = waitingRemoveTokenFail \\
currentDisplayC' = removeToken \\[4pt]
auditTypes\ newElements? \cap USER\_ENTRY\_ELEMENTS = \{entryTimeoutElement\} \\[4pt]
\exists_1\ element : AuditC \bullet element \in newElements? \\
\quad \wedge\ element.elementId = entryTimeoutElement \\
\quad \wedge\ element.logTime \in nowC \mathrel{..} nowC' \\
\quad \wedge\ element.user = extractUser\ currentUserTokenC \\
\quad \wedge\ element.severity = warning \\
\quad \wedge\ element.description = noDescription
\end{array}
$$

▷ See: *UserEntryContextC* (p. 84), *DoorLatchAlarmC* (p. 35), *UserTokenC* (p. 36), *StatsC* (p. 34), *CertificateStore* (p. 34), *UserHasDeparted* (p. 79), *waitingRemoveTokenFail* (p. 37), *USER_ENTRY_ELEMENTS* (p. 29), *entryTimeoutElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *warning* (p. 28), *noDescription* (p. 30)

$$
\begin{array}{l}
TISUnlockDoorC \mathrel{\widehat=} (UnlockDoorOKC \vee WaitingTokenRemovalC \\
\qquad \vee\ TokenRemovalTimeoutC) \setminus (newElements?)
\end{array}
$$

▷ See: *UnlockDoorOKC* (p. 97), *WaitingTokenRemovalC* (p. 98), *TokenRemovalTimeoutC* (p. 98)

6.9 Terminating a failed access

FD.UserEntry.FailedAccessTokenRemoved
FS.UserEntry.FailedAccessTokenRemoved

If an access attempt has failed the system waits for the token to be removed before a new user entry operation can commence. Once the token has been removed a new user entry may start.

The operations in the enclave are not blocked on the presence of a failed user token in the token reader.

$$
\begin{array}{l}
\textit{FailedAccessTokenRemovedC} \\
\hline
\textit{UserEntryContextC} \\
\\
\textit{ClearUserToken} \\
\Xi \textit{DoorLatchAlarmC} \\
\textit{AddFailedEntryToStatsC} \\
\Xi \textit{CertificateStore} \\
\hline
\textit{UserHasDeparted} \\
\textit{statusC} = \textit{waitingRemoveTokenFail} \\
\\
\textit{currentDisplayC}' = \textit{welcome} \\
\textit{statusC}' = \textit{quiescent} \\
\\
\textit{auditTypes}\ \textit{newElements}? \cap \textit{USER\_ENTRY\_ELEMENTS} = \{\textit{userTokenRemovedElement}\} \\
\\
\exists_1\ \textit{element} : \textit{AuditC} \bullet \textit{element} \in \textit{newElements}? \\
\quad \land \textit{element}.\textit{elementId} = \textit{userTokenRemovedElement} \\
\quad \land \textit{element}.\textit{logTime} \in \textit{nowC} \,..\, \textit{nowC}' \\
\quad \land \textit{element}.\textit{user} = \textit{extractUser}\ \textit{currentUserTokenC} \\
\quad \land \textit{element}.\textit{severity} = \textit{information} \\
\quad \land \textit{element}.\textit{description} = \textit{noDescription}
\end{array}
$$

▷ See: *UserEntryContextC* (p. 84), *ClearUserToken* (p. 72), *DoorLatchAlarmC* (p. 35), *AddFailedEntryToStatsC* (p. 61), *CertificateStore* (p. 34), *UserHasDeparted* (p. 79), *waitingRemoveTokenFail* (p. 37), *welcome* (p. 22), *quiescent* (p. 37), *USER_ENTRY_ELEMENTS* (p. 29), *userTokenRemovedElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *information* (p. 28), *noDescription* (p. 30)

$$\textit{TISCompleteFailedAccessC} \mathrel{\widehat=} \textit{FailedAccessTokenRemovedC} \setminus (\textit{newElements}?)$$

▷ See: *FailedAccessTokenRemovedC* (p. 99)

6.10 The Complete User Entry

The complete authentication process, triggered by TIS reading a User Token, involves validating the user Token, reading and validating the fingerprint, writing an authorisation certificate to the user token, waiting for the user to remove the token, opening the door to the enclave and in the case of a failure waiting for the system to be in a state where it can admit another user.

$$
\begin{array}{l}
\textit{TISUserEntryOpC} \mathrel{\widehat=} \textit{TISReadUserTokenC} \lor \textit{TISValidateUserTokenC} \lor \textit{TISReadFingerC} \lor \textit{TISValidateFingerC} \\
\qquad \lor \textit{TISWriteUserTokenC} \lor \textit{TISValidateEntryC} \lor \textit{TISUnlockDoorC} \lor \textit{TISCompleteFailedAccessC}
\end{array}
$$

▷ See: *TISReadUserTokenC* (p. 86), *TISValidateUserTokenC* (p. 89), *TISReadFingerC* (p. 91), *TISValidateFingerC* (p. 93), *TISWriteUserTokenC* (p. 95), *TISValidateEntryC* (p. 97), *TISUnlockDoorC* (p. 99), *TISCompleteFailedAccessC* (p. 100)

This can be divided into starting a user entry:

$$\textit{TISStartUserEntry} \mathrel{\widehat=} \textit{TISReadUserTokenC}$$

▷ See: *TISReadUserTokenC* (p. 86)

FD.UserEntry.ProgressUserEntry
FS.UserEntry.TISUserEntryOp

and progressing a started user entry:

$$\begin{aligned}TISProgressUserEntry \mathrel{\widehat{=}} {} & TISValidateUserTokenC \lor TISReadFingerC \lor TISValidateFingerC \\ & \lor TISWriteUserTokenC \lor TISValidateEntryC \lor TISUnlockDoorC \lor TISCompleteFailedAccessC\end{aligned}$$

▷ See: *TISValidateUserTokenC* (p. 89), *TISReadFingerC* (p. 91), *TISValidateFingerC* (p. 93), *TISWriteUserTokenC* (p. 95), *TISValidateEntryC* (p. 97), *TISUnlockDoorC* (p. 99), *TISCompleteFailedAccessC* (p. 100)

7 OPERATIONS WITHIN THE ENCLAVE

A number of interactions with TIS may occur within the Enclave. These interactions leave some of the *IDStation* state unchanged.

EnclaveContextC

Δ*IDStationC*
RealWorldChangesC

Ξ*TISControlledRealWorldC*

Ξ*UserTokenC*
Ξ*FingerC*
Ξ*StatsC*
Ξ*CertificateStore*
Ξ*Keyboard*

$fingerTimeout' = fingerTimeout$
$tokenRemovalTimeoutC' = tokenRemovalTimeoutC$

- ▷ See: *IDStationC* (p. 38), *RealWorldChangesC* (p. 40), *TISControlledRealWorldC* (p. 24), *UserTokenC* (p. 36), *FingerC* (p. 36), *StatsC* (p. 34), *CertificateStore* (p. 34)

- ▷ The following state components may change *KeyStoreC*, *FloppyC*, *ConfigC*, *AdminC*, *InternalC*, *AdminTokenC* *DoorLatchAlarmC* and *AuditLogC*.
- ▷ The components of the real world controlled by TIS remain unchanged.

The operations that may occur within the enclave include administrator operations and the ID station enrolment. These are described in this section.

7.1 Enrolment of an ID Station

FD.Enclave.TISEnrolOp
FS.Enclave.TISEnrolOp

Before TIS can be used it must be enrolled.

We assume that the initial enrolment is the only possible enrolment activity.

Enrolment is a multi-phase activity, the state transistions for an enrolment are given in Figure 7.1. Before enrolment the system is in state *notEnrolled* and, on successful completion, it enters the *quiescent* state.

The context for all enrolment operations is given below.

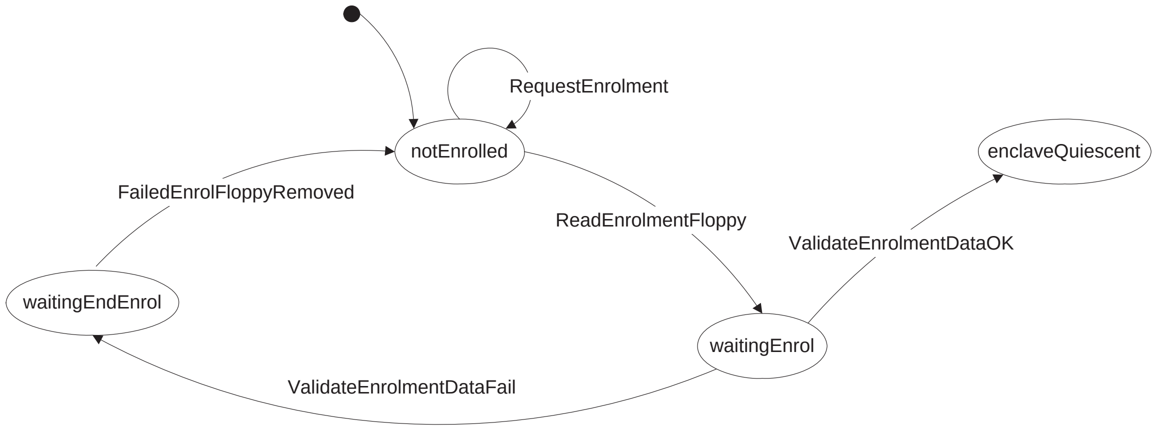

Figure 7.1: Enrolment state transitions

EnrolContextC
EnclaveContextC

Ξ*AdminC*
Ξ*AdminTokenC*
Ξ*DoorLatchAlarmC*
Ξ*ConfigC*
Ξ*FloppyC*
AddElementsToLogC
LogChangeC

$auditTypes\ newElements? \subseteq ENROL\_ELEMENTS \cup USER\_INDEPENDENT\_ELEMENTS$

▷ See: *EnclaveContextC* (p. 102), *AdminC* (p. 35), *AdminTokenC* (p. 36), *DoorLatchAlarmC* (p. 35), *ConfigC* (p. 27), *FloppyC* (p. 36), *AddElementsToLogC* (p. 51), *LogChangeC* (p. 61), *ENROL_ELEMENTS* (p. 29), *USER_INDEPENDENT_ELEMENTS* (p. 29)

▷ The following state components may change *KeyStore*, *Internal* and *AuditLog*.

7.1.1 Requesting Enrolment

FD.Enclave.RequestEnrolment
FS.Enclave.RequestEnrolment

The ID station will request enrolment while there is no Floppy present. This will occur until a successful enrolment is achieved.

$$
\begin{array}{|l}
\textbf{RequestEnrolmentC} \\
EnrolContextC \\
\\
\Xi KeyStoreC \\
\Xi FloppyC \\
\hline
enclaveStatusC = notEnrolled \\
floppyPresenceC = absent \\
\\
currentScreenC'.screenMsgC = insertEnrolmentDataC \\
\\
enclaveStatusC' = enclaveStatusC \\
statusC' = statusC \\
currentDisplayC' = blank \\
\\
auditTypes\ newElements? \cap ENROL\_ELEMENTS = \emptyset
\end{array}
$$

▷ See: *EnrolContextC* (p. 102), *KeyStoreC* (p. 33), *FloppyC* (p. 36), *notEnrolled* (p. 37), *absent* (p. 11), *blank* (p. 22), *ENROL_ELEMENTS* (p. 29)

FD.Enclave.ReadEnrolmentFloppy
FS.Enclave.ReadEnrolmentFloppy

If a floppy is present then TIS goes on to validate the contents. Nothing is written to the log at this stage as log entries will be made on successful or failed enrolment.

$$
\begin{array}{|l}
\textbf{ReadEnrolmentFloppyC} \\
EnrolContextC \\
\\
ReadFloppyC \\
\Xi KeyStoreC \\
\hline
enclaveStatusC = notEnrolled \\
floppyPresenceC = present \\
\\
currentScreenC'.screenMsgC = validatingEnrolmentDataC \\
\\
enclaveStatusC' = waitingEnrol \\
statusC' = statusC \\
currentDisplayC' = blank \\
\\
auditTypes\ newElements? \cap ENROL\_ELEMENTS = \emptyset
\end{array}
$$

▷ See: *EnrolContextC* (p. 102), *ReadFloppyC* (p. 44), *KeyStoreC* (p. 33), *notEnrolled* (p. 37), *present* (p. 11), *validatingEnrolmentDataC* (p. 23), *waitingEnrol* (p. 37), *blank* (p. 22), *ENROL_ELEMENTS* (p. 29)

$$ReadEnrolmentDataC \mathrel{\widehat=} (ReadEnrolmentFloppyC \lor RequestEnrolmentC) \setminus (newElements?)$$

▷ See: *ReadEnrolmentFloppyC* (p. 104), *RequestEnrolmentC* (p. 103)

7.1.2 Validating Enrolment data from Floppy

For the enrolment data to be acceptable the data on the floppy must be valid enrolment data with the ID Station certificate containing this ID station's public key.

$$
\begin{array}{|l}
\hline
\textit{EnrolmentDataOKC} \\
\textit{FloppyC} \\
\textit{KeyStoreC} \\
\hline
\textit{currentFloppyC} \in \operatorname{ran} \textit{enrolmentFileC} \\
(\exists\, \textit{ValidEnrolC} \bullet \theta \textit{ValidEnrolC} = \textit{enrolmentFileC}^{\sim}\, \textit{currentFloppyC}) \\
\hline
\end{array}
$$

▷ See: *FloppyC* (p. 36), *KeyStoreC* (p. 33), *enrolmentFileC* (p. 22), *ValidEnrolC* (p. 21)

FD.Enclave.ValidateEnrolmentDataOK
FS.Enclave.ValidateEnrolmentDataOK

If the data on the floppy is acceptable to be used for enrolment then the Key store is updated. From this point the system is available for use both by users entering the enclave and by administrators.

A successful enrolment is recorded in the audit log, no user can be associated with the enrolment activity.

$$
\begin{array}{|l}
\hline
\textit{ValidateEnrolmentDataOKC} \\
\textit{EnrolContextC} \\
\\
\Xi \textit{FloppyC} \\
\textit{UpdateKeyStoreFromFloppyC} \\
\hline
\textit{enclaveStatusC} = \textit{waitingEnrol} \\
\\
\textit{EnrolmentDataOKC} \\
\\
\textit{currentScreenC}'.\textit{screenMsgC} = \textit{welcomeAdminC} \\
\\
\textit{enclaveStatusC}' = \textit{enclaveQuiescent} \\
\textit{statusC}' = \textit{quiescent} \\
\textit{currentDisplayC}' = \textit{welcome} \\
\\
\textit{auditTypes}\ \textit{newElements}? \cap \textit{ENROL\_ELEMENTS} = \{\textit{enrolmentCompleteElement}\} \\
\\
\exists_1\, \textit{element} : \textit{AuditC} \bullet \textit{element} \in \textit{newElements}? \\
\quad \wedge\ \textit{element}.\textit{elementId} = \textit{enrolmentCompleteElement} \\
\quad \wedge\ \textit{element}.\textit{logTime} \in \textit{nowC} \mathrel{..} \textit{nowC}' \\
\quad \wedge\ \textit{element}.\textit{user} = \textit{noUser} \\
\quad \wedge\ \textit{element}.\textit{severity} = \textit{information} \\
\quad \wedge\ \textit{element}.\textit{description} = \textit{noDescription} \\
\hline
\end{array}
$$

▷ See: *EnrolContextC* (p. 102), *FloppyC* (p. 36), *UpdateKeyStoreFromFloppyC* (p. 68), *waitingEnrol* (p. 37), *EnrolmentDataOKC* (p. 104), *welcomeAdminC* (p. 23), *quiescent* (p. 37), *welcome* (p. 22), *ENROL_ELEMENTS* (p. 29), *enrolmentCompleteElement* (p. 28), *AuditC* (p. 30), *noUser* (p. 30), *information* (p. 28), *noDescription* (p. 30)

FD.Enclave.ValidateEnrolmentDataFail
FS.Enclave.ValidateEnrolmentDataFail

If the enrolment fails then TIS waits for the floppy to be removed before prompting for new enrolment data.

ValidateEnrolmentDataFailC

$$
\begin{array}{l}
EnrolContextC \\
\Xi KeyStoreC \\
\Xi FloppyC \\
\hline
enclaveStatusC = waitingEnrol \\
\neg\ EnrolmentDataOKC \\
currentScreenC'.screenMsgC = enrolmentFailedC \\
enclaveStatusC' = waitingEndEnrol \\
statusC' = statusC \\
currentDisplayC' = blank \\
auditTypes\ newElements? \cap ENROL\_ELEMENTS = \{enrolmentFailedElement\} \\
\exists_1\ element : AuditC \bullet element \in newElements? \\
\quad \land\ element.elementId = enrolmentFailedElement \\
\quad \land\ element.logTime \in nowC \mathrel{..} nowC' \\
\quad \land\ element.user = noUser \\
\quad \land\ element.severity = warning
\end{array}
$$

▷ See: *EnrolContextC* (p. 102), *KeyStoreC* (p. 33), *FloppyC* (p. 36), *waitingEnrol* (p. 37), *EnrolmentDataOKC* (p. 104), *enrolmentFailedC* (p. 23), *waitingEndEnrol* (p. 37), *blank* (p. 22), *ENROL_ELEMENTS* (p. 29), *enrolmentFailedElement* (p. 28), *AuditC* (p. 30), *noUser* (p. 30), *warning* (p. 28)

▷ The value of the *description* is left free here as the description component of the audit element may contain information relating to the reason that the enrolment data failed. This is not formally stated.

$$ValidateEnrolmentDataC \mathrel{\widehat{=}} ValidateEnrolmentDataOKC \lor ValidateEnrolmentDataFailC$$

▷ See: *ValidateEnrolmentDataOKC* (p. 105), *ValidateEnrolmentDataFailC* (p. 105)

7.1.3 Completing a failed Enrolment

A failed enrolment will only terminate once the floppy has been removed, otherwise the system would repeatedly try to validate the same floppy.

FD.Enclave.FailedEnrolFloppyRemoved
FS.Enclave.FailedEnrolFloppyRemoved

Once the floppy has been removed the administrator is prompted for enrolment data again. We do not log the removal of the floppy in the audit log.

$$
\begin{array}{|l}
\hline
\textit{FailedEnrolFloppyRemovedC} \\
\textit{EnrolContextC} \\
\Xi \textit{FloppyC} \\
\Xi \textit{KeyStoreC} \\
\hline
\textit{enclaveStatusC} = \textit{waitingEndEnrol} \\
\textit{floppyPresenceC} = \textit{absent} \\
\textit{currentScreenC}'.\textit{screenMsgC} = \textit{insertEnrolmentDataC} \\
\textit{enclaveStatusC}' = \textit{notEnrolled} \\
\textit{statusC}' = \textit{statusC} \\
\textit{currentDisplayC}' = \textit{blank} \\
\textit{auditTypes}\ \textit{newElements}? \cap \textit{ENROL\_ELEMENTS} = \emptyset \\
\hline
\end{array}
$$

▷ See: *EnrolContextC* (p. 102), *FloppyC* (p. 36), *KeyStoreC* (p. 33), *waitingEndEnrol* (p. 37), *absent* (p. 11), *notEnrolled* (p. 37), *blank* (p. 22), *ENROL_ELEMENTS* (p. 29)

FD.Enclave.WaitingFloppyRemoval
FS.Enclave.WaitingFloppyRemoval

$$
\begin{array}{|l}
\hline
\textit{WaitingFloppyRemovalC} \\
\textit{EnclaveContextC} \\
\Xi \textit{IDStationC} \\
\hline
\textit{enclaveStatusC} = \textit{waitingEndEnrol} \\
\textit{floppyPresenceC} = \textit{present} \\
\hline
\end{array}
$$

▷ See: *EnclaveContextC* (p. 102), *IDStationC* (p. 38), *waitingEndEnrol* (p. 37), *present* (p. 11)

$$\textit{CompleteFailedEnrolmentC} \mathrel{\widehat=} \textit{FailedEnrolFloppyRemovedC} \lor \textit{WaitingFloppyRemovalC}$$

▷ See: *FailedEnrolFloppyRemovedC* (p. 106), *WaitingFloppyRemovalC* (p. 107)

7.1.4 The Complete Enrolment

The complete enrolment process involves reading the enrolment data, validating it and, in the case of a failure waiting for the system to be in a state where it can try another enrolment.

$$
\begin{aligned}
\textit{TISEnrolOpC} \mathrel{\widehat=} (&\textit{ReadEnrolmentDataC} \lor \textit{ValidateEnrolmentDataC} \\
&\lor \textit{CompleteFailedEnrolmentC}) \setminus (\textit{newElements}?)
\end{aligned}
$$

▷ See: *ReadEnrolmentDataC* (p. 104), *ValidateEnrolmentDataC* (p. 106), *CompleteFailedEnrolmentC* (p. 107)

7.2 Administrator Token Tear

The action of removing the administrator Token will result in the administrator being logged out of the system.

This may happen at any point once a token has been inserted into the reader. As soon as the adminitrator's token is torn this action will be logged.

$$
\begin{array}{|l}
\hline
\textit{AdminTokenTearC} \\
\textit{EnclaveContextC} \\
\textit{AddElementsToLogC} \\
\textit{LogChangeC} \\
\\
\textit{ClearAdminToken} \\
\Xi \textit{ConfigC} \\
\Xi \textit{FloppyC} \\
\textit{ResetScreenMessageC} \\
\hline
\textit{adminTokenPresenceC} = \textit{absent} \\
\\
\textit{currentScreenC}'.\textit{screenMsgC} = \textit{welcomeAdminC} \\
\textit{statusC}' = \textit{statusC} \\
\textit{currentDisplayC}' = \textit{currentDisplayC} \\
\\
\textit{enclaveStatusC}' = \textit{enclaveQuiescent} \\
\hline
\end{array}
$$

▷ See: *EnclaveContextC* (p. 102), *AddElementsToLogC* (p. 51), *LogChangeC* (p. 61), *ClearAdminToken* (p. 74), *ConfigC* (p. 27), *FloppyC* (p. 36), *ResetScreenMessageC* (p. 83), *absent* (p. 11), *welcomeAdminC* (p. 23)

If the admin token is torn while the system is processing an activity within the enclave then the activity will be stopped.

$$
\begin{array}{|l}
\hline
\textit{BadAdminTokenTearC} \\
\textit{AdminTokenTearC} \\
\hline
\textit{AdminHasDeparted} \\
\textit{enclaveStatusC} \in \{\textit{gotAdminToken}, \textit{waitingStartAdminOp}, \textit{waitingFinishAdminOp}\} \\
\\
\textit{auditTypes}\ \textit{newElements}? \cap \textit{ADMIN\_ELEMENTS} = \{\textit{adminTokenRemovedElement}\} \\
\\
\exists_1\ \textit{element} : \textit{AuditC} \bullet \textit{element} \in \textit{newElements}? \\
\quad \land \textit{element}.\textit{elementId} = \textit{adminTokenRemovedElement} \\
\quad \land \textit{element}.\textit{logTime} \in \textit{nowC} \mathrel{..} \textit{nowC}' \\
\quad \land \textit{element}.\textit{user} = \textit{extractUser}\ \textit{currentAdminTokenC} \\
\quad \land \textit{element}.\textit{severity} = \textit{warning} \\
\quad \land \textit{element}.\textit{description} = \textit{noDescription} \\
\hline
\end{array}
$$

▷ See: *AdminTokenTearC* (p. 107), *AdminHasDeparted* (p. 80), *waitingStartAdminOp* (p. 37), *waitingFinishAdminOp* (p. 37), *ADMIN_ELEMENTS* (p. 29), *adminTokenRemovedElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *warning* (p. 28), *noDescription* (p. 30)

FD.Enclave.LoginAborted
FS.Enclave.LoginAborted

If the token is torn during the log on validation process then there is no need to log off the administrator.

$$
\begin{array}{|l}
\hline
\textit{LoginAbortedC} \\
\textit{BadAdminTokenTearC} \\
\Xi \textit{AdminC} \\
\hline
\textit{enclaveStatusC} = \textit{gotAdminToken} \\
\hline
\end{array}
$$

▷ See: *BadAdminTokenTearC* (p. 108), *AdminC* (p. 35)

7.3 Administrator Login

An Administrator logs into TIS by inserting a valid token into the *adminToken* reader. The authorisation certificate is verified and the user is logged in with the privileges indicated on the card.

Once the administrator is successfully logged into TIS, the system records that there is a role present. The process of logging on is given by the state transition diagram in Figure 7.2

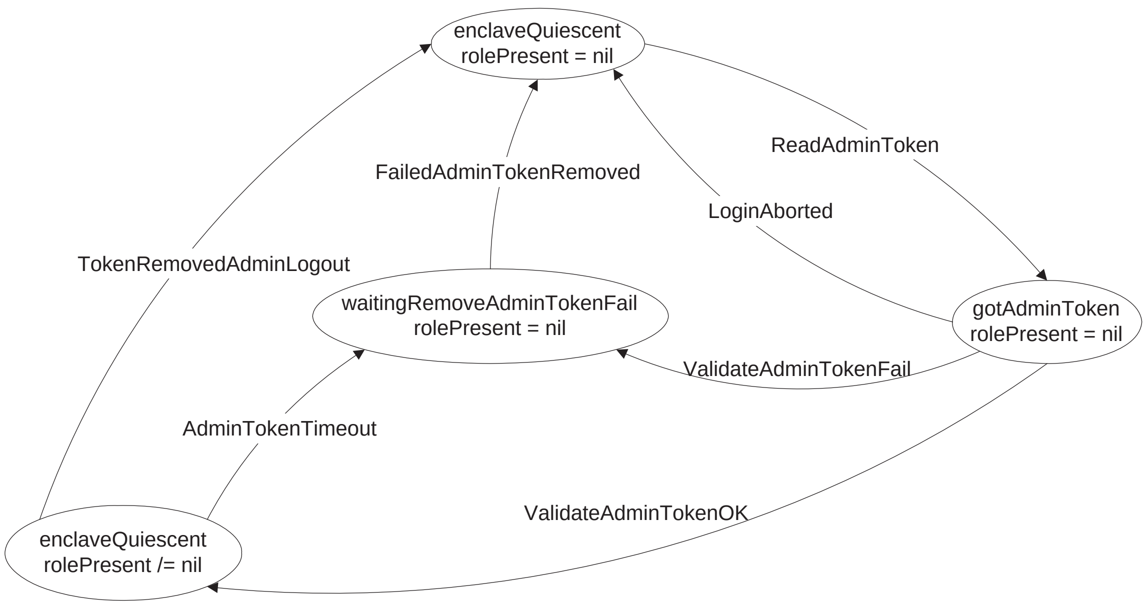

Figure 7.2: Administrator logon state transitions

The context for administrator login is given below.

```
LoginContextC
EnclaveContextC

ΞKeyStoreC
ΞDoorLatchAlarmC
ΞConfigC
AddElementsToLogC
LogChangeC
──────────
statusC′ = statusC
currentDisplayC′ = currentDisplayC
```

▷ See: *EnclaveContextC* (p. 102), *KeyStoreC* (p. 33), *DoorLatchAlarmC* (p. 35), *ConfigC* (p. 27), *AddElementsToLogC* (p. 51), *LogChangeC* (p. 61)

▷ The following state components may change *AdminC*, *InternalC* and *AuditLogC*.

7.3.1 Read Administrator Token

FD.Enclave.GetPresentAdminToken
FS.Enclave.ReadAdminToken

When the admin token is read the action is audited and the internal status changes. No other aspects of the system are modified.

An administrator can only log on when there is no user entry activity in progress or TIS is waiting for a failed user token to be removed from the token reader outside of the enclave.

$$
\begin{array}{|l}
\hline
\textit{GetPresentAdminTokenC} \\
\textit{LoginContextC} \\
\Xi \textit{AdminC} \\
\textit{ReadAdminTokenC} \\
\hline
\textit{AdminLogonCanStart} \\
\textit{enclaveStatusC}' = \textit{gotAdminToken} \\
\textit{currentScreenC}' = \textit{currentScreenC} \\
\textit{auditTypes}\ \textit{newElements}? \cap \textit{ADMIN\_ELEMENTS} = \{\textit{adminTokenPresentElement}\} \\
\exists_1\ \textit{element} : \textit{AuditC} \bullet \textit{element} \in \textit{newElements}? \\
\quad \wedge\ \textit{element}.\textit{elementId} = \textit{adminTokenPresentElement} \\
\quad \wedge\ \textit{element}.\textit{logTime} \in \textit{nowC} \mathrel{..} \textit{nowC}' \\
\quad \wedge\ \textit{element}.\textit{user} = \textit{extractUser}\ \textit{currentAdminTokenC}' \\
\quad \wedge\ \textit{element}.\textit{severity} = \textit{information} \\
\quad \wedge\ \textit{element}.\textit{description} = \textit{noDescription} \\
\hline
\end{array}
$$

▷ See: *LoginContextC* (p. 109), *AdminC* (p. 35), *ReadAdminTokenC* (p. 44), *AdminLogonCanStart* (p. 81), *ADMIN_ELEMENTS* (p. 29), *adminTokenPresentElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *information* (p. 28), *noDescription* (p. 30)

The operation to read the token is as follows:

$$\textit{TISReadAdminTokenC} \mathrel{\widehat{=}} \textit{GetPresentAdminTokenC} \setminus (\textit{newElements}?)$$

▷ See: *GetPresentAdminTokenC* (p. 110)

7.3.2 Validate Administrator Token

An administrator's token is considered valid if it contains a valid and current authorisation certificate. Additionally the privileges assigned to the user within the authorisation certificate must indicate that the user is actually an administrator.

FD.Enclave.ValidateAdminTokenOK
FS.Enclave.ValidateAdminTokenOK

If the token can be validated then the administrator is logged onto TIS.

$$
\begin{array}{l}
\textbf{schema } \mathit{ValidateAdminTokenOKC} \\
\quad \mathit{LoginContextC} \\
\quad \Xi \mathit{AdminTokenC} \\
\hline
\quad \neg\ \mathit{AdminHasDeparted} \\
\quad \mathit{enclaveStatusC} = \mathit{gotAdminToken} \\
\quad \mathit{AdminTokenOKC} \\
\quad \mathit{currentScreenC}'.\mathit{screenMsgC} = \mathit{requestAdminOpC} \\
\quad \mathit{enclaveStatusC}' = \mathit{enclaveQuiescent} \\
\quad \exists\, \mathit{requiredRole}? : \mathit{ADMINPRIVILEGE} \bullet \mathit{AdminLogonC} \\
\qquad \land\ \mathit{requiredRole}? = (\mathit{extractAuthCert}\ (\mathit{the}\ (\mathit{goodTC}^{\sim}\ \mathit{currentAdminTokenC}).\mathit{authCertC})).\mathit{roleC} \\
\quad \mathit{auditTypes}\ \mathit{newElements}? \cap \mathit{ADMIN\_ELEMENTS} = \{\mathit{adminTokenValidElement}\} \\
\quad \exists_1\, \mathit{element} : \mathit{AuditC} \bullet \mathit{element} \in \mathit{newElements}? \\
\qquad \land\ \mathit{element}.\mathit{elementId} = \mathit{adminTokenValidElement} \\
\qquad \land\ \mathit{element}.\mathit{logTime} \in \mathit{nowC} \mathinner{\ldotp\ldotp} \mathit{nowC}' \\
\qquad \land\ \mathit{element}.\mathit{user} = \mathit{extractUser}\ \mathit{currentAdminTokenC}' \\
\qquad \land\ \mathit{element}.\mathit{severity} = \mathit{information} \\
\qquad \land\ \mathit{element}.\mathit{description} = \mathit{noDescription}
\end{array}
$$

▷ See: *LoginContextC* (p. 109), *AdminTokenC* (p. 36), *AdminHasDeparted* (p. 80), *AdminTokenOKC* (p. 74), *ADMINPRIVILEGE* (p. 34), *AdminLogonC* (p. 76), *goodTC* (p. 22), *ADMIN_ELEMENTS* (p. 29), *adminTokenValidElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *information* (p. 28), *noDescription* (p. 30)

FD.Enclave.ValidateAdminTokenFail
FS.Enclave.ValidateAdminTokenFail

If the token can not be validated then TIS waits for it to be removed.

$$
\begin{array}{l}
\textbf{schema } \mathit{ValidateAdminTokenFailC} \\
\quad \mathit{LoginContextC} \\
\quad \Xi \mathit{AdminTokenC} \\
\quad \Xi \mathit{AdminC} \\
\hline
\quad \neg\ \mathit{AdminHasDeparted} \\
\quad \mathit{enclaveStatusC} = \mathit{gotAdminToken} \\
\quad \neg\ \mathit{AdminTokenOKC} \\
\quad \mathit{currentScreenC}'.\mathit{screenMsgC} = \mathit{removeAdminTokenC} \\
\quad \mathit{enclaveStatusC}' = \mathit{waitingRemoveAdminTokenFail} \\
\quad \mathit{auditTypes}\ \mathit{newElements}? \cap \mathit{ADMIN\_ELEMENTS} = \{\mathit{adminTokenInvalidElement}\} \\
\quad \exists_1\, \mathit{element} : \mathit{AuditC}; \mathit{description}! : \mathit{TEXT} \bullet \\
\qquad \mathit{element} \in \mathit{newElements}? \\
\qquad \land\ \mathit{element}.\mathit{elementId} = \mathit{adminTokenInvalidElement} \\
\qquad \land\ \mathit{element}.\mathit{logTime} \in \mathit{nowC} \mathinner{\ldotp\ldotp} \mathit{nowC}' \\
\qquad \land\ \mathit{element}.\mathit{user} = \mathit{extractUser}\ \mathit{currentAdminTokenC}' \\
\qquad \land\ \mathit{element}.\mathit{severity} = \mathit{warning} \\
\qquad \land\ \mathit{element}.\mathit{description} = \mathit{description}! \land \mathit{AdminTokenNotOK}
\end{array}
$$

▷ See: *LoginContextC* (p. 109), *AdminTokenC* (p. 36), *AdminC* (p. 35), *AdminHasDeparted* (p. 80), *AdminTokenOKC* (p. 74), *removeAdminTokenC* (p. 23), *waitingRemoveAdminTokenFail* (p. 37), *ADMIN_ELEMENTS* (p. 29), *adminTokenInvalidElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *warning* (p. 28), *AdminTokenNotOK* (p. 75)

▷ *AdminTokenNotOK* defines the value of the descriptive text applicable based on the reason for the unacceptability of the token.

$$TISValidateAdminTokenC \mathrel{\widehat=} (ValidateAdminTokenOKC \lor ValidateAdminTokenFailC \lor LoginAbortedC) \setminus (newElements?)$$

▷ See: *ValidateAdminTokenOKC* (p. 110), *ValidateAdminTokenFailC* (p. 111), *LoginAbortedC* (p. 108)

7.3.3 Complete Failed Administrator Logon

If an administrator token has failed to be accepted by TIS then no further actions can take place in the enclave until it has been removed.

FD.Enclave.FailedAdminTokenRemoved
FS.Enclave.FailedAdminTokenRemoved

The administrator token may be removed at any point during a user entry, hence the context for this activity does not place restrictions on the value of *status*.

When the admin token is removed TIS returns to a state ready to accept another administrator logon.

FailedAdminTokenRemovedC

$$LoginContextC$$
$$\Xi AdminC$$
$$ClearAdminToken$$

$$AdminHasDeparted$$
$$enclaveStatusC = waitingRemoveAdminTokenFail$$
$$currentScreenC'.screenMsgC = welcomeAdminC$$
$$enclaveStatusC' = enclaveQuiescent$$
$$statusC' = statusC$$
$$currentDisplayC' = currentDisplayC$$
$$auditTypes\ newElements? \cap ADMIN\_ELEMENTS = \{adminTokenRemovedElement\}$$
$$\exists_1\ element : AuditC \bullet element \in newElements? \land element.elementId = adminTokenRemovedElement \land element.logTime \in nowC \mathinner{..} nowC' \land element.user = extractUser\ currentAdminTokenC \land element.severity = information \land element.description = noDescription$$

▷ See: *LoginContextC* (p. 109), *AdminC* (p. 35), *ClearAdminToken* (p. 74), *AdminHasDeparted* (p. 80), *waitingRemoveAdminTokenFail* (p. 37), *welcomeAdminC* (p. 23), *ADMIN_ELEMENTS* (p. 29), *adminTokenRemovedElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *information* (p. 28), *noDescription* (p. 30)

The case where the token is not removed will be captured within the model of the system being idle.

$$TISCompleteFailedAdminLogonC \mathrel{\widehat=} FailedAdminTokenRemovedC$$

▷ See: *FailedAdminTokenRemovedC* (p. 112)

7.3.4 The Complete Administrator Logon

FD.Enclave.TISAdminLogin
FS.Enclave.TISAdminLogin

The complete administrator logon process, from the point that the system has detected the presence of a token in the administrator reader, involves validating the administrator token and, in the case of a failure waiting for the system to be in a state where it can try another logon.

$$TISAdminLogonC \mathrel{\widehat{=}} TISReadAdminTokenC \lor TISValidateAdminTokenC \lor TISCompleteFailedAdminLogonC$$

▷ See: *TISReadAdminTokenC* (p. 110), *TISValidateAdminTokenC* (p. 112), *TISCompleteFailedAdminLogonC* (p. 112)

This can be divided into starting the administrator logon:

$$TISStartAdminLogonC \mathrel{\widehat{=}} TISReadAdminTokenC$$

▷ See: *TISReadAdminTokenC* (p. 110)

and progressing the logon to completion.

$$TISProgressAdminLogon \mathrel{\widehat{=}} TISValidateAdminTokenC \lor TISCompleteFailedAdminLogonC$$

▷ See: *TISValidateAdminTokenC* (p. 112), *TISCompleteFailedAdminLogonC* (p. 112)

7.4 Administrator Logout

Administrator logout can be achieved in two ways, either the administrator removes their token from TIS, or the Authorisation certificate on the token expires, causing the system to automatically log off the administrator.

FD.Enclave.AdminLogout
FS.Enclave.AdminLogout

If TIS is not performing an administrator operation then the token may be removed to log out the administrator.

$$\begin{array}{l}
\textbf{schema } TokenRemovedAdminLogoutC \\
\quad AdminTokenTearC \\
\quad AdminLogoutC \\
\\
\quad ClearAdminToken \\
\hline
\quad PresentAdminHasDeparted \\
\quad enclaveStatusC = enclaveQuiescent \\
\\
\quad auditTypes\ newElements? \cap ADMIN_ELEMENTS = \{adminTokenRemovedElement\} \\
\\
\quad \exists_1\ element : AuditC \bullet element \in newElements? \\
\qquad \land element.elementId = adminTokenRemovedElement \\
\qquad \land element.logTime \in nowC \mathrel{..} nowC' \\
\qquad \land element.user = extractUser\ currentAdminTokenC \\
\qquad \land element.severity = information \\
\qquad \land element.description = noDescription
\end{array}$$

▷ See: *AdminTokenTearC* (p. 107), *AdminLogoutC* (p. 76), *ClearAdminToken* (p. 74), *PresentAdminHasDeparted* (p. 78), *ADMIN_ELEMENTS* (p. 29), *adminTokenRemovedElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *information* (p. 28), *noDescription* (p. 30)

FD.Enclave.BadAdminLogout
FS.Enclave.BadAdminLogout

If the administrator is performing an operation (other than shutdown) when the token is torn then the administrator will be logged off.

BadAdminLogoutC
BadAdminTokenTearC
AdminLogoutC

PresentAdminHasDeparted
$enclaveStatusC \in \{waitingStartAdminOp, waitingFinishAdminOp\}$

▷ See: *BadAdminTokenTearC* (p. 108), *AdminLogoutC* (p. 76), *PresentAdminHasDeparted* (p. 78), *waitingStartAdminOp* (p. 37), *waitingFinishAdminOp* (p. 37)

FD.Enclave.AdminTokenTimeout
FS.Enclave.AdminTokenTimeout

The TIS will automatically logout an administrator whose token expires. This occurs if the validity period on the Authorisation certificate expires.

AdminTokenTimeoutC
LoginContextC

AdminLogoutC
AddElementsToLogC
ResetScreenMessageC

AdminTokenHasExpired

$enclaveStatusC' = waitingRemoveAdminTokenFail$

$auditTypes\ newElements? \cap ADMIN\_ELEMENTS = \{adminTokenExpiredElement\}$

$$\exists_1\ element : AuditC \bullet element \in newElements? \\ \quad \land element.elementId = adminTokenExpiredElement \\ \quad \land element.logTime \in nowC \mathrel{..} nowC' \\ \quad \land element.user = extractUser\ currentAdminTokenC \\ \quad \land element.severity = warning \\ \quad \land element.description = noDescription$$

▷ See: *LoginContextC* (p. 109), *AdminLogoutC* (p. 76), *AddElementsToLogC* (p. 51), *ResetScreenMessageC* (p. 83), *AdminTokenHasExpired* (p. 78), *waitingRemoveAdminTokenFail* (p. 37), *ADMIN_ELEMENTS* (p. 29), *adminTokenExpiredElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *warning* (p. 28), *noDescription* (p. 30)

FD.Enclave.TISCompleteTimeoutAdminLogout
FS.Enclave.TISCompleteTimeoutAdminLogout

If the administrator's token expires then it must be removed before further activities can take place at the TIS console. The behaviour and conditions are identical to the behaviour when the system waits for a the administrator to remove their token following a failed logon.

$$TISCompleteTimeoutAdminLogoutC \mathrel{\widehat{=}} TISCompleteFailedAdminLogonC$$

▷ See: *TISCompleteFailedAdminLogonC* (p. 112)

7.4.1 Complete Administrator Logout

FD.Enclave.TISAdminLogout
FS.Enclave.TISAdminLogout

The complete administrator logout process which must be performed as soon as an Administrator needs to be logged out is given below.

$$\begin{aligned} TISAdminLogoutC \mathrel{\widehat{=}} & (TokenRemovedAdminLogoutC \lor AdminTokenTimeoutC \\ & \lor BadAdminLogoutC) \setminus (newElements?) \end{aligned}$$

▷ See: *TokenRemovedAdminLogoutC* (p. 113), *AdminTokenTimeoutC* (p. 114), *BadAdminLogoutC* (p. 114)

7.5 Administrator Operations

An administrator operation can take place as long as an administrator is present. The operation is started by receiving a valid request to perform an operation from the keyboard. TIS will ensure that the requested operation is one compatible with the current role present.

Once the operation is started the behaviour depends on the type of operation. Operations are either short, and can be implemented in one phase or they are multi-phase operations.

shutdown and *overrideLock* are short operations, while *archiveLog* and *updateCofigData* are multi phase operations.

The state transition diagram for administrator operations is given in Figure 7.3

All administrator operations have a common context, in which the *AdminToken* does not change. An administrator can only perform an operation when there is no user entry activity in progress or TIS is waiting for a failed user token to be removed from the token reader outside of the enclave.

```
┌─ AdminOpContextC ──────────────────────────
│ EnclaveContextC
│
│ ΞKeyStoreC
│ ΞAdminTokenC
│ AddElementsToLogC
│ LogChangeC
└────────────────────────────────────────────
```

▷ See: *EnclaveContextC* (p. 102), *KeyStoreC* (p. 33), *AdminTokenC* (p. 36), *AddElementsToLogC* (p. 51), *LogChangeC* (p. 61)

▷ The following state components may change *FloppyC*, *ConfigC*, *AdminC*, *DoorLatchAlarmC* and *AuditLogC*.

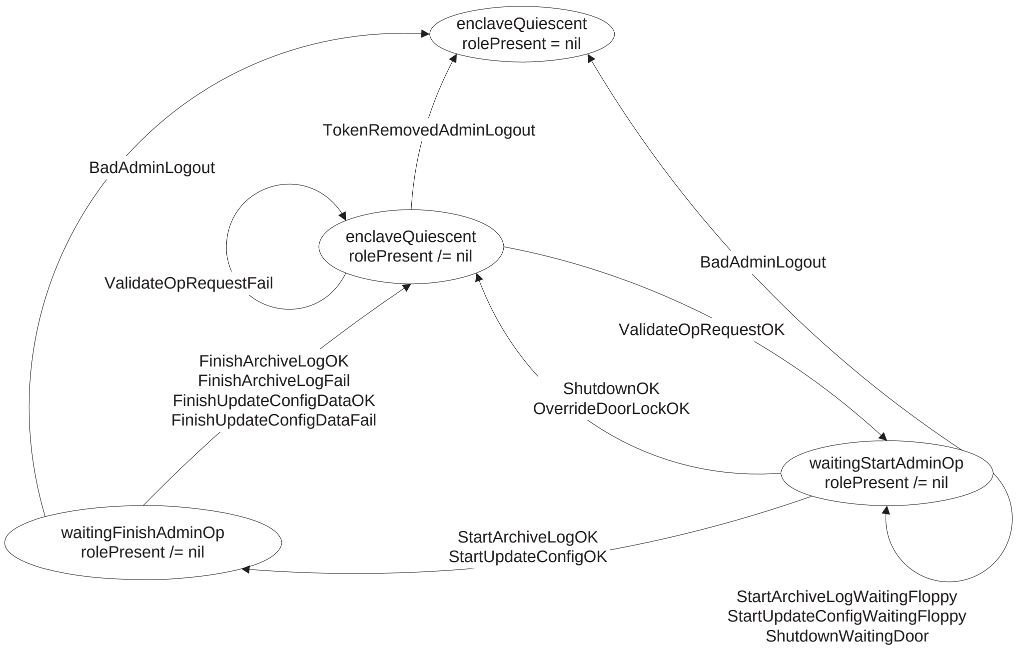

Figure 7.3: Administrator operation state transitions

Once an operation has been started its context is given by:

AdminOpStartedContextC

AdminOpContextC

$\neg$ *AdminHasDeparted*
$enclaveStatusC = waitingStartAdminOp$

$statusC' = statusC$

▷ See: *AdminOpContextC* (p. 115), *AdminHasDeparted* (p. 80), *waitingStartAdminOp* (p. 37)

Some operations are multi-phase, the context for completing a multi-phase operation is given by:

AdminOpFinishContextC

AdminOpContextC

AdminFinishOpC

$\neg$ *AdminHasDeparted*
$enclaveStatusC = waitingFinishAdminOp$

$statusC' = statusC$
$currentDisplayC' = currentDisplayC$

$enclaveStatusC' = enclaveQuiescent$

▷ See: *AdminOpContextC* (p. 115), *AdminFinishOpC* (p. 77), *AdminHasDeparted* (p. 80), *waitingFinishAdminOp* (p. 37)

7.6 Starting Operations

All administrator operations are initiated in the same way. This involves validating the latest keyboard input and determining whether it is a valid operation request.

TIS only attempts to start an operation if there is an administrator present and there is no current activity in the enclave. An administrator can only start an operation when there is no user entry activity in progress or TIS is waiting for a failed user token to be removed from the token reader outside of the enclave.

$$
\begin{array}{l}
\textbf{StartOpContextC} \\
\hline
EnclaveContextC \\[4pt]
\Xi DoorLatchAlarmC \\
\Xi ConfigC \\
\Xi FloppyC \\
\Xi KeyStoreC \\
\Xi AdminTokenC \\
AddElementsToLogC \\
LogChangeC \\
\hline
AdminOpCanStart \\[4pt]
statusC' = statusC \\
currentDisplayC' = currentDisplayC
\end{array}
$$

▷ See: *EnclaveContextC* (p. 102), *DoorLatchAlarmC* (p. 35), *ConfigC* (p. 27), *FloppyC* (p. 36), *KeyStoreC* (p. 33), *AdminTokenC* (p. 36), *AddElementsToLogC* (p. 51), *LogChangeC* (p. 61), *AdminOpCanStart* (p. 81)

▷ The following state components may change *InternalC*, *AdminC* and *AuditLogC*.

▷ We strengthen the precondition of this context to give priority to starting a user entry over starting an administrator operation.

7.6.1 Validating an Operation Request

FD.Enclave.ValidateOpRequestOK
FS.Enclave.ValidateOpRequestOK

Once the data from the keyboard has been read this must be validated to ensure it corresponds to a valid operation.

$$keyedDataText : KEYBOARD \longrightarrow TEXT$$

▷ See: *KEYBOARD* (p. 23)

ValidateOpRequestOKC

$StartOpContextC$

$keyedDataPresenceC = present$
$\exists request? : KEYBOARD \bullet request? = keyboardC \land AdminOpIsAvailable$

$currentScreenC'.screenMsgC = doingOpC$

$enclaveStatusC' = waitingStartAdminOp$

$\exists requestedOp? : ADMINOP \bullet requestedOp? = keyedOps^{\sim} keyboardC$
$\quad \land AdminStartOpC$

$auditTypes\ newElements? \cap ADMIN\_ELEMENTS = \{operationStartElement\}$

$\exists_1 element : AuditC \bullet element \in newElements?$
$\quad \land element.elementId = operationStartElement$
$\quad \land element.logTime \in nowC \mathrel{..} nowC'$
$\quad \land element.user = extractUser\ currentAdminTokenC$
$\quad \land element.severity = information$
$\quad \land element.description = keyedDataText\ keyboardC$

▷ See: *StartOpContextC* (p. 117), *present* (p. 11), *KEYBOARD* (p. 23), *AdminOpIsAvailable* (p. 77), *doingOpC* (p. 23), *waitingStartAdminOp* (p. 37), *ADMINOP* (p. 34), *keyedOps* (p. 23), *AdminStartOpC* (p. 76), *ADMIN_ELEMENTS* (p. 29), *operationStartElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *information* (p. 28), *keyedDataText* (p. 117)

FD.Enclave.ValidateOpRequestFail
FS.Enclave.ValidateOpRequestFail

If the data from the keyboard doesn't correspond to an operation that can be performed at present then the operation is not started and the attempt to start an illegal operation is logged.

ValidateOpRequestFailC

$StartOpContextC$

$\Xi AdminC$

$keyedDataPresenceC = present$
$\exists request? : KEYBOARD \bullet request? = keyboardC \land \lnot AdminOpIsAvailable$

$currentScreenC'.screenMsgC = invalidRequestC$

$enclaveStatusC' = enclaveStatusC$

$auditTypes\ newElements? \cap ADMIN\_ELEMENTS = \{invalidOpRequestElement\}$

$\exists_1 element : AuditC \bullet element \in newElements?$
$\quad \land element.elementId = invalidOpRequestElement$
$\quad \land element.logTime \in nowC \mathrel{..} nowC'$
$\quad \land element.user = extractUser\ currentAdminTokenC$
$\quad \land element.severity = warning$
$\quad \land element.description = keyedDataText\ keyboardC$

▷ See: *StartOpContextC* (p. 117), *AdminC* (p. 35), *present* (p. 11), *KEYBOARD* (p. 23), *AdminOpIsAvailable* (p. 77), *invalidRequestC* (p. 23), *ADMIN_ELEMENTS* (p. 29), *invalidOpRequestElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *warning* (p. 28), *keyedDataText* (p. 117)

FD.Enclave.NoOpRequest
FS.Enclave.NoOpRequest

If there is no data at the keyboard then TIS waits for user interaction.

$$
\begin{array}{|l}
\textit{NoOpRequestC} \\
\hline
\textit{StartOpContextC} \\
\Xi \textit{IDStationC} \\
\hline
\textit{keyedDataPresenceC} = \textit{absent}
\end{array}
$$

▷ See: *StartOpContextC* (p. 117), *IDStationC* (p. 38), *absent* (p. 11)

$$\textit{ValidateOpRequestC} \mathrel{\widehat{=}} \textit{ValidateOpRequestOKC} \lor \textit{ValidateOpRequestFailC} \lor \textit{NoOpRequestC}$$

▷ See: *ValidateOpRequestOKC* (p. 117), *ValidateOpRequestFailC* (p. 118), *NoOpRequestC* (p. 119)

7.6.2 Complete Operation Start

FD.Enclave.TISStartAdminOp
FS.Enclave.TISStartAdminOp

The process of starting an administrator operation involves exactly the validation of an operation request.

$$\textit{TISStartAdminOpC} \mathrel{\widehat{=}} \textit{ValidateOpRequestC}$$

▷ See: *ValidateOpRequestC* (p. 119)

7.7 Archiving the Log

When the log is archived it is copied to floppy and the internally held log is truncated.

The internally held log can only be truncated if the write to floppy succeeds.

To check that the archive succeeded the floppy is read back and the data compared with that held by the system.

This is a two phase operation, during the first phase the log is written to floppy, during the second phase the data on the floppy is validated.

7.7.1 Writing the archive Log

FD.Enclave.StartArchiveLogOK
FS.Enclave.StartArchiveLogOK

The first phase of this operation is to write the archive log to floppy.

$$
\begin{array}{|l}
\textit{StartArchiveLogOKC} \\
\hline
\textit{EnclaveContextC} \\
\Xi \textit{AdminTokenC} \\
\Xi \textit{KeyboardC} \\
\Xi \textit{KeyStoreC} \\
\Xi \textit{Config} \\
\Xi \textit{Admin} \\
\Xi \textit{AdminToken} \\
\textit{LogChangeC} \\
\textit{newElements}? : \mathbb{F}\ \textit{AuditC} \\
\hline
\neg\ \textit{AdminHasDeparted} \\
\textit{enclaveStatusC} = \textit{waitingStartAdminOp} \\
\textit{the}\ \textit{currentAdminOpC} = \textit{archiveLog} \\
\textit{floppyPresenceC} = \textit{present} \\
\textit{floppyPresenceC}' = \textit{floppyPresenceC} \\
\textit{currentFloppyC}' = \textit{currentFloppyC} \\
\textit{currentScreenC}'.\textit{screenMsgC} = \textit{doingOpC} \\
\textit{statusC}' = \textit{statusC} \\
\textit{enclaveStatusC}' = \textit{waitingFinishAdminOp} \\
(\exists\ \textit{archive}! : \mathbb{F}\ \textit{AuditC} \bullet \textit{ArchiveLogC} \land \textit{writtenFloppyC}' = \textit{auditFileC}\ \textit{archive}!) \\
\hline
\end{array}
$$

▷ See: *EnclaveContextC* (p. 102), *AdminTokenC* (p. 36), *KeyboardC* (p. 36), *KeyStoreC* (p. 33), *LogChangeC* (p. 61), *AuditC* (p. 30), *AdminHasDeparted* (p. 80), *waitingStartAdminOp* (p. 37), *archiveLog* (p. 34), *present* (p. 11), *doingOpC* (p. 23), *waitingFinishAdminOp* (p. 37), *ArchiveLogC* (p. 55)

▷ Note this operation makes other altertions to the audit log so cannot use the *AdminOpStartedContext*.

We wait indefinitely for a floppy to be present.

FD.Enclave.StartArchiveLogWaitingFloppy
FS.Enclave.StartArchiveLogWaitingFloppy

$$
\begin{array}{|l}
\textit{StartArchiveLogWaitingFloppyC} \\
\hline
\textit{AdminOpStartedContextC} \\
\Xi \textit{ConfigC} \\
\Xi \textit{AdminC} \\
\Xi \textit{FloppyC} \\
\hline
\textit{the}\ \textit{currentAdminOpC} = \textit{archiveLog} \\
\textit{floppyPresenceC} = \textit{absent} \\
\textit{currentScreenC}'.\textit{screenMsgC} = \textit{insertBlankFloppyC} \\
\textit{currentDisplayC}' = \textit{currentDisplayC} \\
\textit{enclaveStatusC}' = \textit{enclaveStatusC} \\
\hline
\end{array}
$$

▷ See: *AdminOpStartedContextC* (p. 116), *ConfigC* (p. 27), *AdminC* (p. 35), *FloppyC* (p. 36), *archiveLog* (p. 34), *absent* (p. 11), *insertBlankFloppyC* (p. 23)

$$
\begin{aligned}
\textit{StartArchiveLogC} \mathrel{\widehat=} (&(\textit{StartArchiveLogOKC} \mathbin{⨾} \textit{UpdateFloppyC}) \\
&\lor \textit{StartArchiveLogWaitingFloppyC}) \setminus (\textit{newElements}?)
\end{aligned}
$$

▷ See: *StartArchiveLogOKC* (p. 119), *UpdateFloppyC* (p. 47), *StartArchiveLogWaitingFloppyC* (p. 120)

7.7.2 Clearing the archive Log

Note this operation makes altertions to the audit log other than the addition of elements so cannot use the *AdminOpFinishContext*. We define a specific context for completing the archive log.

FinishArchiveLogContext

EnclaveContextC

Ξ*AdminTokenC*
Ξ*KeyboardC*
Ξ*KeyStoreC*
Ξ*ConfigC*
AdminFinishOpC
Ξ*DoorLatchAlarmC*
LogChangeC

$statusC' = statusC$
$currentDisplayC' = currentDisplayC$

$enclaveStatusC' = enclaveQuiescent$

▷ See: *EnclaveContextC* (p. 102), *AdminTokenC* (p. 36), *KeyboardC* (p. 36), *KeyStoreC* (p. 33), *ConfigC* (p. 27), *AdminFinishOpC* (p. 77), *DoorLatchAlarmC* (p. 35), *LogChangeC* (p. 61)

FD.Enclave.FinishArchiveLogOK
FS.Enclave.FinishArchiveLogOK

The audit log is only truncated after a check has been made to ensure that the actual floppy data matches what the system believes is on the floppy.

FinishArchiveLogOKC

FinishArchiveLogContext

ReadFloppyC
ClearLogC

$\neg\ AdminHasDeparted$
$enclaveStatusC = waitingFinishAdminOp$
$\text{the}\ currentAdminOpC = archiveLog$
$floppyPresenceC = present$

$writtenFloppyC = currentFloppyC'$

$currentScreenC'.screenMsgC = requestAdminOpC$

▷ See: *FinishArchiveLogContext* (p. 121), *ReadFloppyC* (p. 44), *ClearLogC* (p. 56), *AdminHasDeparted* (p. 80), *waitingFinishAdminOp* (p. 37), *archiveLog* (p. 34), *present* (p. 11)

FD.Enclave.FinishArchiveLogNoFloppy
FS.Enclave.FinishArchiveLogNoFloppy

If the administrator is impatient and removes the floppy early then the archive fails as the system cannot check that the archive was taken.

The audit log entry for this failure is distinguished from the failure caused by the written data failing to match by the descriptive text in the audit record.

$$floppyRemoved, floppyHasBadData : TEXT$$

FinishArchiveLogNoFloppyC

$$FinishArchiveLogContext$$
$$CancelArchive$$
$$\Xi FloppyC$$

$$\begin{array}{l}
\mathrm{the}\ currentAdminOpC = archiveLog \\
floppyPresenceC = absent \\
currentScreenC'.screenMsgC = archiveFailedC \\
auditTypes\ newElements? \cap ADMIN_ELEMENTS = \{archiveCheckFailedElement\} \\
\exists_1\ element : AuditC \bullet element \in newElements? \\
\quad \land element.elementId = archiveCheckFailedElement \\
\quad \land element.logTime \in nowC \mathrel{..} nowC' \\
\quad \land element.user = extractUser\ currentAdminTokenC \\
\quad \land element.severity = warning \\
\quad \land element.description = floppyRemoved
\end{array}$$

▷ See: *FinishArchiveLogContext* (p. 121), *CancelArchive* (p. 57), *FloppyC* (p. 36), *archiveLog* (p. 34), *absent* (p. 11), *ADMIN_ELEMENTS* (p. 29), *archiveCheckFailedElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *warning* (p. 28), *floppyRemoved* (p. 122)

FD.Enclave.FinishArchiveLogBadMatch
FS.Enclave.FinishArchiveLogBadMatch

If the data read back from the floppy does not match what the ID station believes should be on the floppy then the archive fails.

FinishArchiveLogBadMatchC

$$FinishArchiveLogContext$$
$$CancelArchive$$
$$ReadFloppyC$$

$$\begin{array}{l}
\mathrm{the}\ currentAdminOpC = archiveLog \\
floppyPresenceC = present \\
writtenFloppyC \neq currentFloppyC' \\
currentScreenC'.screenMsgC = archiveFailedC \\
auditTypes\ newElements? \cap ADMIN_ELEMENTS = \{archiveCheckFailedElement\} \\
\exists_1\ element : AuditC \bullet element \in newElements? \\
\quad \land element.elementId = archiveCheckFailedElement \\
\quad \land element.logTime \in nowC \mathrel{..} nowC' \\
\quad \land element.user = extractUser\ currentAdminTokenC \\
\quad \land element.severity = warning \\
\quad \land element.description = floppyHasBadData
\end{array}$$

▷ See: *FinishArchiveLogContext* (p. 121), *CancelArchive* (p. 57), *ReadFloppyC* (p. 44), *archiveLog* (p. 34), *present* (p. 11), *ADMIN_ELEMENTS* (p. 29), *archiveCheckFailedElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *warning* (p. 28), *floppyHasBadData* (p. 122)

$$FinishArchiveLogFailC \mathrel{\widehat=} FinishArchiveLogBadMatchC \lor FinishArchiveLogNoFloppyC$$

$$FinishArchiveLogC \mathrel{\widehat=} (FinishArchiveLogOKC \lor FinishArchiveLogFailC) \setminus (newElements?)$$

▷ See: *FinishArchiveLogBadMatchC* (p. 122), *FinishArchiveLogNoFloppyC* (p. 122), *FinishArchiveLogOKC* (p. 121)

7.7.3 The complete archive Log operation

FD.Enclave.TISArchiveLogOp
FS.Enclave.TISArchiveLogOp

Combining the start and finish phase of this operation gives the complete operation.

$$TISArchiveLogOpC \mathrel{\widehat=} StartArchiveLogC \lor FinishArchiveLogC$$

▷ See: *StartArchiveLogC* (p. 120), *FinishArchiveLogC* (p. 123)

7.8 Updating Configuration Data

The operation to update the configuration data is a two phase operation. During the first phase the configuration data is read from floppy. During the second phase the configuration data provided on the floppy is checked (currently the check is purely that the data is configuration data) and the TIS configuration data is replaced by the new data.

7.8.1 Reading Configuration Data

FD.Enclave.StartUpdateConfigDataOK
FS.Enclave.StartUpdateConfigDataOK

In order to update configuration data the administrator must supply replacement configuration data on a floppy disk.

StartUpdateConfigOKC

$$\begin{array}{l}
AdminOpStartedContextC \\[4pt]
ReadFloppyC \\
\Xi ConfigC \\
\Xi AdminC \\
\Xi DoorLatchAlarmC \\
\hline
\mathrm{the}\ currentAdminOpC = updateConfigData \\
floppyPresenceC = present \\[4pt]
currentScreenC'.screenMsgC = doingOpC \\
currentDisplayC' = currentDisplayC \\[4pt]
enclaveStatusC' = waitingFinishAdminOp
\end{array}$$

▷ See: *AdminOpStartedContextC* (p. 116), *ReadFloppyC* (p. 44), *ConfigC* (p. 27), *AdminC* (p. 35), *DoorLatchAlarmC* (p. 35), *updateConfigData* (p. 34), *present* (p. 11), *doingOpC* (p. 23), *waitingFinishAdminOp* (p. 37)

FD.Enclave.StartUpdateConfigWaitingFloppy
FS.Enclave.StartUpdateConfigWaitingFloppy

We wait indefinitely for a floppy to be present.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{StartUpdateConfigWaitingFloppyC}} \\
\hline
AdminOpStartedContextC \\
\\
\Xi FloppyC \\
\Xi ConfigC \\
\Xi AdminC \\
\Xi DoorLatchAlarmC \\
\hline
\mathit{the}\ currentAdminOpC = updateConfigData \\
floppyPresenceC = absent \\
\\
currentScreenC'.screenMsgC = insertConfigDataC \\
currentDisplayC' = currentDisplayC \\
\\
enclaveStatusC' = enclaveStatusC \\
\hline
\end{array}
$$

▷ See: *AdminOpStartedContextC* (p. 116), *FloppyC* (p. 36), *ConfigC* (p. 27), *AdminC* (p. 35), *DoorLatchAlarmC* (p. 35), *updateConfigData* (p. 34), *absent* (p. 11), *insertConfigDataC* (p. 23)

$$StartUpdateConfigDataC \mathrel{\widehat=} (StartUpdateConfigOKC \lor StartUpdateConfigWaitingFloppyC) \setminus (newElements?)$$

▷ See: *StartUpdateConfigOKC* (p. 123), *StartUpdateConfigWaitingFloppyC* (p. 124)

7.8.2 Storing Configuration Data

FD.Enclave.FinishUpdateConfigDataOK
FS.Enclave.FinishUpdateConfigDataOK

The supplied data will be used to replace the current configuration data if it is valid configuration data.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FinishUpdateConfigDataOKC}} \\
\hline
AdminOpFinishContextC \\
\\
\Xi FloppyC \\
\Xi DoorLatchAlarmC \\
\hline
\mathit{the}\ currentAdminOpC = updateConfigData \\
\\
currentFloppyC \in \operatorname{ran} configFileC \\
\\
\theta ConfigC' = configFileC^{\sim}\, currentFloppyC \\
\\
currentScreenC'.screenMsgC = requestAdminOpC \\
\\
auditTypes\ newElements? \cap ADMIN\_ELEMENTS = \{updatedConfigDataElement\} \\
\\
\exists_1\, element : AuditC \bullet element \in newElements? \\
\quad \land element.elementId = updatedConfigDataElement \\
\quad \land element.logTime \in nowC \mathrel{..} nowC' \\
\quad \land element.user = extractUser\ currentAdminTokenC \\
\quad \land element.severity = information \\
\hline
\end{array}
$$

▷ See: *AdminOpFinishContextC* (p. 116), *FloppyC* (p. 36), *DoorLatchAlarmC* (p. 35), *updateConfigData* (p. 34), *configFileC* (p. 22), *ConfigC* (p. 27), *ADMIN_ELEMENTS* (p. 29), *updatedConfigDataElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *information* (p. 28)

▷ The description within the audit element should summarise the new configuration data values. This is not formally stated here so the value of the description is left free.

FD.Enclave.FinishUpdateConfigDataFail
FS.Enclave.FinishUpdateConfigDataFail

If the supplied data is not valid configuration data the operation terminates without changing the TIS configuration data.

$$
\begin{array}{l}
\textit{FinishUpdateConfigDataFailC} \\
\hline
\textit{AdminOpFinishContextC} \\
\\
\Xi \textit{ConfigC} \\
\Xi \textit{FloppyC} \\
\Xi \textit{DoorLatchAlarmC} \\
\hline
\textit{the currentAdminOpC} = \textit{updateConfigData} \\
\\
\textit{currentFloppyC} \notin \operatorname{ran} \textit{configFileC} \\
\\
\textit{currentScreenC}'.\textit{screenMsgC} = \textit{invalidDataC} \\
\\
\textit{auditTypes newElements}? \cap \textit{ADMIN\_ELEMENTS} = \{\textit{invalidConfigDataElement}\} \\
\\
\exists_1\ \textit{element} : \textit{AuditC} \bullet \textit{element} \in \textit{newElements}? \\
\qquad \land\ \textit{element}.\textit{elementId} = \textit{invalidConfigDataElement} \\
\qquad \land\ \textit{element}.\textit{logTime} \in \textit{nowC} \ldots \textit{nowC}' \\
\qquad \land\ \textit{element}.\textit{user} = \textit{extractUser currentAdminTokenC} \\
\qquad \land\ \textit{element}.\textit{severity} = \textit{warning} \\
\qquad \land\ \textit{element}.\textit{description} = \textit{noDescription}
\end{array}
$$

▷ See: *AdminOpFinishContextC* (p. 116), *ConfigC* (p. 27), *FloppyC* (p. 36), *DoorLatchAlarmC* (p. 35), *updateConfigData* (p. 34), *configFileC* (p. 22), *invalidDataC* (p. 23), *ADMIN_ELEMENTS* (p. 29), *invalidConfigDataElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *warning* (p. 28), *noDescription* (p. 30)

$$\textit{FinishUpdateConfigDataC} \mathrel{\widehat=} (\textit{FinishUpdateConfigDataOKC} \lor \textit{FinishUpdateConfigDataFailC}) \setminus (\textit{newElements}?)$$

▷ See: *FinishUpdateConfigDataOKC* (p. 124), *FinishUpdateConfigDataFailC* (p. 125)

7.8.3 The complete update configuration data operation

FD.Enclave.TISUpdateConfigDataOp
FS.Enclave.TISUpdateConfigDataOp

Combining the start and finish phase of this operation gives the complete operation.

$$\textit{TISUpdateConfigDataOpC} \mathrel{\widehat=} \textit{StartUpdateConfigDataC} \lor \textit{FinishUpdateConfigDataC}$$

▷ See: *StartUpdateConfigDataC* (p. 124), *FinishUpdateConfigDataC* (p. 125)

7.9 Shutting Down the ID Station

Shutting down the ID Station is a single phase operation.

When the ID Station is shutdown the door is automatically locked so the system is in a secure state. The ID Station cannot be shutdown if the door is currently open, this prevents the enclave being left in an insecure state once TIS is shutdown.

FD.Enclave.ShutdownOK
FS.Enclave.ShutdownOK

ShutdownOKC

$\Delta IDStationC$
$RealWorldChangesC$

$\Xi TISControlledRealWorldC$

$ClearUserToken$
$ClearAdminToken$
$\Xi FingerC$
$\Xi StatsC$
$\Xi CertificateStore$
$\Xi Keyboard$
$\Xi KeyStore$
$\Xi ConfigC$
$\Xi FloppyC$
$LockDoorC$
$AdminLogoutC$
$AddElementsToLogC$

$enclaveStatusC = waitingStartAdminOp$
$the\ currentAdminOpC = shutdownOp$
$currentDoorC = closed$

$currentScreenC'.screenMsgC = clearC$

$enclaveStatusC' = shutdown$
$currentDisplayC' = blank$

$auditTypes\ newElements? \cap ADMIN\_ELEMENTS = \{shutdownElement\}$

$$\begin{aligned}&\exists_1\ element : AuditC \bullet element \in newElements?\\ &\quad \wedge element.elementId = shutdownElement\\ &\quad \wedge element.logTime \in nowC \mathrel{..} nowC'\\ &\quad \wedge element.user = extractUser\ currentAdminTokenC\\ &\quad \wedge element.severity = information\\ &\quad \wedge element.description = noDescription\end{aligned}$$

▷ See: *IDStationC* (p. 38), *RealWorldChangesC* (p. 40), *TISControlledRealWorldC* (p. 24), *ClearUserToken* (p. 72), *ClearAdminToken* (p. 74), *FingerC* (p. 36), *StatsC* (p. 34), *CertificateStore* (p. 34), *ConfigC* (p. 27), *FloppyC* (p. 36), *LockDoorC* (p. 65), *AdminLogoutC* (p. 76), *AddElementsToLogC* (p. 51), *waitingStartAdminOp* (p. 37), *shutdownOp* (p. 34), *closed* (p. 21), *clearC* (p. 23), *blank* (p. 22), *ADMIN_ELEMENTS* (p. 29), *shutdownElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *information* (p. 28), *noDescription* (p. 30)

▷ This operation cannot be aborted by the administrator tearing their token, hence the *AdminOpStartedContextC* cannot be used here.

FD.Enclave.ShutdownWaitingDoor
FS.Enclave.ShutdownWaitingDoor

TIS waits indefinitely for the door to be closed before completing the shutdown.

$$
\begin{array}{|l}
\textit{ShutdownWaitingDoorC} \\
\hline
\textit{AdminOpContextC} \\
\\
\Xi \textit{ConfigC} \\
\Xi \textit{FloppyC} \\
\Xi \textit{DoorLatchAlarmC} \\
\Xi \textit{AdminC} \\
\hline
\textit{enclaveStatusC} = \textit{waitingStartAdminOp} \\
\textit{the currentAdminOpC} = \textit{shutdownOp} \\
\textit{currentDoorC} = \textit{open} \\
\\
\textit{currentScreenC}'.\textit{screenMsgC} = \textit{closeDoorC} \\
\\
\textit{statusC}' = \textit{statusC} \\
\textit{enclaveStatusC}' = \textit{enclaveStatusC} \\
\textit{currentDisplayC}' = \textit{currentDisplayC} \\
\hline
\end{array}
$$

▷ See: *AdminOpContextC* (p. 115), *ConfigC* (p. 27), *FloppyC* (p. 36), *DoorLatchAlarmC* (p. 35), *AdminC* (p. 35), *waitingStartAdminOp* (p. 37), *shutdownOp* (p. 34), *open* (p. 21), *closeDoorC* (p. 23)

FD.Enclave.TISShutdownOp
FS.Enclave.TISShutdownOp

There is nothing that can go wrong with the shutdown operation.

$$\textit{TISShutdownOpC} \mathrel{\widehat{=}} (\textit{ShutdownOKC} \lor \textit{ShutdownWaitingDoorC}) \setminus (\textit{newElements}?)$$

▷ See: *ShutdownOKC* (p. 126), *ShutdownWaitingDoorC* (p. 127)

7.10 Unlocking the Enclave Door

Unlocking the enclave door is a single phase operation.

FD.Enclave.OverrideDoorLockOK
FS.Enclave.OverrideDoorLockOK

A guard may need to open the enclave door to admit someone who cannot be admitted by the system.

$$
\begin{array}{|l}
\textit{OverrideDoorLockOKC} \\
\hline
\textit{AdminOpStartedContextC} \\
\\
\Xi \textit{FloppyC} \\
\Xi \textit{ConfigC} \\
\textit{AdminFinishOpC} \\
\textit{UnlockDoorC} \\
\hline
\textit{the}\ \textit{currentAdminOpC} = \textit{overrideLock} \\
\\
\textit{currentScreenC}'.\textit{screenMsgC} = \textit{requestAdminOpC} \\
\textit{currentDisplayC}' = \textit{doorUnlocked} \\
\\
\textit{enclaveStatusC}' = \textit{enclaveQuiescent} \\
\\
\textit{auditTypes}\ \textit{newElements}? \cap \textit{ADMIN\_ELEMENTS} = \{\textit{overrideLockElement}\} \\
\\
\exists_1\ \textit{element} : \textit{AuditC} \bullet \textit{element} \in \textit{newElements}? \\
\quad \wedge \textit{element}.\textit{elementId} = \textit{overrideLockElement} \\
\quad \wedge \textit{element}.\textit{logTime} \in \textit{nowC} \mathrel{..} \textit{nowC}' \\
\quad \wedge \textit{element}.\textit{user} = \textit{extractUser}\ \textit{currentAdminTokenC} \\
\quad \wedge \textit{element}.\textit{severity} = \textit{information} \\
\quad \wedge \textit{element}.\textit{description} = \textit{noDescription} \\
\hline
\end{array}
$$

▷ See: *AdminOpStartedContextC* (p. 116), *FloppyC* (p. 36), *ConfigC* (p. 27), *AdminFinishOpC* (p. 77), *UnlockDoorC* (p. 64), *overrideLock* (p. 34), *doorUnlocked* (p. 22), *ADMIN_ELEMENTS* (p. 29), *overrideLockElement* (p. 28), *AuditC* (p. 30), *extractUser* (p. 30), *information* (p. 28), *noDescription* (p. 30)

FD.Enclave.TISUnlockDoorOp
FS.Enclave.TISUnlockDoorOp

This operation has no failures, other than the administrator tearing their token before the operation completes, the token tear is covered in Section 7.4.

$$\textit{TISOverrideDoorLockOpC} \mathrel{\widehat=} (\textit{OverrideDoorLockOKC}) \setminus (\textit{newElements}?)$$

▷ See: *OverrideDoorLockOKC* (p. 127)

8 THE INITIAL SYSTEM AND STARTUP

8.1 The Initial System

FD.TIS.InitIDStation
FS.TIS.InitIDStation

After initial installation the system has the following properties

- an empty key store, which means it is unable to authorise entry to anyone;
- default configuration data, which does not permit entry to anyone;
- the door latched;
- an empty audit log;
- the internal times all set to zero (a time before the current time).

The door is assumed closed at initialisation, this ensures that the alarm will not sound before the first time that data is polled.

$$
\begin{array}{|l}
\hline
\textit{InitDoorLatchAlarmC} \\
\textit{DoorLatchAlarmC} \\
\hline
\textit{currentTimeC} = \textit{zeroTime} \\
\textit{currentDoorC} = \textit{closed} \\
\textit{latchTimeoutC} = \textit{zeroTime} \\
\textit{alarmTimeoutC} = \textit{zeroTime} \\
\textit{doorAlarmC} = \textit{silent} \\
\textit{currentLatchC} = \textit{locked} \\
\hline
\end{array}
$$

▷ See: *DoorLatchAlarmC* (p. 35), *zeroTime* (p. 11), *closed* (p. 21), *silent* (p. 21), *locked* (p. 21)

There are no keys held by the system.

$$
\begin{array}{|l}
\hline
\textit{InitKeyStoreC} \\
\textit{KeyStoreC} \\
\hline
\textit{keys} = \varnothing \\
\hline
\end{array}
$$

▷ See: *KeyStoreC* (p. 33)

The initial certificate store has the 0 as the available serial number.

$$
\begin{array}{|l}
\hline
\textit{InitCertificateStore} \\
\textit{CertificateStore} \\
\hline
\textit{nextSerialNumber} = 0 \\
\hline
\end{array}
$$

▷ See: *CertificateStore* (p. 34)

This default configuration assumes the lowest classification possible for the enclave. This ensures that it does not give inadvertently high clearance to the authorisation certificate. The parameters that define the *authPeriod* and *entryPeriod* functions are set to enable entry into the enclave to re-configure the TIS. This configuration will allow Auth Certificates to be generated with a validity of 2 hours from the point of issue.

InitConfigC

ConfigC

$alarmSilentDurationC = 10$
$latchUnlockDurationC = 150$
$tokenRemovalDurationC = 100$
$fingerWaitDuration = 100$
$enclaveClearanceC = unmarked$
$minEntryClass = unmarked$

$maxAuthDuration = 72000$
$accessPolicy = allHours$
$systemMaxFar = 1000$

▷ See: *ConfigC* (p. 27), *unmarked* (p. 12), *allHours* (p. 27)

▷ The initial values of *workingHoursStart*, *workingHoursEnd* will not impact the entry or authorisation periods so are not defined here, they are free to be implemented with any value.

Initially no administrator is logged on and no administator operations are taking place.

InitAdminC

AdminC

$rolePresentC = nil$
$currentAdminOpC = nil$

▷ See: *AdminC* (p. 35)

Initially the statistics are set to zero, indicating no use of the system to date.

InitStatsC

StatsC

$successEntryC = 0$
$failEntryC = 0$
$successBioC = 0$
$failBioC = 0$

▷ See: *StatsC* (p. 34)

The initial audit Log is empty and there is no audit alarm.

InitAuditLogC

AuditLogC

$logFiles = LOGFILEINDEX \times \{\varnothing\}$
$auditAlarmC = silent$

▷ See: *AuditLogC* (p. 32), *LOGFILEINDEX* (p. 31), *silent* (p. 21)

Initially the internal state is *notEnrolled*.

```
┌ InitInternalC ─────────────────────────
│ InternalC
├──────────
│ enclaveStatusC = notEnrolled
│ statusC = quiescent
└─────────────────────────────────────────
```

▷ See: *InternalC* (p. 37), *notEnrolled* (p. 37), *quiescent* (p. 37)

▷ In the above states the timeouts *fingerTimeout* and *tokenRemovalTimeoutC* are not used so their values are not important. The implementation is free to set their initial value to any valid value.

Entities that model the real world and are polled and have no security implications are not set at initialisation, these will be updated at the first poll of the real world entities.

Initially the screen and the display are clear.

```
┌ InitIDStationC ────────────────────────
│ IDStationC
│
│ InitDoorLatchAlarmC
│ InitConfigC
│ InitKeyStoreC
│ InitStatsC
│ InitAuditLogC
│ InitAdminC
│ InitInternalC
│ InitCertificateStore
├──────────
│ currentScreenC.screenMsgC = clearC
│
│ currentDisplayC = blank
└─────────────────────────────────────────
```

▷ See: *IDStationC* (p. 38), *InitDoorLatchAlarmC* (p. 129), *InitConfigC* (p. 130), *InitKeyStoreC* (p. 129), *InitStatsC* (p. 130), *InitAuditLogC* (p. 130), *InitAdminC* (p. 130), *InitInternalC* (p. 131), *InitCertificateStore* (p. 129), *clearC* (p. 23), *blank* (p. 22)

8.2 Starting the ID Station

FD.TIS.TISStartup
FS.TIS.TISStartup

We assume that some of the state within TIS is persistent through shutdown and some is not. The persistent items are *ConfigC*, *KeyStoreC*, *CertificateStore* and *AuditLogC* all other state components are set at startup. Those values that are polled can take any valid value, we assume for simplicity that they remain unchanged.

$$
\begin{array}{|l}
\textit{StartContextC} \\
\hline
\Delta IDStationC \\
RealWorldChangesC \\[1ex]
\Xi ConfigC \\
\Xi KeyStoreC \\[1ex]
InitDoorLatchAlarmC' \\
InitStatsC' \\
InitAdminC' \\
AddElementsToLogC \\
LogChangeC \\[1ex]
\Xi UserTokenC \\
\Xi AdminTokenC \\
\Xi FingerC \\
\Xi FloppyC \\
\Xi KeyboardC \\
\hline
auditTypes\ newElements? \subseteq STARTUP\_ELEMENTS \cup USER\_INDEPENDENT\_ELEMENTS \\
\hline
\end{array}
$$

▷ See: *IDStationC* (p. 38), *RealWorldChangesC* (p. 40), *ConfigC* (p. 27), *KeyStoreC* (p. 33), *InitDoorLatchAlarmC* (p. 129), *InitStatsC* (p. 130), *InitAdminC* (p. 130), *AddElementsToLogC* (p. 51), *LogChangeC* (p. 61), *UserTokenC* (p. 36), *AdminTokenC* (p. 36), *FingerC* (p. 36), *FloppyC* (p. 36), *KeyboardC* (p. 36), *STARTUP_ELEMENTS* (p. 29), *USER_INDEPENDENT_ELEMENTS* (p. 29)

In the case that TIS does not have any private keys in the *KeyStoreC* the ID station is assumed to require enrolment.

$$
\begin{array}{|l}
\textit{StartNonEnrolledStationC} \\
\hline
StartContextC \\[1ex]
InitCertificateStore' \\
\hline
privateKey = nil \\[1ex]
currentScreenC'.screenMsgC = clearC \\[1ex]
currentDisplayC' = blank \\
enclaveStatusC' = notEnrolled \\
statusC' = quiescent \\[1ex]
auditTypes\ newElements? \cap STARTUP\_ELEMENTS = \{startUnenrolledTISElement\} \\[1ex]
\exists_1\ element : AuditC \bullet element \in newElements? \\
\quad \wedge element.elementId = startUnenrolledTISElement \\
\quad \wedge element.logTime \in nowC \mathrel{..} nowC' \\
\quad \wedge element.user = noUser \\
\quad \wedge element.severity = information \\
\quad \wedge element.description = noDescription \\
\hline
\end{array}
$$

▷ See: *StartContextC* (p. 131), *InitCertificateStore* (p. 129), *clearC* (p. 23), *blank* (p. 22), *notEnrolled* (p. 37), *quiescent* (p. 37), *STARTUP_ELEMENTS* (p. 29), *AuditC* (p. 30), *noUser* (p. 30), *information* (p. 28), *noDescription* (p. 30)

In the case that TIS does have a private key the ID station is assumed to have been previously enrolled.

StartEnrolledStationC

$StartContextC$

$\Xi CertificateStore$

$privateKey \neq nil$

$currentScreenC'.screenMsgC = welcomeAdminC$

$currentDisplayC' = welcome$
$enclaveStatusC' = enclaveQuiescent$
$statusC' = quiescent$

$auditTypes\ newElements? \cap STARTUP\_ELEMENTS = \{startEnrolledTISElement\}$

$\exists_1\ element : AuditC \bullet element \in newElements?$
$\quad \wedge element.elementId = startEnrolledTISElement$
$\quad \wedge element.logTime \in nowC \mathrel{..} nowC'$
$\quad \wedge element.user = noUser$
$\quad \wedge element.severity = information$
$\quad \wedge element.description = noDescription$

▷ See: *StartContextC* (p. 131), *CertificateStore* (p. 34), *welcomeAdminC* (p. 23), *welcome* (p. 22), *quiescent* (p. 37), *STARTUP_ELEMENTS* (p. 29), *startEnrolledTISElement* (p. 28), *AuditC* (p. 30), *noUser* (p. 30), *information* (p. 28), *noDescription* (p. 30)

The complete startup operation is given by:

$$TISStartupC \mathrel{\widehat=} (StartEnrolledStationC \vee StartNonEnrolledStationC) \setminus (newElements?)$$

▷ See: *StartEnrolledStationC* (p. 132), *StartNonEnrolledStationC* (p. 132)

9 THE WHOLE ID STATION

9.1 Startup

When the TIS is powered up it needs to establish whether it is enrolled or not. This is formally described by

TISStartUpC

9.2 The main loop

FD.TIS.TISMainLoop
FS.TIS.TISMainLoop

The TIS achieves its function by repeatedly performing a number of activities within a main loop.

The main loop is broken down into several phases:

- *Poll* - Polling reads the simple real world entities (door, time) and the reads the presence or absence of the complex entities (user token reader, admin token reader, fingerprint reader, floppy).
- *Early Updates* - Critical updates of the door latch and alarm are performed as soon as new polled data is available.
- *TIS processing* - TIS processing is the activity performed by TIS, this is influenced by the current *status* of TIS and the recently read inputs.
- *Updates* - Critical updates of the door latch and alarm are repeated once the processing is complete to ensure any internal state changes result in the latch and alarm being set correctly. Less critical updates of the screen and display are also performed once the processing is complete.

The the TIS processing depends on the current internal *status*.

Initially the only activity that can be performed is enrolment, formally captured as *TISEnrol*.

When it is in a quiescent state it can start a number of activities. These are started by either reading a user token, an adminstrator token or keyboard data. In addition an administrator may logoff.

If the conditions for performing activities are not satisfied then the system is idle.

FD.TIS.Idle
FS.Enclave.WaitingAdminTokenRemoval

TISIdleC

$\Xi IDStationC$

$\neg\ EnrolmentIsInProgress$
$\neg\ AdminMustLogout$
$\neg\ CurrentUserEntryActivityPossible$
$\neg\ UserEntryCanStart$
$\neg\ CurrentAdminActivityPossible$
$\neg\ AdminLogonCanStart$
$\neg\ AdminOpCanStart$

▷ See: *IDStationC* (p. 38), *EnrolmentIsInProgress* (p. 78), *AdminMustLogout* (p. 79), *CurrentUserEntryActivityPossible* (p. 79), *UserEntryCanStart* (p. 80), *CurrentAdminActivityPossible* (p. 80), *AdminLogonCanStart* (p. 81), *AdminOpCanStart* (p. 81)

If the administrator is logged on and conditions change such that the administrator should be logged off, either token removal or token expiry, then the short lived administrator logoff activity is performed, even during a user entry.

Once a user token has been presented to TIS the only activities that can be performed are stages in the multi-phase user entry authentication operation, formally captured as *TISProgressUserEntry*.

Once an administrator token has been presented to TIS the administrator is logged onto the ID Station, formally captured as *TISProgressAdminLogon*. Having logged the administrator on TIS returns to a *quiescent* state waiting for the administrator to perform an operation, without preventing user entry.

Once an operation request has been made by a logged on administrator TIS performs the, potentially multi-phase, administrator operation, formally captured as *TISAdminOpC* captured below:

$$\begin{aligned} &TISAdminOpC \mathrel{\widehat=} TISOverrideDoorLockOpC \lor TISShutdownOpC \\ &\qquad \lor TISUpdateConfigDataOpC \lor TISArchiveLogOpC \end{aligned}$$

▷ See: *TISOverrideDoorLockOpC* (p. 128), *TISShutdownOpC* (p. 127), *TISUpdateConfigDataOpC* (p. 125), *TISArchiveLogOpC* (p. 123)

The various possible activities with conditions that ensure the desired priority of handling are given below.

$$TISDoEnrolOp \mathrel{\widehat=} EnrolmentIsInProgress \land TISEnrolOpC$$

$$TISDoAdminLogout \mathrel{\widehat=} \lnot\ EnrolmentIsInProgress \land AdminMustLogout \land TISAdminLogoutC$$

$$\begin{aligned} &TISDoProgressUserEntry \mathrel{\widehat=} \lnot\ EnrolmentIsInProgress \land \lnot\ AdminMustLogout \\ &\qquad \land CurrentUserEntryActivityPossible \land TISProgressUserEntry \end{aligned}$$

$$\begin{aligned} &TISDoProgressAdminActivity \mathrel{\widehat=} \lnot\ EnrolmentIsInProgress \land \lnot\ AdminMustLogout \\ &\qquad \land \lnot\ CurrentUserEntryActivityPossible \land \\ &\qquad CurrentAdminActivityPossible \land (TISProgressAdminLogon \lor TISAdminOpC) \end{aligned}$$

$$\begin{aligned} &TISDoStartUserEntry \mathrel{\widehat=} \lnot\ EnrolmentIsInProgress \land \lnot\ AdminMustLogout \\ &\qquad \land \lnot\ CurrentUserEntryActivityPossible \land \lnot\ CurrentAdminActivityPossible \\ &\qquad \land UserEntryCanStart \land TISStartUserEntry \end{aligned}$$

$$\begin{aligned} &TISDoStartAdminActivity \mathrel{\widehat=} \lnot\ EnrolmentIsInProgress \land \lnot\ AdminMustLogout \\ &\qquad \land \lnot\ CurrentUserEntryActivityPossible \land \lnot\ CurrentAdminActivityPossible \\ &\qquad \land \lnot\ UserEntryCanStart \\ &\qquad \land (TISStartAdminLogonC \lor TISStartAdminOpC) \end{aligned}$$

▷ See: *EnrolmentIsInProgress* (p. 78), *TISEnrolOpC* (p. 107), *AdminMustLogout* (p. 79), *TISAdminLogoutC* (p. 115), *CurrentUserEntryActivityPossible* (p. 79), *TISProgressUserEntry* (p. 101), *CurrentAdminActivityPossible* (p. 80), *TISProgressAdminLogon* (p. 113), *TISAdminOpC* (p. 135), *UserEntryCanStart* (p. 80), *TISStartUserEntry* (p. 100), *TISStartAdminLogonC* (p. 113), *TISStartAdminOpC* (p. 119)

The TIS processing activity is described by the following:

$TISProcessingC \mathrel{\widehat=}$
$\quad TISDoEnrolOp$
$\quad \lor TISDoAdminLogout$
$\quad \lor TISDoProgressUserEntry$
$\quad \lor TISDoProgressAdminActivity$
$\quad \lor TISDoStartUserEntry$
$\quad \lor TISDoStartAdminActivity$
$\quad \lor TISIdleC$

▷ See: *TISDoEnrolOp* (p. 135), *TISDoAdminLogout* (p. 135), *TISDoProgressUserEntry* (p. 135), *TISDoProgressAdminActivity* (p. 135), *TISDoStartUserEntry* (p. 135), *TISDoStartAdminActivity* (p. 135), *TISIdleC* (p. 134)

A APPENDIX: COMMENTARY ON THIS DESIGN

This design is intended to give a representative formal refinement of the Formal Specification [4].

A.1 The structure of the Z

The formal design follows the structure of the Formal Specification. This is done to aid the refinement process and provide a natural refinement step from specification to implementation.

As in the specification every effort has been taken to ensure schemas are simple.

The section containing internal operations and checks has been expanded. A number of common constraints have been factored out as checks that can be performed in the context of one or a small number of subsystems. In order to simplify the step from design to implementation invariants which define key values, such as the door alarm, have been replaced by subsystem operations. The design then shows where these operations need to be invoked to ensure that the desired invariants are maintained.

A.2 Issues

A few issues arose while writing this design, some of which point to shortfalls which would need to be resolved if EAL level 6 or 7 were required.

We present the more interesting observations here:

A.2.1 Peripheral Failures and System Faults

The design does not address fully the possibility of peripheral failures. This would certainly need to be addressed for EAL 6 or 7 where fully formal proof of the implementation conforming to the design is required.

We do model the possibility of a system fault being raised and this is intended to cover peripheral failures; however, we do not elaborate what should occur in the event of such a failure. It is likely that peripheral failures would, in a full development be categorised in terms of their criticality and the desired system behaviour as part of the requirements elicitation activity.

There are a number of points where the modelling of failures could be improved. The manner in which these could be improved is discussed below.

The model makes the assumption that any attempt to read a token is successful in that the internal representation exactly reflects the real world contents of the token. In order to model the possibility of failure during the read the model should allow the non-deterministic possibility of the internal value of the token becoming *badT* representing a corrupt or failed read. This non-determinism would also need to be present in the specification to ensure that the design is a refinement of the specification.

A small number of system faults are deemed security critical. These are likely to include

- failure to be able to write to the audit log;
- any detectable failure in operating the latch
- any detectable failure to be able to monitor the state of the door.

These failures will occur non-deterministically and we should specify the desired behaviour if each of these occur.

A failure to write to the audit log is severe since it means that activities could proceed un-audited. In the event of a failure to write to the audit log the audit alarm should be raised and the system should be shutdown preventing it from participating an any further activities.

In the event of a failure we can assume little about the state of the current log file, we assume that nothing old was lost but some elements may have been added.

$$
\begin{array}{l}
\textbf{AuditLogFailure} \\
\hline
\Delta AuditLogC \\
\hline
auditAlarmC' = alarming \\
logFilesStatus' = logFilesStatus \\
currentLogFile' = currentLogFile \\
usedLogFiles' = usedLogFiles \\
freeLogFiles' = freeLogFiles \\
\{currentLogFile\} \ntriangleleft logFiles' = \{currentLogFile\} \ntriangleleft logFiles \\
logFiles\ currentLogFile \subseteq logFiles'\ currentLogFile \\
\hline
\end{array}
$$

▷ See: *AuditLogC* (p. 32), *alarming* (p. 21)

In the event of such a failure the administrator should be logged off and the system shutdown. The door should be locked to ensure the enclave is left in a secure state.

$$
\begin{array}{l}
\textbf{ShutdownAuditFailure} \\
\hline
\Delta IDStationC \\
RealWorldChangesC \\
\\
LockDoorC \\
\Xi KeyStoreC \\
\Xi CertificateStore \\
\Xi ConfigC \\
\Xi FloppyC \\
\Xi KeyboardC \\
\Xi AdminTokenC \\
\Xi UserTokenC \\
AdminLogoutC \\
\Xi FingerC \\
\Xi StatsC \\
AuditLogFailure \\
\hline
enclaveStatusC' = shutdown \\
statusC' = quiescent \\
\\
currentDisplayC' = blank \\
currentScreenC'.screenMsgC = clearC \\
\hline
\end{array}
$$

▷ See: *IDStationC* (p. 38), *RealWorldChangesC* (p. 40), *LockDoorC* (p. 65), *KeyStoreC* (p. 33), *CertificateStore* (p. 34), *ConfigC* (p. 27), *FloppyC* (p. 36), *KeyboardC* (p. 36), *AdminTokenC* (p. 36), *UserTokenC* (p. 36), *AdminLogoutC* (p. 76), *FingerC* (p. 36), *StatsC* (p. 34), *AuditLogFailure* (p. 138), *quiescent* (p. 37), *blank* (p. 22), *clearC* (p. 23)

It is likely that the desired behaviour in the event of a failure of the door or latch is to assume that the system is in an insecure state and raise an alarm. It may also be desirable to shutdown the system, preventing any further action.

$$
\begin{array}{l}
\textit{DoorLatchFailure} \\
\hline
\Delta DoorLatchAlarmC \\
\hline
doorAlarmC' = alarming \\
currentTimeC' = currentTimeC \\
currentDoorC' = open \\
currentLatchC' = locked \\
latchTimeoutC' = zeroTime \\
alarmTimeoutC' = zeroTime
\end{array}
$$

▷ See: *DoorLatchAlarmC* (p. 35), *alarming* (p. 21), *open* (p. 21), *locked* (p. 21), *zeroTime* (p. 11)

In the event of such a failure, the fault can be logged and the system shutdown.

$$
\begin{array}{l}
\textit{ShutdownDoorLatchFailure} \\
\hline
\Delta IDStationC \\
RealWorldChangesC \\
\\
DoorLatchFailure \\
\Xi KeyStoreC \\
\Xi CertificateStore \\
\Xi ConfigC \\
\Xi FloppyC \\
\Xi KeyboardC \\
\Xi AdminTokenC \\
\Xi UserTokenC \\
AdminLogoutC \\
\Xi FingerC \\
\Xi StatsC \\
AddElementsToLogC \\
LogChangeC \\
\hline
enclaveStatusC' = shutdown \\
statusC' = quiescent \\
\\
currentDisplayC' = blank \\
currentScreenC'.screenMsgC = clearC \\
\\
\exists_1\ element : AuditC \bullet element \in newElements? \\
\quad \land element.elementId = systemFaultElement \\
\quad \land element.logTime \in nowC \mathinner{\ldotp\ldotp} nowC' \\
\quad \land element.user = noUser \\
\quad \land element.severity = critical
\end{array}
$$

▷ See: *IDStationC* (p. 38), *RealWorldChangesC* (p. 40), *DoorLatchFailure* (p. 138), *KeyStoreC* (p. 33), *CertificateStore* (p. 34), *ConfigC* (p. 27), *FloppyC* (p. 36), *KeyboardC* (p. 36), *AdminTokenC* (p. 36), *UserTokenC* (p. 36), *AdminLogoutC* (p. 76), *FingerC* (p. 36), *StatsC* (p. 34), *AddElementsToLogC* (p. 51), *LogChangeC* (p. 61), *quiescent* (p. 37), *blank* (p. 22), *clearC* (p. 23), *AuditC* (p. 30), *systemFaultElement* (p. 28), *noUser* (p. 30), *critical* (p. 28)

As faults are not modelled in the specification refinement would not be achievable if system faults were modelled in the design.

A.2.2 Unelaborated aspects of the Design

Normally all types within the design would be elaborated in terms of entities that closely model the implementation type.

There are some aspects of certificates that have not been fully elaborated within this design. These are *FINGERPRINT*, and *FINGERPRINTTEMPLATE*. All of these would normally be elaborated in terms of a model of the implementation types. This is unnecessary for these three entities. The core TIS has no reason to use the *FINGERPRINT* or *FINGERPRINTTEMPLATE*, it simply passes the information to the Biometric library.

The components of an issuer *USERID* and *USERNAME* are free types within the design. The only property that is utilised within the design is equality of *USERID*. For this demonstration implementation the user Id is simplified to a numeric although this is not completely realistic so is not elaborated within the design.

A.2.3 Enrolment Protocol

Enrolment is a simplified model of part of the enrolment protocol. The likely enrolment protocol would involve the following stages.

1. TIS generates a pubic/private key pair at initialisation and uses the public key to create a request for enrolment.
2. The enrolment request is presented to a CA. The CA would generate an Id certificate for the TIS, this will contain the authorised name of the TIS as its subject and the TIS public key.
3. An AA constructs the enrolment data. Enrolment data comprises a number of Id certificates, including the Id certificate of the TIS itself and the Id certificate of the CA that issued the TIS Id certificate.
4. TIS accepts the enrolment data and uses this to establish known issuers.

Within the design we only model the final phase of enrolment.

TIS would only actually participate in the first and last phase of this protocol, the other two activities being performed by a CA and AA.

Due to budgetary limitations we have omitted the first phase of this protocol from the design model. This is possible since this demonstration mimics the keys and the encryption process. There is no need for our demonstration to be supplied with the public key that corresponds to an internally held private key as the private key is not used in the mimicked encryption. Instead TIS will record the presence of the private key once enrolment has supplied its ID Certificate.

A.2.4 Reading Tokens

The formal design shows all certificates on a token being read when anything from the token is required. In actuality the authorisation certificate will only ever be read from the administrator token, while the reading of certificates from the user token will follow the following ordering.

- An attempt is made to read the authorisation certificate and ID certificate.
- If these are present then they are validated.
- If they fail to validate or are not present then the remaining certificates are read.

Due to budgetary limitations this was not progressed within this design although it would be necessary to achieve EAL 6 or above to enable formal proof of the implementation satisfying the design.

The design as it stands is not invalid, it just presents a slightly larger step between design and implementation than might be desirable.

A.2.5 Token Representation

Within the formal design we represent tokens as containing a number of raw certificates. This is an effective model for the real world view of the tokens but it is a less satisfactory model for the internal representation of the token.

Given more resources we would have modelled the internal tokens as containing the contents of the certificates that we are interested in. So for the administrators token only the contents of the Authorisation Certificate would be preserved, while for the user token the contents of the all the certificates may be maintained.

This would have the advantage within the design of removing the need to extract the required fields from the various tokens every time they are required.

If this design were to be progressed further it would be worth modelling the internal representation of tokens as maintaining the contents of selected certificates rather than the raw certificates. This would then result in a smaller step to implementation.

A.2.6 Relating enclave entry and Auth Cert generation

Within the specification independent configuration is used to determine the authorisation period applied to authorisation certificates and the times at which entry to the enclave should be allowed.

It should be noted that if the user obtains an authorisation certificate and the system is reconfigured before the authorisation certificate expires then it is still possible for the user to possess a current authorisation certificate and be denied entry.

A.2.7 Comments on the refinement relation

There are a number of circumstances where not all the abstract entities in the specification can be retrieved from the concrete entities in this design. In particular this applies to certificates on tokens and the configuration data. In the case of certificates this is due to the validity period used in the specification not necessarily being contiguous. In the case of the configuration data this is due to the enormous freedom in the definition of the abstract *authPeriod* and *entryPeriod* functions.

- The concrete *authPeriod* and *entryPeriod* are independent of *role*.
- There is no relationship between the *authPeriod* and *entryPeriod*.
- The concrete *authPeriod* is always a contiguous range of times.

As the specification has a very abstract view of what the real world can do, this is acceptable.

During TIS operations the real world undergoes change which is relatively unconstrained, in both the concrete and abstract model time must not decrease but all other real world entities that are not controlled by TIS may change arbitrarily. In the specification there are more possible states in which the real world can change into, the abstract tokens can change to ones that cannot be represented in the concrete model, floppy data can change to contain configuration data that is not valid configuration data in the concrete model. In all these cases the concrete “bad” value is a refinement of the abstract values that are not attainable in the concrete real world.

This refinement is acceptable as long as our concrete real world still allows all values that the requirements consider should be valid inputs.

In a real development of a working product there would be a part of requirements elictation in which the exact nature of all the inputs is discussed. This discussion may well be postponed until the Formal Specification is in place providing a useful context for discussion, this would very much depend on the nature of the inputs and whether the product development can control the allowable range of values.

Our design has constrained the validity periods on certificates to reflect the contiguous ranges that can be specified reflecting true requirements constraints on the nature of X509 certificates.

The new constraints on the configuration data have been introduced to limit allowed configurations to those that can be specified with a small number of parameters. In a real development these are design constraints and would need to be discussed with the customer to ensure that sufficient flexibility remains in the allowed configuration values.

B APPENDIX: THE ABSTRACTION RELATION

This chapter defines the retreival relation between the concrete state presented in this document and the abstract state in the formal specification [4]. The reader is referred to the formal specification for definitions of the schemas within that specification.

B.1 Fingerprint

FD.FingerprintTemplate.Retrieval	
FS.Types.FingerprintTemplate	*FD.Types.FingerprintTemplate*

$$
\begin{array}{l}
\textit{FingerprintTemplateR} \\
\hline
\textit{FingerprintTemplate} \\
\textit{FingerprintTemplateC} \\
\hline
\textit{template} = \textit{templateC}
\end{array}
$$

▷ See: *FingerprintTemplateC* (p. 14)

▷ The abstract model did not consider the *far* so this is free.

This relation can be used to define an abstraction function:

$$
\begin{array}{l}
\textit{fingerprintTemplateR} : \textit{FingerprintTemplateC} \longrightarrow \textit{FingerprintTemplate} \\
\hline
\textit{fingerprintTemplateR} = \{\textit{FingerprintTemplateC}; \textit{FingerprintTemplate} \mid \textit{FingerprintTemplateR} \bullet \\
\qquad \theta \textit{FingerprintTemplateC} \mapsto \theta \textit{FingerprintTemplate}\}
\end{array}
$$

▷ See: *FingerprintTemplateC* (p. 14), *FingerprintTemplateR* (p. 143)

B.2 Certificates

FD.Certificates.Retrieval	
FS.Types.Certificates	*FD.Types.Certificates*

We state that there is a bijection between the concrete *User* type and the abstract type *USER*. The abstract type *USER* was a basic type with no constraints on its structure or contents, the concrete *Issuer* is implemented as two fields, a name and an Id.

$$
\begin{array}{l}
\textit{userR} : \textit{User} \rightarrowtail\!\!\!\!\twoheadrightarrow \textit{USER} \\
\hline
\textit{userR}(\!|\textit{Issuer}|\!) = \textit{ISSUER}
\end{array}
$$

▷ See: *User* (p. 13), *Issuer* (p. 13)

There is a simple retrieval relation for certificate Ids.

$$
\begin{array}{l}
\textit{CertificateIdR} \\
\hline
\textit{CertificateId} \\
\textit{CertificateIdC} \\
\hline
\textit{issuer} = \textit{userR}\ \textit{issuerC}
\end{array}
$$

▷ See: *CertificateIdC* (p. 16), *userR* (p. 143)

▷ The abstract model did not consider the *serialNumber* so this is free.

This relation can be used to define an abstraction function:

$$certificateIdR : CertificateIdC \longrightarrow CertificateId$$
$$certificateIdR = \{CertificateIdC; CertificateId \mid CertificateIdR \bullet \theta CertificateIdC \mapsto \theta CertificateId\}$$

▷ See: *CertificateIdC* (p. 16), *CertificateIdR* (p. 143)

The abstract Certificates can be retrieved from the concrete (Raw) certificates, by making use of the appropriate extraction functions.

IDCertR

IDCert
IDCertC

$$\begin{aligned}
&\exists IDCertContents \bullet \theta IDCertContents = extractIDCert\, \theta RawCertificate\\
&\quad \land id = certificateIdR\ idC\\
&\quad \land validityPeriod = notBefore \mathinner{..} notAfter\\
&\quad \land isValidatedBy = \{key : KEYPART \mid\\
&\quad\quad (mechanism, digest\ mechanism\ data, signature)\ \mathsf{isVerifiedBy}\ key \bullet key\}\\
&\quad \land subject = userR\ subjectC\\
&\quad \land subjectPubK = subjectPubKC
\end{aligned}$$

▷ See: *IDCertC* (p. 18), *IDCertContents* (p. 16), *extractIDCert* (p. 17), *RawCertificate* (p. 15), *certificateIdR* (p. 144), *digest* (p. 15), *userR* (p. 143)

▷ We make the assumption here that there is no more than one possible key that will validate the data.

The same retrieval relation works for ID certificates of CAs.

$$CAIdCertR \mathrel{\widehat{=}} CAIdCert \land CAIdCertC \land IDCertR$$

▷ See: *CAIdCertC* (p. 18), *IDCertR* (p. 144)

PrivCertR

PrivCert
PrivCertC

$$\begin{aligned}
&\exists PrivCertContents \bullet \theta PrivCertContents = extractPrivCert\, \theta RawCertificate\\
&\quad \land id = certificateIdR\ idC\\
&\quad \land validityPeriod = notBefore \mathinner{..} notAfter\\
&\quad \land isValidatedBy = \{key : KEYPART \mid\\
&\quad\quad (mechanism, digest\ mechanism\ data, signature)\ \mathsf{isVerifiedBy}\ key \bullet key\}\\
&\quad \land baseCertId = certificateIdR\ baseCertIdC\\
&\quad \land tokenID = tokenIDR\ baseCertIdC.serialNumber\\
&\quad \land role = roleC\\
&\quad \land clearance = clearanceC
\end{aligned}$$

▷ See: *PrivCertC* (p. 18), *PrivCertContents* (p. 17), *RawCertificate* (p. 15), *certificateIdR* (p. 144), *digest* (p. 15)

▷ We make the assumption here that there is no more than one possible key that will validate the data.

$$
\begin{array}{l}
\textbf{schema } AuthCertR \\
\quad AuthCert \\
\quad AuthCertC \\
\hline
\quad \exists AuthCertContents \bullet \theta AuthCertContents = extractAuthCert\ \theta RawCertificate \\
\qquad \land id = certificateIdR\ idC \\
\qquad \land validityPeriod = notBefore \mathrel{..} notAfter \\
\qquad \land isValidatedBy = \{key : KEYPART \mid \\
\qquad\qquad (mechanism, digest\ mechanism\ data, signature)\ \mathsf{isVerifiedBy}\ key \bullet key\} \\
\qquad \land baseCertId = certificateIdR\ baseCertIdC \\
\qquad \land tokenID = tokenIDR\ baseCertIdC.serialNumber \\
\qquad \land role = roleC \\
\qquad \land clearance = clearanceC
\end{array}
$$

▷ See: *AuthCertC* (p. 18), *AuthCertContents* (p. 17), *RawCertificate* (p. 15), *certificateIdR* (p. 144), *digest* (p. 15)

▷ We make the assumption here that there is no more than one possible key that will validate the data.

$$
\begin{array}{l}
\textbf{schema } IandACertR \\
\quad IandACert \\
\quad IandACertC \\
\hline
\quad \exists IandACertContents \bullet \theta IandACertContents = extractIandACert\ \theta RawCertificate \\
\qquad \land id = certificateIdR\ idC \\
\qquad \land validityPeriod = notBefore \mathrel{..} notAfter \\
\qquad \land isValidatedBy = \{key : KEYPART \mid \\
\qquad\qquad (mechanism, digest\ mechanism\ data, signature)\ \mathsf{isVerifiedBy}\ key \bullet key\} \\
\qquad \land baseCertId = certificateIdR\ baseCertIdC \\
\qquad \land tokenID = tokenIDR\ baseCertIdC.serialNumber \\
\qquad \land template = fingerprintTemplateR\ templateC
\end{array}
$$

▷ See: *IandACertC* (p. 18), *IandACertContents* (p. 17), *RawCertificate* (p. 15), *certificateIdR* (p. 144), *digest* (p. 15), *fingerprintTemplateR* (p. 143)

▷ We make the assumption here that there is no more than one possible key that will validate the data.

These relations can be used to define abstraction functions for obtaining abstract certificates from concrete certificates. These functions are not surjections since the abstract validity periods may not be contiguous but the concrete validity periods are always contiguous.

$$
\begin{array}{l}
\quad idCertR : IDCertC \longrightarrow IDCert \\
\quad privCertR : PrivCertC \longrightarrow PrivCert \\
\quad authCertR : AuthCertC \longrightarrow AuthCert \\
\quad iandACertR : IandACertC \longrightarrow IandACert \\
\hline
\quad idCertR = \{IDCertC; IDCert \mid IDCertR \bullet \theta IDCertC \mapsto \theta IDCert\} \\
\quad privCertR = \{PrivCertC; PrivCert \mid PrivCertR \bullet \theta PrivCertC \mapsto \theta PrivCert\} \\
\quad authCertR = \{AuthCertC; AuthCert \mid AuthCertR \bullet \theta AuthCertC \mapsto \theta AuthCert\} \\
\quad iandACertR = \{IandACertC; IandACert \mid IandACertR \bullet \theta IandACertC \mapsto \theta IandACert\}
\end{array}
$$

▷ See: *IDCertC* (p. 18), *PrivCertC* (p. 18), *AuthCertC* (p. 18), *IandACertC* (p. 18), *IDCertR* (p. 144), *PrivCertR* (p. 144), *AuthCertR* (p. 145), *IandACertR* (p. 145)

B.3 Tokens

FD.Tokens.Retrieval	
FS.Types.Tokens	*FD.Types.Tokens*

We state that there is a bijection between the concrete *TOKENIDC* type and the abstract type *TOKENID*. The abstract type *TOKENID* was a basic type with no constraints on its structure or contents, the concrete *TOKENIDC* is implemented as a natural number.

$$tokenIDR : TOKENIDC \rightarrowtail\!\!\!\!\twoheadrightarrow TOKENID$$

▷ See: *TOKENIDC* (p. 18)

The retrieval relation makes use of the retrieval relations for each of the certificate types.

We cannot define a retrieval relation for *Tokens* that is true for all concrete tokens. This is because the abstract tokens do not themselves have the possibility of a token containing the wrong type of certificate data. However we can define a retrieval relation for tokens where certificate contents can all be extracted from the concrete raw certificates.

TokenR

$$\begin{array}{l}
Token \\
TokenC \\
\hline
idCertC \in \{IDCertC\} \\
privCertC \in \{PrivCertC\} \\
iandACertC \in \{IandACertC\} \\
authCertC = nil \lor the\ authCertC \in \{AuthCertC\} \\
\\
tokenID = tokenIDR\ tokenIDC \\
\\
idCert = idCertR\ idCertC \\
privCert = privCertR\ privCertC \\
iandACert = iandACertR\ iandACertC \\
\\
authCert = nil \land authCertC = nil \\
\quad \lor \\
authCert \neq nil \land authCertC \neq nil \land the\ authCert = authCertR\ (the\ authCertC)
\end{array}$$

▷ See: *TokenC* (p. 19), *IDCertC* (p. 18), *PrivCertC* (p. 18), *IandACertC* (p. 18), *AuthCertC* (p. 18), *tokenIDR* (p. 146), *idCertR* (p. 145)

This relation holds for all *ValidToken*s.

$$ValidTokenR \mathrel{\widehat=} ValidToken \land ValidTokenC \land TokenR$$

▷ See: *ValidTokenC* (p. 19), *TokenR* (p. 146)

This relation can be used to define a partial abstraction function.

$$\begin{array}{l}
tokenR : TokenC \mathrel{\ooalign{$\rightarrow$\cr\hfil$\mapstochar$\hfil}} Token \\
\hline
tokenR = \{TokenC; Token \mid TokenR \bullet \theta TokenC \mapsto \theta Token\}
\end{array}$$

▷ See: *TokenC* (p. 19), *TokenR* (p. 146)

The retrieval relation for current tokens uses the retrieval relation for valid tokens and preserves *now*.

$$
\begin{array}{|l}
\textit{CurrentTokenR} \\
\hline
\textit{CurrentToken} \\
\textit{CurrentTokenC} \\
\hline
\textit{ValidTokenR} \\
\textit{now} = \textit{nowC}
\end{array}
$$

▷ See: *CurrentTokenC* (p. 19), *ValidTokenR* (p. 146)

B.4 Enrolment

FD.Enrolment.Retrieval	
FS.Types.Enrolment	*FD.Types.Enrolment*

$$
\begin{array}{|l}
\textit{EnrolR} \\
\hline
\textit{Enrol} \\
\textit{EnrolC} \\
\hline
\textit{idStationCert} = \textit{idCertR}\ \textit{idStationCertC} \\
\#\textit{issuerCerts} = \#\textit{issuerCertsC} \\
\forall\, \textit{certC} : \operatorname{ran} \textit{issuerCertsC} \bullet \exists\, \textit{cert} : \textit{issuerCerts} \bullet \textit{cert} = \textit{idCertR}\ \textit{certC} \\
\forall\, \textit{cert} : \textit{issuerCerts} \bullet \exists\, \textit{certC} : \operatorname{ran} \textit{issuerCertsC} \bullet \textit{cert} = \textit{idCertR}\ \textit{certC}
\end{array}
$$

▷ See: *EnrolC* (p. 20), *idCertR* (p. 145)

This relation can be used to define an abstraction function.

$$
\begin{array}{|l}
\textit{enrolR} : \textit{EnrolC} \longrightarrow \textit{Enrol} \\
\hline
\textit{enrolR} = \{\textit{EnrolC}; \textit{Enrol} \mid \textit{EnrolR} \bullet \theta \textit{EnrolC} \mapsto \theta \textit{Enrol}\}
\end{array}
$$

▷ See: *EnrolC* (p. 20), *EnrolR* (p. 147)

The same retrieval relation works for a valid enrolment.

$$\textit{ValidEnrolR} \mathrel{\widehat=} \textit{ValidEnrolC} \land \textit{ValidEnrol} \land \textit{EnrolR}$$

▷ See: *ValidEnrolC* (p. 21), *EnrolR* (p. 147)

B.5 Configuration Data

FD.ConfigData.Retrieval	
FS.ConfigData.State	*FD.ConfigData.State*

$$\begin{array}{l}
\textit{ConfigR} \\
\quad \textit{Config} \\
\quad \textit{ConfigC} \\
\hline
\quad \textit{alarmSilentDuration} = \textit{alarmSilentDurationC} \\
\quad \textit{latchUnlockDuration} = \textit{latchUnlockDurationC} \\
\quad \textit{tokenRemovalDuration} = \textit{tokenRemovalDurationC} \\
\quad \textit{enclaveClearance}.\textit{class} = \textit{enclaveClearanceC} \\
\quad \textit{authPeriod} = \{p : \textit{PRIVILEGE} \bullet p \mapsto \textit{authPeriodC}\} \\
\quad \textit{entryPeriod} = \{p : \textit{PRIVILEGE} \bullet p \mapsto \textit{entryPeriodC}\} \\
\quad \textit{minPreservedLogSize} = \textit{minPreservedLogSizeC} \\
\quad \textit{alarmThresholdSize} = \textit{alarmThresholdSizeC}
\end{array}$$

▷ See: *ConfigC* (p. 27), *PRIVILEGE* (p. 12)

This relation is not surjective, it cannot retrieve an *authPeriod* that depends on the *role* for instance.

We can define a function that retreives the abstract configuration data from the concrete:

$$\begin{array}{l}
\textit{configR} : \textit{ConfigC} \longrightarrow \textit{Config} \\
\hline
\textit{configR} = \{\textit{ConfigC}; \textit{Config} \mid \textit{ConfigR} \bullet \theta \textit{ConfigC} \mapsto \theta \textit{Config}\}
\end{array}$$

▷ See: *ConfigC* (p. 27), *ConfigR* (p. 147)

B.6 Real World

FD.RealWorld.Retrieval	
FD.Types.RealWorld	*FS.Types.RealWorld*

We define a retrieval relation mapping entities of type *TOKENTRYC* to their abstract representation. Note that all abstract tokens that cannot be retrieved from concrete tokens are related to concrete bad tokens.

$$\begin{array}{l}
\textit{tokenTryR} : \textit{TOKENTRYC} \longleftrightarrow \textit{TOKENTRY} \\
\hline
\textit{tokenTryR} = \{\textit{noTC} \mapsto \textit{noT}, \textit{badTC} \mapsto \textit{badT}\} \\
\qquad \cup \{\textit{TokenC} \mid \theta \textit{TokenC} \notin \operatorname{dom} \textit{tokenR} \bullet \textit{goodTC}\, \theta \textit{TokenC} \mapsto \textit{badT}\} \\
\qquad \cup \{\textit{TokenC} \mid \theta \textit{TokenC} \in \operatorname{dom} \textit{tokenR} \bullet \textit{goodTC}\, \theta \textit{TokenC} \mapsto \textit{goodT}\,(\textit{tokenR}\, \theta \textit{TokenC})\} \\
\qquad \cup \{\textit{Token} \mid \theta \textit{Token} \in \operatorname{ran} \textit{tokenR} \bullet \textit{badTC} \mapsto \textit{goodT}\, \theta \textit{Token}\}
\end{array}$$

▷ See: *TOKENTRYC* (p. 22), *noTC* (p. 22), *badTC* (p. 22), *TokenC* (p. 19), *tokenR* (p. 146), *goodTC* (p. 22)

▷ Concrete tokens that contain raw certificates from which the correct contents cannot be extracted are modelled as *badT* within the abstract model.

We define a retrieval relation mapping entities of type *FLOPPYC* to their abstract representation.

$$\begin{array}{l}
\textit{floppyR} : \textit{FLOPPYC} \longleftrightarrow \textit{FLOPPY} \\
\hline
\textit{floppyR} = \{\textit{noFloppyC} \mapsto \textit{noFloppy}, \textit{emptyFloppyC} \mapsto \textit{emptyFloppy}, \textit{badFloppyC} \mapsto \textit{badFloppy}\} \\
\qquad \cup \{\textit{ValidEnrolC} \bullet \textit{enrolmentFileC}\, \theta \textit{ValidEnrolC} \mapsto \textit{enrolmentFile}\,(\textit{enrolR}\, \theta \textit{ValidEnrolC})\} \\
\qquad \cup \{\textit{ValidEnrol} \mid \theta \textit{ValidEnrol} \notin \operatorname{ran} \textit{enrolR} \bullet \textit{badFloppyC} \mapsto \textit{enrolmentFile}\, \theta \textit{ValidEnrol}\} \\
\qquad \cup \{\textit{auditData} : \mathbb{F}\, \textit{AuditC} \bullet \textit{auditFileC}\, \textit{auditData} \mapsto \textit{auditFile}\,(\textit{auditR}(\!|\textit{auditData}|\!))\} \\
\qquad \cup \{\textit{ConfigC} \bullet \textit{configFileC}\, \theta \textit{ConfigC} \mapsto \textit{configFile}\,(\textit{configR}\, \theta \textit{ConfigC})\} \\
\qquad \cup \{\textit{Config} \mid \theta \textit{Config} \notin \operatorname{ran} \textit{configR} \bullet \textit{badFloppyC} \mapsto \textit{configFile}\, \theta \textit{Config}\}
\end{array}$$

▷ See: *FLOPPYC* (p. 22), *noFloppyC* (p. 22), *emptyFloppyC* (p. 22), *badFloppyC* (p. 22), *ValidEnrolC* (p. 21), *enrolmentFileC* (p. 22), *enrolR* (p. 147), *AuditC* (p. 30), *ConfigC* (p. 27), *configFileC* (p. 22), *configR* (p. 148)

We define a partial retrieval relation mapping entities of type *SCREENTEXTC* to their abstract representation.

$$screenTextR : SCREENTEXTC \leftrightarrow SCREENTEXT$$

$$\begin{array}{l}
screenTextR = \{clearC \mapsto clear, welcomeAdminC \mapsto welcomeAdmin, busyC \mapsto busy, \\
\quad removeAdminTokenC \mapsto removeAdminToken, closeDoorC \mapsto closeDoor, \\
\quad requestAdminOpC \mapsto requestAdminOp, doingOpC \mapsto doingOp, \\
\quad invalidRequestC \mapsto invalidRequest, invalidDataC \mapsto invalidData, \\
\quad insertEnrolmentDataC \mapsto insertEnrolmentData, validatingEnrolmentDataC \mapsto validatingEnrolmentData, \\
\quad enrolmentFailedC \mapsto enrolmentFailed, insertBlankFloppyC \mapsto insertBlankFloppy, \\
\quad insertConfigDataC \mapsto insertConfigData\} \\
\quad \cup \{StatsC \bullet displayStatsC\ \theta StatsC \mapsto displayStats\ (statsR\ \theta StatsC)\} \\
\quad \cup \{ConfigC; Config \mid ConfigR \bullet displayConfigDataC\ \theta ConfigC \mapsto displayConfigData\ \theta Config\}
\end{array}$$

▷ See: *SCREENTEXTC* (p. 23), *clearC* (p. 23), *welcomeAdminC* (p. 23), *busyC* (p. 23), *removeAdminTokenC* (p. 23), *closeDoorC* (p. 23), *doingOpC* (p. 23), *invalidRequestC* (p. 23), *invalidDataC* (p. 23), *validatingEnrolmentDataC* (p. 23), *enrolmentFailedC* (p. 23), *insertBlankFloppyC* (p. 23), *insertConfigDataC* (p. 23), *StatsC* (p. 34), *displayStatsC* (p. 23), *ConfigC* (p. 27), *ConfigR* (p. 147), *displayConfigDataC* (p. 23)

▷ The elements of *SCREENTEXTC* not in the domain are only used in the definition of screen state components that have no equivalent in the abstract model. Hence this function being partial will not affect our ability to define retrieval relations for the TIS state.

B.6.1 The Real World State

The retrieval relations for the controlled and monitored real world are simple.

TISControlledRealWorldR

$$\begin{array}{l}
TISControlledRealWorld \\
TISControlledRealWorldC \\
\hline
latch = latchC \\
alarm = alarmC \\
display = displayC \\
screen = screenR\ screenC
\end{array}$$

▷ See: *TISControlledRealWorldC* (p. 24)

TISMonitoredRealWorldR

$$\begin{array}{l}
TISMonitoredRealWorld \\
TISMonitoredRealWorldC \\
\hline
now = nowC \\
door = doorC \\
finger = fingerC \\
userTokenC \mapsto userToken \in tokenTryR \\
adminTokenC \mapsto adminToken \in tokenTryR \\
floppyC \mapsto floppy \in floppyR \\
keyboard = keyboardC
\end{array}$$

▷ See: *TISMonitoredRealWorldC* (p. 24), *tokenTryR* (p. 148), *floppyR* (p. 148)

Combining these relations we obtain the relation for the whole real world.

$$TISRealWorldR \mathrel{\widehat=} TISControlledRealWorldR \land TISMonitoredRealWorldR$$

▷ See: *TISControlledRealWorldR* (p. 149), *TISMonitoredRealWorldR* (p. 149)

B.7 Audit Log

FD.AuditLog.Retrieval	
FD.AuditLog.State	*FS.AuditLog.State*

We state that there is a bijection between the concrete *AuditC* type and the abstract type *Audit*. The abstract type *Audit* was a basic type with no constraints on its structure or contents.

$$auditR : AuditC \rightarrowtail\!\!\!\twoheadrightarrow Audit$$

▷ See: *AuditC* (p. 30)

We observe that within the implementation all log elements have the same size so the implementations of the functions *sizeElement* and *sizeLog* are given by:

$$\begin{array}{l}
sizeElementC : AuditC \longrightarrow \mathbb{N} \\
sizeLogC : \mathbb{F}\, AuditC \longrightarrow \mathbb{N} \\
\hline
sizeElementC = AuditC \times \{sizeAuditElement\} \\
sizeLogC = \{X : \mathbb{F}\, AuditC \bullet X \mapsto (sizeAuditElement * \#X)\}
\end{array}$$

▷ See: *AuditC* (p. 30), *sizeAuditElement* (p. 31)

$$\begin{array}{l}
\underline{AuditLogR} \\
AuditLog \\
AuditLogC \\
\hline
auditLog = auditR(\!| \bigcup(\mathrm{ran}\ logFiles) |\!) \\
auditAlarmC = auditAlarm
\end{array}$$

▷ See: *AuditLogC* (p. 32), *auditR* (p. 150)

▷ The *auditLog* is the contents of all the *logFiles*.

B.8 Key Store

FD.KeyStore.Retrieval	
FD.KeyStore.State	*FS.KeyStore.State*

$$\begin{array}{l}
\underline{KeyStoreR} \\
KeyStore \\
KeyStoreC \\
\hline
ownName = \{key : keys \mid key.keyType = private \bullet userR\ key.keyOwner\} \\
issuerKey = \{key : keys \mid key.keyType = public \bullet userR\ key.keyOwner \mapsto key.keyData\}
\end{array}$$

▷ See: *KeyStoreC* (p. 33), *private* (p. 14), *userR* (p. 143), *public* (p. 14)

B.9 System Statistics

FD.Stats.Retrieval
FS.Stats.State *FD.Stats.State*

$$
\begin{array}{|l}
\multicolumn{1}{l}{StatsR} \\ \hline
Stats \\
StatsC \\ \hline
successEntry = successEntryC \\
failEntry = failEntryC \\
successBio = successBioC \\
failBio = failBioC \\ \hline
\end{array}
$$

▷ See: *StatsC* (p. 34)

from this we can define a total retrieval bijection for system statistics.

$$
\begin{array}{|l}
statsR : StatsC \rightarrowtail\!\!\!\twoheadrightarrow Stats \\ \hline
statsR = \{Stats; StatsC \mid StatsR \bullet \theta StatsC \mapsto \theta Stats\}
\end{array}
$$

▷ See: *StatsC* (p. 34), *StatsR* (p. 151)

B.10 Administration

FD.Admin.Retrieval
FD.Admin.State *FS.Admin.State*

$$
\begin{array}{|l}
\multicolumn{1}{l}{AdminR} \\ \hline
Admin \\
AdminC \\ \hline
rolePresent = rolePresentC \\
availableOps = availableOpsC \\
currentAdminOp = currentAdminOpC \\ \hline
\end{array}
$$

▷ See: *AdminC* (p. 35)

B.11 Real World Entities

FD.RealWorldState.Retrieval
FD.RealWorld.State *FS.RealWorld.State*

$$
\begin{array}{|l}
\multicolumn{1}{l}{DoorLatchAlarmR} \\ \hline
DoorLatchAlarm \\
DoorLatchAlarmC \\ \hline
currentTime = currentTimeC \\
currentDoor = currentDoorC \\
currentLatch = currentLatchC \\
doorAlarm = doorAlarmC \\
latchTimeout = latchTimeoutC \\
alarmTimeout = alarmTimeoutC \\ \hline
\end{array}
$$

▷ See: *DoorLatchAlarmC* (p. 35)

$$
\begin{array}{|l}
\textbf{UserTokenR} \\
\hline
UserToken \\
UserTokenC \\
\hline
currentUserTokenC \mapsto currentUserToken \in tokenTryR \\
userTokenPresence = userTokenPresenceC \\
\hline
\end{array}
$$

▷ See: *UserTokenC* (p. 36), *tokenTryR* (p. 148)

$$
\begin{array}{|l}
\textbf{AdminTokenR} \\
\hline
AdminToken \\
AdminTokenC \\
\hline
currentAdminTokenC \mapsto currentAdminToken \in tokenTryR \\
adminTokenPresence = adminTokenPresenceC \\
\hline
\end{array}
$$

▷ See: *AdminTokenC* (p. 36), *tokenTryR* (p. 148)

$$
\begin{array}{|l}
\textbf{FingerR} \\
\hline
Finger \\
FingerC \\
\hline
fingerPresence = fingerPresenceC \\
\hline
\end{array}
$$

▷ See: *FingerC* (p. 36)

$$
\begin{array}{|l}
\textbf{FloppyR} \\
\hline
Floppy \\
FloppyC \\
\hline
currentFloppyC \mapsto currentFloppy \in floppyR \\
writtenFloppyC \mapsto writtenFloppy \in floppyR \\
floppyPresence = floppyPresenceC \\
\hline
\end{array}
$$

▷ See: *FloppyC* (p. 36), *floppyR* (p. 148)

$$
\begin{array}{|l}
\textbf{ScreenR} \\
\hline
Screen \\
ScreenC \\
\hline
screenStatsC \mapsto screenStats \in screenTextR \\
screenMsgC \mapsto screenMsg \in screenTextR \\
screenConfigC \mapsto screenConfig \in screenTextR \\
\hline
\end{array}
$$

▷ See: *ScreenC* (p. 36), *screenTextR* (p. 149)

▷ As the abstract *Screen* does not include components for displaying the current alarms, these are free.

From this we can define a retrieval relation for screens.

$$screenR : ScreenC \leftrightarrow Screen$$

$$screenR = \{Screen; ScreenC \mid ScreenR \bullet \theta ScreenC \mapsto \theta Screen\}$$

▷ See: *ScreenC* (p. 36), *ScreenR* (p. 152)

KeyboardR

Keyboard
KeyboardC

$$keyedDataPresence = keyedDataPresenceC$$

▷ See: *KeyboardC* (p. 36)

B.12 Internal State

FD.Internal.Retrieval	
FS.Internal.State	*FD.Internal.State*

The retrieval relation for the Internal state is trivial.

InternalR

Internal
InternalC

$$status = statusC$$
$$enclaveStatus = enclaveStatusC$$
$$tokenRemovalTimeout = tokenRemovalTimeoutC$$

▷ See: *InternalC* (p. 37)

B.13 The whole Token ID Station

FD.TIS.Retrieval	
FD.TIS.State	*FS.TIS.State*

The retrieval relation for the whole Token ID Station is constructed from combining the retrieval relations for the state components, with the addition of retrieval rules for the remaining state components.

IDStationR

$IDStation$
$IDStationC$

$UserTokenR$
$AdminTokenR$
$FingerR$
$DoorLatchAlarmR$
$FloppyR$
$KeyboardR$
$ConfigR$
$StatsR$
$KeyStoreR$
$AdminR$
$AuditLogR$
$InternalR$

$currentDisplay = currentDisplayC$
$currentScreenC \mapsto currentScreen \in screenR$

▷ See: *IDStationC* (p. 38), *UserTokenR* (p. 152), *AdminTokenR* (p. 152), *FingerR* (p. 152), *DoorLatchAlarmR* (p. 151), *FloppyR* (p. 152), *KeyboardR* (p. 153), *ConfigR* (p. 147), *StatsR* (p. 151), *KeyStoreR* (p. 150), *AdminR* (p. 151), *AuditLogR* (p. 150), *InternalR* (p. 153), *screenR* (p. 153)

C APPENDIX: EXAMPLE REFINEMENT

This chapter presents part of the refinement argument, showing that the Formal Design is a correct refinement of the Formal Specification.

The refinement that we have carried out from formal specification to design is not particularly complex. For this reason, and to constrain costs, we have focused on the parts we believe will give the best cost-benefit. We have therefore carried out hand proofs of pre-conditions (that the pre-conditions of the designed operations are at least as permissive as the pre-conditions of the specified operations) and of the correctness of the most complex design step: auditing.

All of these proofs have hand-written documentation. The benefit to the correctness of the system stems from the action of doing the proofs, not of documenting them. If we expected this system to have a long life and be subject to maintenence, we would document the proofs in electronic form.

For the purposes of this project, we have documented here the correctness proof for the audit actions.

C.1 Refinement proof obligations

The general proof rules for refinement in Z are given below. These are a simplification of the common ‘forward’ proof rules, sufficient in most situations.

We use the following general schemas:

Abstract State	A
Abstract Initialisation	$AInit$
Abstract Operation	AOp
Concrete State	C
Concrete Initialisation	$CInit$
Concrete Operation	COp
Retrieve between A and C	R

Initialisation

Proof that whenever the concrete system can be initialised ($CInit$), it is possible to find an abstract state that both retrieves (R) and correctly initialises ($AInit$). “If you can switch on the concrete, you could have achieved the same by switching on the abstract.”

$$CInit \vdash \exists A \bullet AInit \land R$$

Applicability (pre-conditions) Proof that whenever there is a concrete state that retrieves to an abstract state able to undergo the abstract operation (R contains both C and A), then the concrete state is also able to undergo the equivalent concrete operation (COp). “Concrete operations are applicable whenever the abstract operation is.”

$$R \mid \mathrm{pre}\ AOp \vdash \mathrm{pre}\ COp$$

Correctness Proof that whenever a concrete operation (COp) is carried out when the abstract operation would also have been allowed ($\mathrm{pre}\ AOp$), then the answer achieved (the C' in COp) is an allowed answer ($\exists A'$) from the abstract operation (*AOp*). "A concrete operation always yields an answer that could have been seen in the abstract."

$$R; COp \mid \mathrm{pre}\ AOp \vdash \exists A' \bullet AOp \land R'$$

C.2 Audit correctness proof

The most complex step in the design is the realisation of the abstract auditing process as writing to a series of individual audit files.

We can draw the auditing part out by noticing that it appears in the abstract and the concrete conjoined with the 'meat' of each operation, but acting on entirely independent variables. *AddElementsToLog* and *AddElementsToLogC* act on *AuditLog* and *AuditLogC* respectively, using only the variable *newElements*?, which is defined by the meat of the operation. Therefore, it is valid to consider refining *AddElementsToLog* by *AddElementsToLogC* in isolation.

The design tackles auditing in two stages: first strictly declaratively, and then recursively element-by-element. We will consider the refinement of the declarative version first.

C.2.1 Declarative version

The abstract has the variable *newElements*? embedded within it, existentially quantified. We can draw it out explicitly to make the signatures of the abstract and concrete compatible without altering the underlying meaning of the schemas. We can define a schema in the obvious way that has the property

$$AddElementsToLogExplicit \setminus (newElements) \equiv AddElementsToLog$$

The design is expressed as a disjunction of four behaviours. The abstract operation is total, provided that the recorded times in *newElements* are all newer that all the times already in the logs. This is equivalent to the concrete requirement that all *newElements*? have times newer than *nowC*, as all elements in the logs must have been added in previous cycles, and time only increases.

(Note that we also require *newElements* to be non-empty, which it will be in use.)

The concrete is a little less total: $\#newElements? < maxLogFileEntries$. We accept this as a practical limitation, and ensure only that no cycle can ever produce more log entries than allowed by this constraint.

We have now simplified the correctness proof obligation to:

$$\begin{array}{l}
ConfigR; AuditLogR; AddElementsToLogC \mid 0 < \#newElements? < maxLogFileEntries \\
\vdash \\
\exists AuditLog'; newElements : \mathbb{F}\, Audit \bullet \\
\quad AddElementsToLogExplicit \\
\quad \land\ newElements = auditR(\!| newElements? |\!) \\
\quad \land\ AuditLogR'
\end{array}$$

▷ See: *ConfigR* (p. 147), *AuditLogR* (p. 150), *AddElementsToLogC* (p. 51), *auditR* (p. 150)

The four disjuncts cover the range of inputs: no elements; enough for current file; too many, but got another file; and the rest. So having shown they cover the pre-condition sufficiently, we need only show that each one independently refines the abstract.

For simplicity, we will take the bijection

$$auditR : AuditC \rightarrowtail\!\!\!\!\twoheadrightarrow Audit$$

▷ See: *auditR* (p. 150), *AuditC* (p. 30)

as read, and identify *newElements*? and *newElements*.

AddElementsToLog refined by AddNoElementsToLog

$$\begin{array}{l} ConfigR; AuditLogR; AddNoElementsToLog \mid 0 < \#newElements? < maxLogFileEntries \\ \vdash \\ \exists AuditLog' \bullet \\ \quad AddElementsToLogExplicit \\ \quad \land AuditLogR' \end{array}$$

▷ See: *ConfigR* (p. 147), *AuditLogR* (p. 150), *AddNoElementsToLog* (p. 48)

Extends pre-condition to empty *newElements*?, so hypothesis is always false.

The result is proved.

AddElementsToLog refined by AddElementsToCurrentFile

$$\begin{array}{l} ConfigR; AuditLogR; AddElementsToCurrentFile \mid 0 < \#newElements? < maxLogFileEntries \\ \vdash \\ \exists AuditLog' \bullet \\ \quad AddElementsToLogExplicit \\ \quad \land AuditLogR' \end{array}$$

▷ See: *ConfigR* (p. 147), *AuditLogR* (p. 150), *AddElementsToCurrentFile* (p. 49)

We choose to prove the first disjunct of *AddElementsToLog* only (which we are free to do, and will in fact be the case because we are not truncating the logs.)

From the retrieve relation we know

$$auditLog = \bigcup(\operatorname{ran} logFiles)$$

▷ See: *AuditLogR* (p. 150), *auditR* (p. 150)

then the predicate in *AddElementsToLogExplicit*

$$auditLog' = auditLog \cup newElements?$$

clearly retrieves from the predicate in *AddElementsToCurrentFile*

$$logFiles' = logFiles \oplus \{currentFile \mapsto logFiles\ currentLogFile \cup newElements?\}$$

(Note that the logic also works if we choose random log files rather than the current one. We need only ensure that the file whose size we check is the file we use.)

To prove the predicates on *alarming*, we need to relate the concrete sizes and numbers of audit elements to the abstract size functions. From the *alarming* predicate in *AddElementsToCurrentFile* take

$$numberLogEntries' \geq alarmThresholdEntries$$

Multiply both sides by *sizeAuditElement*

$$numberLogEntries' * sizeAuditElement \geq alarmThresholdEntries * sizeAuditElements$$

But *ConfigC* tells us that

$$alarlThresholdEntries * sizeAuditElement \geq alarmThresholdSizeC$$

and therefore we can deduce

$$numberLogEntries' * sizeAuditElement \geq alarmThresholdSizeC$$

From the retrieves, and the properties of *sizeLog* given with the retrieves, these values can be replaced with

$$sizeLog\ auditLog' \geq alarmThresholdSize$$

as needed for the abstract predicate.

The second predicate is derived similarly:

$$numberLogEntries' < alarmThresholdEntries$$

Replace the strict less-than by reducing the RHS by 1 (they are integers)

$$numberLogEntries' \leq alarmThresholdEntries - 1$$

Multiply both sides by *sizeAuditElement*

$$numberLogEntries' * sizeAuditElement \leq (alarmThresholdEntries - 1) * sizeAuditElements$$

But from *ConfigC* the RHS is strictly less than *alarmThresholdSizeC*, giving us

$$numberLogEntries' * sizeAuditElement < alarmThresholdSizeC$$

From the retrieves, and the properties of *sizeLog* given with the retrieves, these values can be replaced with

$$sizeLog\ auditLog' < alarmThresholdSize$$

This gives us the predicates on alarming, and completes this branch.

AddElementsToLog refined by AddElementsToNextFileNoTruncate

$$
\begin{array}{l}
\mathit{ConfigR};\ \mathit{AuditLogR};\ \mathit{AddElementsToNextFileNoTruncate} \mid 0 < \#\mathit{newElements}? < \mathit{maxLogFileEntries} \\
\vdash \\
\exists\, \mathit{AuditLog}' \bullet \\
\quad \mathit{AddElementsToLogExplicit} \\
\quad \land\ \mathit{AuditLogR}'
\end{array}
$$

▷ See: *ConfigR* (p. 147), *AuditLogR* (p. 150), *AddElementsToNextFileNoTruncate* (p. 49)

The argument runs exactly as above, but now *newElements*? is split between *elementsInCurrentFile* and *elementsInNextFile*. But these get combined directly in $\bigcup(\mathrm{ran}\ \mathit{logFiles})$, so all the same arguments hold.

AddElementsToLog refined by AddElementsToNextFileWithTruncate

$$
\begin{array}{l}
\mathit{ConfigR};\ \mathit{AuditLogR};\ \mathit{AddElementsToNextFileWithTruncate} \mid 0 < \#\mathit{newElements}? < \mathit{maxLogFileEntries} \\
\vdash \\
\exists\, \mathit{AuditLog}' \bullet \\
\quad \mathit{AddElementsToLogExplicit} \\
\quad \land\ \mathit{AuditLogR}'
\end{array}
$$

▷ See: *ConfigR* (p. 147), *AuditLogR* (p. 150), *AddElementsToNextFileWithTruncate* (p. 50)

Choose to refine the second branch of the abstract schema, which we can choose whenever

$$\mathit{sizeLog}\ \mathit{auditLog} + \mathit{sizeLog}\ \mathit{newElements}? > \mathit{minPreservedLogSize}$$

We know this is true from the hypothesis because only one file is discarded, and as the implementation has the property that all-files-minus-one is bigger than *maxSupportedLogSize* (which is itself bigger than *minPreservedLogSize*), we always preserve at least this size of audit information, and we only ever consider truncating when larger than *minPreservedLogSize*.

The property on audit log holding the correct elements is again achieved by the assignment of $\mathit{logFiles}'$, together with correct time constraints. The choice of the file to discard as the head of the list of used files ensures it is the oldest.

Alarm is explicitly set in both concrete and abstract operations.

Note the *numberLogEntries* is calculated to preserve its correct value as the number of entires actually stored.

Refinement

We have therefore shown that the abstract audit operations, including the option of truncating the audit log, is correctly refined by the declarative design.

C.2.2 Recursive

We now show that the concrete element-by-element additions are an alternative representation of the same behaviour.

First, we show that *AddElementToLogC* is just a specialisation of *AddElementsToLogC* for single elements, i.e

$$[AddElementsToLogC \mid \#newElements? = 1] \equiv AddElementToLogC$$

We consider two cases:

Truncate not required

The precondition for the single element schema can be derived from *TruncateLogNotRequired* and *AddElementsToLogFile*. It is

$$freeLogFiles \neq \varnothing \land \#(logFiles\ currentLogFile) = maxLogFileEntries$$
$$\lor\ \#(logFiles\ currentLogFile) < maxLogFilesEntries$$

The precondition for the multiple element schema is

$$freeLogFiles \neq \varnothing \land \#newElements? + \#(logFiles\ currentLogFile) > maxLogFilesEntries$$
$$\lor\ \#newElements? + \#(logFiles\ currentLogFile) \leq maxLogFilesEntries$$

Assuming a single element in *newElements*?, we can replace $\#newElements?$ with 1, and given that the sizes are integers, these can be seen to be identical (given that we can show that the size of the log files never actually exceeds *maxLogFileEntries*).

Both schemas break into two disjuncts:

current file: which can be seen to be identical in the two by inspection, and

next file: which can also be seen to be identical in the two by inspection, given that we can choose *elementsInCurrentFile* to be empty and hence $elementsInNextFile = newElements?$.

Truncate is required

The precondition for the single element schema is derived from the sequential composition of three schemas, but can be seen to be the negation of the precondition for no truncation:

$$freeLogFiles = \varnothing$$
$$\land\ \#(logFiles\ currentLogFile) = maxLogFilesEntries$$

(We do need to confirm that the apparent precondition seen in *TruncateLog* is not restricted by the later sequential compositions. But releasing a log file and reducing the number of log entries ensures that the two applications of *AddElementToLog* will proceed.)

The precondition for the multiple element schema is

$$freeLogFiles = \varnothing$$
$$\land \#newElements? + \#(logFiles\ currentLogFile) \geq maxLogFilesEntries$$

As before, these are the same when $\#newElements? = 1$.

In the single element schema, the log is truncated, then the truncation element added, then the real audit element is added.

We will now look at each predicate in the declarative version and see how its equivalent is constructed by these sequential operations in the single element version.

predicate 1:

$$numberLogEntries' = numberLogEntries + 1 - maxLogFileEntries + 1$$

The subtraction is defined in *TruncateLog*, and each of the additions comes from an application of *AddElementToLogFile*.

predicate 2:

$$\exists\, truncElement \ldots$$

Each component property can be compared with the equivalent in the single element version and seen to be the same.

predicate 3:

$$elementsInCurrentFile \subseteq newElements?$$

Choose this to be empty.

predicate 4:

$$\#(logFiles\ currentLogFile) + \#elementsInCurrentFile = maxLogFileEntries$$

$\#elementsInCurrentFile$ is zero by choice. This predicate is then true by precondition.

predicate 5:

$$elementsInNextFile = newElements? \setminus elementsInCurrentFile$$

By choosing *elementsInCurrentFile* empty, this forces *elementsInNextFile* to equal *newElements*?.

predicate 6:

$$oldestLogTime\ elementsInNextFile \geq truncElement.logTime$$

See predicate 7.

predicate 7:

$$truncElement.logTime \geq newestLogTimeC\ elementsInCurrentFile$$

There are three time intervals in the sequential version: *Truncate*, *AddElementToLogFile* (which adds the truncate audit element), and *AddElementToLogFile* (which adds *newElement*?). Time is forced to move on between each of these intervals, and this constrains these two predicates 6 & 7 to be true.

predicate 8:

$$logFiles' = \ldots$$

Application of *TruncateLog* updates *head usedLogFiles* to empty, then *AddElementToNextLogFile* (this one because of pre-conditions) adds the truncation element (due to renaming in composition) to this file, updating *currentLogFile* to this file (which is the only one in the list of *freeLogFiles*, put there by *TruncateLog*), and then *AddElementToCurrentLogFile* (because only single element in this file now, so conditions choose this one) adds the *newElement*? to this.

predicate 9:

$$currentLogFile' = head\ usedLogFiles$$

Explained above with predicate 9.

predicate 10:

$$usedLogFiles' = tail\ usedLogFiles \frown \langle currentLogFile' \rangle$$

Truncate tails, then next adds the new one, then next leaves it alone.

predicate 11:

$$freeLogFiles' = freeLogFiles$$

Adds one, removes it, leaves alone.

predicate 12:

$$logFilesStatus' = logFileStatus \oplus \{currentLogFile' \mapsto used\}$$

Set to free, then used, then left.

predicate 13:

$$auditAlarmC' = alarming$$

Truncate sets, rest leave it alone.

So *AddElementToLogC* is equivalent to *AddElementsToLogC* for an individual element, as we wished to show.

(We don't actually need to check the recursive definition, because the implementation will actually apply *AddElementToLogC* sequentially, chronologically.)

This is sufficient to show that the design step made from the abstract formal specification to the more concrete design specification is correct.

D APPENDIX: Z INDEX

This section contains an index of Z terms. This contains all the Z schemas, types and functions defined in the specification.

E APPENDIX: TRACEUNIT INDEX

An index of traceunits. This contains all the traceunits placed in the specification to enable the elements of the specification to be traced to the design.

F APPENDIX: REQUIREMENTS INDEX

An index of traceunits. This contains all the traceunits in the requirements documents . All requirements are listed with the pages from which they are referenced.