ROYAL SIGNALS & RADAR
ESTABLISHMENT

TRIAL IMPLEMENTATION OF A SECURE APPLICATION
USING Ten15

Author: E R Blissby

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

DISTRIBUTION STATEMENT A
Approved for public disclosure
Distribution: Unlimited

UNLIMITED 91-02933
CONDITIONS OF RELEASE

COPYRIGHT (c) 1988
CONTROLLER
HMSO LONDON

Reports quoted are not necessarily available to members of the public or to commercial organisations
ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4465

Title: Trial Implementation of a Secure Application Using Ten15

Author: Elizabeth R. Bilsby

Date: March 1991

ABSTRACT

This report describes how a subset of SERCUS has been implemented using Ten15. SERCUS is a research implementation of a multi-level secure workstation based on the SMITE approach and running a classified document handling application. SMITE is an approach to the construction of secure systems which uses strong typing. Ten15 is an algebraically defined, strongly typed abstract machine running on a VAX station.

This work was performed while the author was a Vacation Student at RSRE and used the Ten15 Cross Compilation System as it existed in Summer 1990.

Copyright © Controller HMSO London 1991
# CONTENTS

1. Introduction ........................................ 1
2. Overview of SERCUS .................................. 2
   2.1 General Overview of SERCUS on the Perq ............ 2
   2.2 The Ten15 Implementation of SERCUS ................. 2
3. The Cross Compilation System .......................... 4
   3.1 Compiling and Type Checking a Module .............. 4
   3.2 PERQ to VAX Transfers ............................. 5
   3.3 Executing a Procedure on the VAX .................... 6
   3.4 Overview of the Ten15 Filestore ...................... 7
4. SERCUS Modes and Ops ................................ 8
5. Overview of the Ten15 Notation ........................ 9
6. Exception Handling Strategy ............................ 11
7. Classifications Module ............................... 13
   7.1 Modes for Classifications Module .................... 13
   7.2 Procedures in the Classifications Module .......... 13
   7.3 Testing the Classifications Module .................. 16
   7.4 General Comments .................................. 16
8. Context Module ....................................... 17
   8.1 Modes for Context Module ........................... 17
   8.2 Procedures in the Context Module ................... 17
   8.3 Testing the Context Module ......................... 19
9. Journalling Module .................................... 21
   9.1 Modes for the Journalling Module .................... 21
   9.2 Procedures in the Journalling Module ................ 21
   9.3 Testing the Journalling Module ...................... 24
10. Login and Related Procedures ......................... 25
    10.1 Modes Required for the Login related Procedures .... 25
    10.2 Ions and Procedures Related to Login ............... 26
11. Regst/ Module ....................................... 34
    11.1 Modes for Registry Module .......................... 34
    11.2 Procedures in the Registry Module .................. 34
12. References .......................................... 41
INTENTIONALLY BLANK
2. OVERVIEW OF SERCUS

2.1. GENERAL OVERVIEW OF SERCUS ON THE PERQ

SERCUS is a research implementation of a multi-level secure workstation running a classified document handling system. The overall security requirement of SERCUS is that classified information cannot be discovered by a user with insufficient clearance, e.g., a secret document cannot be read by a user only cleared to restricted.

SERCUS is essentially an electronic registry system controlling the creation of, and access to classified documents and mail messages. Users are assigned clearances which limit their ability to view and modify information in the system. All users have a personal cupboard where they may store objects such as the documents they are drafting. Whilst in the cupboard, these objects may be referred to by an unclassified name. A unclassified list is maintained of all the finished classified documents in the system, and this is called the Classified Document Registry (CDR). Users may view the CDR and ask to read any of the documents it holds. An additional requirement of documents is that their classification may be altered. However, to ensure that the new classification is appropriate, this requires the agreement of the security officer in addition to the ordinary user.

SERCUS also maintains a journal for each document in which interesting events that have occurred in its life are recorded. For example, which users have accessed its contents, and those who have agreed to re-classification of the document. Additionally, a journal is maintained for each user registered on the system in which security relevant actions are recorded such as when the user logs on and off SERCUS, documents they were prevented from seeing because their clearance was insufficient, and any users to which they have sent mail messages. The user's journal aims to make users accountable for their actions.

When a user logs on to SERCUS, they are presented with a display consisting of a number of non-overlapping windows. All the window software is completely trustworthy, i.e., a trusted path (a trusted path is a validated link between the human user and a system's trusted software which mutually authenticates both parties.) The trusted path may be used to invoke untrusted software such as a commercial word processing package. While untrusted software is active in a window, the classification of the information is displayed prominently. SERCUS monitors the movement of information between windows and uses a high water mark mechanism to correctly maintain the classification levels.

For more information about the SERCUS Secure Registry see RSRE report, 'An Example Secure System Specified Using the Terry-Wiseman Approach', Harrold [1]

2.2 THE TEN15 IMPLEMENTATION OF SERCUS

The present implementation of SERCUS in Ten15 is a demonstrable subset of the above. The Ten15 SERCUS incorporates users, classifications, and clearances, journals, and documents. Documents can be created, opened, and read, and document and user journals can be reviewed. In addition, the classification of a document can be found, as can the clearance of a user. The system
1. INTRODUCTION

This report describes the implementation of a subset of the SERCUS demonstration in the abstract
algebraically defined, strongly typed language Ten15. Ten15 contains constructs that implement
all of the features of a modern high level programming language, as well as facilities for
manipulating system level aspects of a computer such as filestore.

A general overview of SERCUS as implemented on the Perq is provided, along with a summary of
the Ten15 implementation of SERCUS using the Ten15 Cross Compilation System. The use of the
Cross Compilation System is then described, along with an overview of the Ten15 notation.

The report then describes in detail the implementation of the individual modules making up the
Ten15 SERCUS demonstration. These include the classifications, context, journaling, login
and related procedures, and the registry modules. Each section describes the modes and operators
defined in the module SercusModesandOps for use in the module, followed by descriptions of the
procedures contained in the module. The exception/failure strategy employed in the Ten15
SERCUS is also described.

The convention used throughout the report is that Ten15 operators are bolded in the text, as are the
names of the modules used in the SERCUS demonstration.
can also tell the user whether they are on the trusted path, and move users on and off the trusted path as desired. It should be noted that the trusted path and invocation of untrusted software is at present only a simulation.

The Ten15 demonstration, unlike the Perq version, has only one window at present. It also lacks High Water Marks, thus the creation of documents does not perform high water mark related checks before a document is created. Cupboards and mail between users has not yet been incorporated either, and nor has the regrading of documents.
3. THE CROSS COMPILATION SYSTEM

The subset of the SERCUS demonstration described in this report is implemented using the Ten15 Cross Compilation System on the Perq and the VAX machines. The VAX has only one dictionary so there are no separate users and only one process running. As yet there is no Ten15 notation compiler on the VAX, thus the cross compilation system is vital in the establishment and development of software on the VAX while completion of the editor and notation compiler are awaited.

This document assumes the reader has a reasonable working knowledge of the FLEX system.

3.1. COMPILE AND TYPE CHECKING A MODULE

VAX modules for use in the demonstration are written in Ten15 notation contained in edfiles, and compiled on the Perq. Edfiles and compiled modules, etc, appear on the Perq in the form of cartouches (boxes). An edfile containing notation is first compiled by applying the procedure `make_ten15`. This procedure does the syntax checking of the Ten15 notation; it checks such things as declarations and grouping, and produces intermediate Tenfifteen code. If a syntax error is found, then `make_ten15` calls up the Perq editor which will indicate the nature and position of the error(s). When the notation has been successfully checked syntactically, i.e., the editor does not get called, or only gives warnings drawing the users attention to possible oversights in the notation e.g., an object which is declared and not used (possible error in the scope of the object), then the procedure `vax` is applied.

The `vax` procedure does the type checking and generates the VAX code. The type checking detects illegal mode coerction attempts, displaying the offending modes as cartouches. These can be duplicated and examined using the Show Ten15 Mode option on the Ctrl I menu (SMITE Team systems only). If the type checking is unsuccessful, the `vax` procedure calls the Perq editor in the same way.

The error messages produced by both the `make_ten15` procedure and the `vax` procedure are very basic and not always very helpful. This is largely due to the fact that we are using the first embryonic version of the cross compilation system, and the clarity of the error messages is likely to improve as the system is further developed. The `make_ten15` and `vax` procedures are applied to an editable file (edfile) as follows:

```
[edfile]make_ten15 ! vax!
```

N.B. ! represents procedure application on the Perq.

Once the notation has been successfully compiled and translated, it can either be amended into an existing module or made into a new one as follows:

---

1 A useful trick that can be used to check types is to define what you think the type should be, and then to let the system tell you what it thinks it should be if it is unable to coerce the one type to the other.
Creating a Module:

```
(make_ten15 vaxl, "module_name") new_vaxl
```

Amending a Module:

```
(make_ten15 vaxl module_name Module RoPtr..) amend_vaxl!
```

If the specification of the module has changed then it must be change_speced as follows:

```
(make_ten15 vaxl module_name Module RoPrt..) change_spec_vaxl!
```

N.B All other modules which use the change_speced module must also be recompiled. (See later)

3.2. PERQ TO VAX TRANSFERS

Having created and compiled a module on the Perq, the VAX code can be transferred across to the VAX for running in the Ten15 Evaluation System

N.B. The VAX should already be running the Ten15 Evaluation system, invoked by typing

```
> @run_ten15 simple_load15 <RETURN>
> <RETURN>
```

This sequence of commands can be defined in the users login.com so that a shorter command such as ten15 can be used to invoke the Ten15 system.

The kernel module for the Ten15 system must then be loaded by pressing return (since kernel_mod is the default module).

To transfer a module from the Perq to the VAX, the following commands must be obeyed on the two machines with the Perq end being obeyed first.

3.2.1. On Perq:

Apply send_to_vax to the module as follows

```
module_name Module RoPtr.. send_to_vax!
```

N.B. The transfer routine always fails on the Perq with the error message "No Capability for Creating Uniques - see FDH", but should be ignored.

3.2.2. On VAX.

Modules can be transferred to the VAX by issuing the relevant commands from either the true (command line interpreter) or from the editor itself.
Apply the procedure Transfer_module to the name of the Perq holding the required module to
Transfer_module."perq name". Note that procedure application is indicated by a \( \cdot \) on the VAX, and that the notation is forward polish rather than reverse polish as on the Perq.

True only has a basic teletype interface so values created are given names of the form @0, @1 etc. To use the module it must be loaded using Load which delivers the re-pointer to the keep list of the VAX module, then opened using open, and finally named either temporarily using \( \cdot \), or permanently in the dictionary using \( \cdot \cdot \cdot \), as follows:

```
Input Transfer_module "tigger"
@0
Input Load @0
@1
Input open @1
PrTo @2
Input @2 \cdot proc1
```

From the editor:

The editor must first be called from true by obeying the command Load.edit, or login() if SERCUS is to be run as follows:

```
Input Load edit
```
or
```
Input login()
```

Once in the editor the command Transfer_module."perq name" can be evaluated within an evaluation box. The commands used to load a module from the editor are similar to those issued from true, except cartouches are used instead of the @ numbers. The module is loaded using Load, then the pointer to the keep list is de_pointered using deptr:

```
Transfer_module "tigger" = m
deptr (Load m) \cdot proc2
```

N B The command line interpreter in the editor and true use the same dictionary.

### 3.3 EXECUTING A PROCEDURE ON THE VAX

Having transferred a module from the Perq to the VAX, loaded it into the Ten15 Evaluation System and de_pointered its keep list, the procedure can be applied to its parameters.

The syntax for procedure calls on the VAX is procedure.parameter. For example proc1 takes void ie no parameters and proc2 takes two parameters, thus to execute a procedure whose name is stored either temporarily or permanently, type

```
proc1
```

```
proc2 @1 @2
```
3.4. OVERVIEW OF THE TEN15 FILESTORE

The Ten15 filestore is simply one large VAX file. The implementation of the SERCUS demonstration will involve some manipulation of the Ten15 filestore, so a brief overview of terminology is provided here.

Filestore values in Ten15 are accessed via filestore capabilities, and these are of two types:

1. Persistent Values (persistents)
2. Persistent Variables (pvars)

3.4.1. Persistent Values

These are mainstore values written to filestore (persisted). They can be read (unpersisted) via their disc capability, but they cannot be assigned to. The operator Persist is used to store a representation of a mainstore value on datastore, returning a persistent value.

N.B. Persist takes the datastore as a parameter as well.

While the operator UnPersist uses this persistent to retrieve a copy of the value originally stored on the datastore.

3.4.2. Persistent Variables

These are essentially updateable references on disc containing a persistent which can be assigned to. (Persistents may contain other persistents or pvars). The operator AssPvar assigns a persistent value to a pvar, and returns void. DePvar is used to retrieve and deliver the persistent value most recently assigned to a pvar.

3.4.3. Pointer

A pointer points to something in main memory. The operator Pack generates and delivers a pointer to an area of memory of appropriate size which has been initialised to contain the value. Conversely the operator De_ptrs the pointer, returning the value most recently assigned to it.

3.4.4. PSet

A Pset is a set of Pvars
4. SERCUS MODES AND OPS

SercusModesandOps is a MoModule. It contains all the mode, operator and assertion definitions required for any part of the Ten15 notation used in the SERCUS demonstration. The mode and operator names available to a module are all those declared in the MoModule, plus all those that occur in the use-lists defined in the MoModule.

SercusModesandOps

(Basic_kernel_modes : MoModule)

Use Del, DelVac, DelSimpLine, Line
Operators : ToExistsDelSimpLine, ToExistsDelVac
From [EditLineMo : MoModule]

Modes:
--- Described in Following Sections -----

Operators:
Output = 402;
Input = 403;
GetTime = 407;
VduMessage = 528(S)

Finish

Definitions can be used from previously defined MoModules by specifying a use-list of the mode and operator names that need to be made available from it eg. EditLineMo above. If no use-list is specified then all the definitions in the module will be passed through, eg. Basic_kernel_modes above. All other modes, operators and assertions are then defined.

The MoModule is compiled by applying the procedure modesandops to the edfile containing the definitions, and then created by applying the procedure new_mo.

editl modesandops (modes_module) new_mo

If new definitions are added to an existing MoModule then it can simply be amended using amend_mo:

editl modesandops [modes_module] amend_mo!

However if definitions are changed or removed then the MoModule will have to be change-specified using change_spec_mo:

editl modesandops [modes_module] change_spec_mo!

N B. All Modules using this particular MoModule will have to be recompiled. This is best done by applying recompile_vax to a top module.
5. OVERVIEW OF THE TEN15 NOTATION

The edfile containing the Ten15 notation must have a single MoModule at the top. MoModules are the only place where modes and operators can be declared, and all the modes and operators in the notation report, "A Notation for Ten15" Goodenough and Rees [2], are automatically defined in the MoModule.

The MoModule is followed by Ten15: and the end of the notation is indicated by Finish. All terminal symbols in the notation start with a capital letter and then continue in lower case. The end of a clause is indicated by the key word reversed or Endkeyword (where the keyword is generally in lower case) eg.

```
if x Then y Else z Fi
In ...Endproc
```

Since a full modular compilation system does not as yet exist, keep lists have to be constructed by hand. The operators Ro and Pack are used in the last statement in the module to pack whatever it is you wish to keep into a read only block, returning a pointer to this block. eg.

```
Ro Pack (proc1, proc2, proc3)
```

Objects kept in one module may be used in another by extracting them from their defining module using a Let statement eg.

```
Let (proc1, proc2, proc3) = anymodule : Module(RoPtr)
```

Commas can be used if the kept values are to be ignored eg.

```
Let (...) = anymodule : Module(RoPtr)
```

N.B. The Module is automatically loaded when it is encountered in the text, and the keeps extracted.

Identifiers and variables defined within declarative statements (ie. Let and Var statements), are only in scope between the following In...Nil. Declarative statements can also be used to declare the type of an identifier or variable. The translator compares the user defined type with the actual type the object should have. The editor is called on the Perq allowing the user to examine the modes and discover the error if the types fail to match eg.

```
Let identifier3 : Model = proc1(identifier1, identifier2)
```

N.B. Individual Let statements require no list separators, while statements within the scope of a Let statement need to be separated by semi-colons.

---

1 Getting scopes wrong can lead to contradictory error messages such as "Warning - x declared and not used" and "x not declared".
The Ten15 notation for procedures allows them to be named using a Let statement, and their non-locals to be explicitly identified using a Use..In statement as follows:

```
Let proc1 = Use ( non-local1, non-local2 ) In
  Proc( identifiers1, identifier2 : Model2 ) => Model1;

..... procedure body .....;
Endproc
```

N.B. There is no closing In for the In which identifies the non-locals to the procedure.
6. EXCEPTION HANDLING STRATEGY

Exceptions occur when for example the user_details or trusted path flag cannot be extracted from the context or the cdr number supplied is out of the range of the registry. In such cases the procedure where the exception occurred should fail.

The following modes are defined for use in handling exceptions in the code. Failure is a structure of an integer representing the failure number, and a string representing the corresponding failure message. FailVec is a vector of entries of type Failure.

```
Failure = struct(Int Number, String Fail_Mess)
FailVec = vec(Failure, PosInt)
```

A module, FailureNumbers, is provided for use by all the other modules in the SERCUS demonstration. It contains a vector of Failure entries consisting of a failure number and a failure message relating to the various exceptions that could possibly arise while running the demonstration. These failure messages will be used in the diagnosis of exceptions throughout the code, eg.

```
Let base : Int = 60914
Let class_fad : Int = base +1
Let class_mass : String = "Invalid Classification"
Let class_details : Failure = (class_fad, class_mass) etc.
Let fail_vec : FailVec = Vec(7 Of class_details)
In fail_vec2 := .......... etc.
```

N.B. The base number is a random large number.

When an error is detected in a module, the operator Failure is used. This causes the procedure in which it was called to fail, and an exception is formed from the trap value. The trap value is generated by applying the operator IntToTrap to the appropriate integer failure number taken from the module FailureNumbers.

```
Let (class_fad, ....) = FailureNumbers : Module(TopPr1)
In Failure IntToTrap class_fad
```

This exception can either be caught in the enclosing procedure, which is usually the case, or propagated on further to be caught at a later date.

The operator Trapply takes a procedure and it's parameters and returns a union. This union contains the result of the procedure if it completed normally, the trap value if an exception was caused during the execution of the procedure, or if the procedure was jumped out of, a procedure with which to complete the long jump. (This third option will not apply in the SERCUS demonstration code).
A procedure is defined in the module FailureNumbers which takes the trap value generated using the operators IntToTrap and Failure from the failure number defined in FailureNumbers corresponding to the nature of the error. It returns the string which diagnoses the error, also defined in FailureNumbers.

\[
\text{Let } \text{fail\_proc} = \text{Use (fail\_vec, base) In}
\]
\[
\text{Proc (fail\_trap, trap) -> String:}
\]
\[
\text{Let trap\_no : Int = TrapToInt(fail\_trap)}
\]
\[
\text{Let fail\_no : Int = trap\_no + base}
\]
\[
\text{Let details : @ (fail\_vec VecInt fail\_no)}
\]
\[
\text{Let message : String = Fail\_Mess details}
\]
\[
\text{In message}
\]
\[
\text{NS}
\]
\[
\text{Endproc}
\]

The operator TrapToInt is used to retrieve the failure reason from the trap value ie. the value used by IntToTrap to create the trap. The base number is then extracted from this number to give the index into fail_vec. The appropriate diagnostic message can then be extracted from the vector using field extraction.

This procedure is used within Trapply as follows:

\[
\text{Let byebye = Module(RetPr...)}
\]
\[
\text{Trapply (read doc, cdr\_num}
\]
\[
| \text{OpText: text}
\]
\[
| \text{Op trap:}
\]
\[
| \text{Let mess = fail\_proc (trap)}
\]
\[
| \text{In message (mess);}
\]
\[
| \text{Failure IntToTrap byebye}
\]
\[
| \text{NS}
\]
\[
| \text{Op Jmp: Jmp0}
\]

N.B. The exception is propagated by calling Failure IntToTrap on the integer byebye, a large number used to signify fatal errors.

Error diagnostics are written to the bar at the top of the currently active window using the procedure message which uses the system operator VduMessage with the appropriate parameters ie. the display and window dimensions and the error message string.
7. CLASSIFICATIONS MODULE

This module is used to provide a representation of document and user classifications for the SERCUS demonstration.

7.1. MODES FOR CLASSIFICATIONS MODULE

The classifications mode is defined as the range of integers 1..5 as follows:

\[ \text{Class} = \text{range}(4, 1..5) \]

The 4 indicates that it is an integer range, rather than a character or long integer range.

The classifications represented by the hierarchy of integers are:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>unclassified</td>
</tr>
<tr>
<td>2</td>
<td>restricted</td>
</tr>
<tr>
<td>3</td>
<td>confidential</td>
</tr>
<tr>
<td>4</td>
<td>secret</td>
</tr>
<tr>
<td>5</td>
<td>top secret</td>
</tr>
</tbody>
</table>

String mode (used throughout the code) defines a string as a read only vector of characters with a positive integer as its upper bound.

String = ro_vec( Char, Postint )

7.2. PROCEDURES IN THE CLASSIFICATIONS MODULE

The classifications module contains the procedures required to implement classifications in Ten15 notation for the SERCUS demonstration. They are as follows:

7.2.1. Dominates

Dominates is a procedure which takes two parameters of type Class and returns True if \( a \geq b \), and False otherwise i.e. Unclassified (1) is dominated by Restricted (2) since \( 2 \geq 1 \), but Restricted is not dominated by Unclassified since \( 1 \geq 2 \) is false. This procedure is implemented using the basic integer greater than or equal operator, \( \geq \).

\[
\text{let dominates} = \text{Use}() \text{ In}
\begin{align*}
\text{Proc} (a, b : \text{Class}) & \rightarrow \text{Bool}.
\text{a} \geq \text{b}
\end{align*}
\]

7.2.2. Least

Least is a procedure which takes two Class integers and returns the lowest dominating classification in the pair, i.e. given the pair of classifications Confidential (3) and Unclassified (1), the lowest dominating classification is Confidential. This procedure is implemented using
the basic operator \texttt{Max}, which delivers the maximum of two integers. \texttt{Max} is equivalent to the construct "If \( a > b \) Then \( a \) Else \( b \) Fi".

\begin{verbatim}
Let least = Use () In
Proc (a, b : Class) => Class:
  a Max b
Endproc
\end{verbatim}

7.2.3. Greatest

Greatest is a procedure which takes two Class integers and returns the greatest non dominating classification in the pair, ie. given the pair Secret (4) and Restricted (2), then the greatest non dominating classification is Restricted. This procedure is implemented using the basic operator \texttt{Min}, which delivers the minimum of two integers. \texttt{Min} is essentially equivalent to "If \( a < b \) Then \( a \) Else \( b \) Fi".

\begin{verbatim}
Let greatest = Use () In
Proc (a, b : Class) => Class:
  a Min b
Endproc
\end{verbatim}

7.2.4. Class.to_str

\texttt{Class.to_str} takes a Class integer and returns the classification string which the integer represents. For example the integer 1 represents the classification string \texttt{Unclassified}

The following shows how the vector of valid classifications, \texttt{valid.class}, is constructed. Firstly five lower case strings corresponding to the five classification classes are declared.

eg.

\begin{verbatim}
Let unclassified : String = "unclassified"
Let restricted : String = "restricted" etc.
\end{verbatim}

\texttt{Valid.class} is defined as a vector of five elements of type \texttt{unclassified}, and then the remaining four elements in the vector are set to the other four classifications strings defined above. The \textsc{Tn15} compilation system is able to work out automatically the replacement of strings in the vector by strings of different lengths.

\begin{verbatim}
Let valid.class = Vec(5 Of unclassified)
In
  valid.class = restricted; etc.
\end{verbatim}

The procedure \texttt{class.to.str} is implemented using vector indexing into the \texttt{valid.class} vector as follows:

\begin{verbatim}
Let class.to.str = Use (valid.class) In
Proc (class : Class) => String:
  Let class_str : String = @ (valid.class VecInd class)
  In
    class_str
Endproc
\end{verbatim}
N.B. The operator @ is used to de_ref a reference or vector, returning the value most recently assigned to it.

7.2.5. Str_to_class

This procedure returns the Class integer corresponding to a classifications string. Initially its primary use will be in the testing of the classifications module, although it will also be used to assign clearances to users. It will eventually be used in the creation and regrading of documents which require the user to be prompted to enter a classification.

The first implementation of str_to_class used an If..Then..Else..FI statement to recognise the classification string as a whole eg. Unclassified, or the first letter of the string eg. U, to facilitate the return of the correct class. This implementation was by no means ideal. A better implementation of str_to_class was found which involved converting the given string to upper case characters, so that it is not case sensitive, and then matching it against the classifications strings held in the valid_class vector, which are also converted to upper case.

```
Let to_upper = [to_upper : Module( RoPtr, )]
Let { class Fail..... } = [FailureNumbers : Module( RoPtr, )]

Let str_to_class = Use( valid_class, to_upper, class Fail ) In
Proc( str : String ) => Class:
  Let upb : PosInt = Upo str
  Let new_str : String = to_upper( str )
  Var i : ref PosInt := 0
  In
    For n To Upo valid_class While @i = 0
      Do
        Let class_str : String = @( valid_class VectId n )
        Let compare : String = to_upper( class_str )
        Let trimmed : String = compare Trim upb
        In
          If trimmed VecEq new_str
            Then
              i = n
            FI
        FI
      Od;
      If @i = 0
        Then
          Failure Info To Trap class Fail
        Else
          CiClass @i
      FI
  Endfor
Endproc
```

The idea of the procedure is to take a classifications string, convert it to upper case characters and find its length, say upb. Before comparison, each string in the valid_class vector is trimmed to upb characters using the vector operation Trimc (Trim when less than or equal to upb). The string is then compared with the first upb characters of the upper case conversion of each string in the valid_class vector, using a For..To..While..Do..Od loop to index the vector until a match is

1. @ without a space is used for labels eg. @label. When @ is used to de_ref an object, care must be taken to ensure a space is inserted as this is a very difficult error to spot.
found, if one exists. The Class represented by the string is returned, and this corresponds to the index into the vector which provided the match. This Class must be changed ranged using the operator \texttt{CrClass} which takes an operand and converts it into a value with type Class, so that the value returned will actually be in the range 1..5, e.g. 1 is given range 1..1.

If no match is found then the procedure will fail using the \texttt{Failure} operation which forms an exception from the trap value. The operation \texttt{IntToTrap} is used to create the trap value from the integer class\_fail. Class\_fail is defined in the FailureNumbers module along with an appropriate error message. All other failure numbers and messages are also defined in this module. The exception can be ‘caught’ in an enclosing procedure using \texttt{Trapply}.

This implementation of \texttt{str\_to\_class} is much more versatile since any shortening of the classification classes, ranging from the initial letter to the whole word will be recognised.

7.3. TESTING THE CLASSIFICATIONS MODULE

Unfortunately in the cross compilation system at present there is no capability to examine structures in the Ten15 editor in order to extract procedures etc. for test purposes. Thus the classifications module was not sent directly to the VAX, but instead five test procedures were written in separate modules to test each of the five classification procedures, and these were sent separately.

The test procedures for dominates, least, greatest, and \texttt{class\_to\_str} require Class integers as parameters. The procedure \texttt{str\_to\_class} must therefore be applied to some classifications strings so as to produce Class integers for use in the test procedures.

\begin{verbatim}
str\_to\_class("Unclassified") = u  etc.
\end{verbatim}

This will give for example, the string "Unclassified" the classification 1, and name it u. Some examples of testing the classifications procedures are:

\begin{verbatim}
dommates(s, u)  
least(u, c)  
greatest(t, t)  
\texttt{class\_to\_str}(s)
\end{verbatim}

7.4. GENERAL COMMENTS

The classifications module as specified above forms an abstract data type (ADT) representing the classifications system used in the SERCUS demonstration. Thus assuming the required operations on classes remain unchanged, the implementation of the procedures can be altered or amended with no effect on the use of classifications in the demonstration.

N.B. All modules using the classifications module will need to be recompiled, this can be done using the procedure \texttt{recompile\_vax}.
From a security point of view, procedures accessing documents and journals must be able to determine the clearance of the user initiating the request. The context, which holds this and other information, is implemented as procedures to add, delete and get entries from a list kept non-local to the procedures. The names of the objects will also have to be unique and unforgable, and must be passed to the procedures as such, if security is to be maintained.

8.1. MODES FOR CONTEXT MODULE

Entry mode is defined as a polymorphic structure of a unique of type mode, and a pointer to something of that mode. Polymorphic types are represented by the form $\exists X. M$.

$$Entry = \exists (X: \text{struct}(\exists X, \text{unique } X))$$

Context mode is defined as a vector of polymorphic entries as follows:

$$Context = \text{vec } (Entry, \text{Posint})$$

The variable context is a vector of polymorphic entries non-local to the procedures. It is created as a zero length vector, which requires a dummy entry (given mode integer for simplicity).

$$\text{Let dummy_key : unique int = MakeUnique : int}$$
$$\text{Let dummy_value : int = Pack Cint } 1$$
$$\text{Let dummy : Entry = ToExists : struct } (Entry, \text{int}) (\text{dummy_value, dummy_key})$$
$$\text{Var context : ref Context := Vec 0 Of dummy }$$

MakeUnique is an operator which generates and delivers a unique value of type Unique type. This value is totally unique since it is different from any other unique of any type and also includes a coding of the machine on which it was generated. If a key is known to have type Unique Y, then since all uniques are different, Unique X = key implies that X = Y, and this is the basis behind the assertion Match which is used throughout the context module. Pack generates and delivers a pointer to an area of memory which has been initialised to contain the integer value 1. ToExists takes the operand i.e. (dummy_value, dummy_key) and delivers it unchanged but with its type generalised to the polymorphic type struct(Entry, X).

8.2. PROCEDURES IN THE CONTEXT MODULE

The context module contains the procedures required to implement the process context in Ten15 notation for the SERCUS demonstration. They are as follows:

8.2.1. Set_context

Set_context is a polymorphic procedure taking a key and its corresponding pointer value and adding them to the context Set_context has the context vector as a non-local variable, and the

1 Define the polymorphic structure the more natural way round with the unique followed by the pointer, will cause the assertion Match to fail.
procedure returns void. If the key already exists then the existing value is overwritten with the new value. A polymorphic procedure can be recognised by a definition of the form Proc Formals X etc. as below:

Proc Formals X (key: unique X, value: ptr X) -> void:

Let set_context = Use (context) In
  Proc Formals X (key: unique X, value: ptr X) -> void:
  Let new_entry = ToExists : struct (Entry, X) (value, key)
  In
  For n To Ub @ context
  Repeat @ again:
    Match @ (@ context VecInd n), key
    [ @ context VecInd n := new_entry
    | Goto @ again ]
  Giving
  Let new : Context = Vec( 1 Of new_entry )
  In
  context := new Concat @ context
  N
  Endfor
Endproc

The key and value are first converted to the correct mode for insertion into the context vector (ie the polymorphic type Entry), using the polymorphic operator ToExists.

A Repeat .. Giving loop is then used to run through the entries in the context vector looking for a match with the procedure key. The match is done using the assertion Match which tests the polymorphic operand to determine whether the unique component of the structure matches the key. If the uniques are equal then their types are also equal and the match is obeyed with the other component of the structure as argument. In this case the matching entry in the context is overwritten with the new entry, and the loop is exited. If the key does not match, the loop is repeated.

If after examination of the whole context vector a match is still not found then the Giving part of the loop construct is executed. This involves forming a new vector of mode Context with one entry ie. new_entry, and then concatenating this new vector onto the existing context vector to give the new context.

8.2.2. Zap-context

Zap-context is a polymorphic procedure which takes a unique key as parameter and finds the corresponding entry in the context, which is then removed. Zap-context has the context vector as a non-local variable, and the procedure returns void. If there is no entry corresponding to the key, then the procedure fails.

Proc Formals X (key : unique X) -> void:

Let zap_context = Use (context, context Last) In
  Proc Formals X (key : unique X) -> void
  For n To Ub @ context
  Repeat @ again:
    Match @ (@ context VecInd n), key
    [ Let below : Context = @ context VecInd n
    | Let above : Context = @ context VecInd n
    | In
    context = below Concat above
  N
  Endfor
Endproc
A Repeat..Giving loop is used to run through the context vector to find the entry with the parameter key. Once this entry has been identified it is removed by trimming the vector entries below this particular entry with the operator Triml, and trimming those above using the operator Trimg. The new context vector is formed by concatenating these two trimmed vectors together, the loop then terminates.

If the key does not match, then the loop is repeated until a match is found. If no matching entry is located then the procedure will fail with the exception being created from the integer context_fail defined in the module FailureNumbers. This exception can be trapped using Trapply in an enclosing procedure.

8.3.1. Generating Uniques

Unique keys are generated using the following piece of Ten15 notation contained in an edfile

```
Define test_key = MakeUnique type()
Keep test_key
```

MakeUnique (type) generates and delivers a unique value of type Unique type.
To actually generate a unique the procedures `compile_ten15` and `run_ten15` must be applied to the edfile as follows:

```
edit compile_ten15\run_ten15
```

### 8.3.2. Context test

Three unique integer keys were generated by compiling and running the above edfile with type defined as integer. An integer value was then associated with each of the unique keys generated. This test procedure takes a void as parameter and returns void.

Keys are set by calling the procedure `set_context` with a unique key and a pointer to the integer value eg.

```
set( key1, Pack value1 )
```

Keys are removed by calling the procedure `zap_context` from the context module with a unique key as parameter, from within `Trapapply` eg.

```
Trapapply( zap, key1
default case
  Output "No Match"
  Op jmp : jmp()
)
```

If `zap_context` is successful then void is returned, if no match is found for the key then the message "No Match" is output.

The value associated with a given key is retrieved using the procedure `get_context` within a `Trapapply`.

```
Trapapply( get, key1
default case
  Output "Not Found"
  Op jmp : jmp()
)
```

If `get_context` is successful the a ptr to the value corresponding to the key is returned. If the procedure fails then the message "Not Found" is output.
9. JOURNALING MODULE

Document journals are used to record the events that happen in the life of a document. User journals are used to record the actions carried out by a particular user e.g., login and logout etc. A document journal is protected from arbitrary access by making it non-local to the read and review procedures and not exporting it from the module. The user's journal is protected from arbitrary access by setting it in the context so allowing only a trusted procedure holding the key to recover it.

9.1. MODES FOR THE JOURNALING MODULE

Both document and user journals are implemented as persistent variables (pvars) containing a linked list of events. Modes required for the implementation of journals are defined in SercusModesandOps and are as follows:

\[
\text{cycle ( Event = struct( String Who, What, Date, choice ptr pers Event Last) )}
\]

\[
\text{Journal = pvar Event}
\]

Event is a structure of three strings, Who, What, Date, and a choice ptr pers Event. The names of the three strings are used as operators to select the fields from the structure e.g.,

\[
\text{Let name : String = Who event}
\]

The choice ptr pers Event is effectively a pointer to the previous entry in the linked list. It is defined as a choice ptr so as to allow the previous event to be null if the end of the list has been reached.

9.2. PROCEDURES IN THE JOURNALING MODULE

The journaling module contains the procedures required to implement the user and document journals in Ten15 notation for the SERCUS demonstration. They are as follows.

9.2.1. Add_Journal

Add_Journal takes as parameters three strings, who, what and date and adds them to the journal (also supplied as a parameter), the procedure returns void. Datastore is supplied to the procedure as a non-local.

\[
\text{Let add_journal = Use (datastore ) In}
\]

\[
\text{Proc (journal : Journal, who, what, date : String ) = void}
\]

\[
\text{Let } 
\text{: pers Event = DePvar journal}
\]

\[
\text{Let c: choice ptr pers Event = ToChoice Pack;}
\]

\[
\text{Let new_event : Event = (who, what, date, c;)}
\]

\[
\text{Let new : pers Event = Persist (new_event, datastore;)}
\]

\[
\text{In}
\]

\[
\text{journal AssPvar new}
\]

Endproc

21
Add_journal retrieves and delivers the pers Event most recently assigned to the pvar re, the last
Event in the journal. This pers Event is turned into a choice ptr pers Event\(^1\) by first applying Pack
which generates and delivers a pointer, followed by ToChoice which converts the operand into the
Corresponding non-null choice. A new event is defined as being a struct of the three parameter
strings and the newly formed choice ptr pers Event. This new event is then stored on the datastore
using the operation Persist, and returning a persistent value which is assigned to the pvar journal
using AssPvar.

9.2.2. Review_jurnal

Review_jurnal takes a journal and displays its contents as an editable vertical (vector of
Strings). Review_jurnal uses the procedure display_event defined as follows:

\[
\begin{align*}
\text{Let blank} & := \text{ToExistsDelSimpleLine}(\text{Pack AsLine" ', del_simple_line_u}) \\
\text{Let display_event} & := \text{Use (blank, del_simple_line_u) In} \\
& \quad \text{Proc (event : Event) \rightarrow DelVec:} \\
& \quad \quad \text{Let whatstr : String = "Event: " Concat What Event} \\
& \quad \quad \text{Let whatstr:......} \\
& \quad \quad \text{Let whatstr:......} \\
& \quad \quad \text{Let whatline = ToExistsDelSimpleLine(Pack AsLine whatstr, del_simple_line_u)} \\
& \quad \quad \text{Let wholine......} \\
& \quad \quad \text{Let wholine......} \\
& \quad \quad \text{Let vencal = AsDelVec Vec(4 Of AsDel whatline) In} \\
& \quad \quad \quad \text{vencal}_0 := \text{wholine}; \\
& \quad \quad \quad \text{vencal}_1 := \text{wholine}; \\
& \quad \quad \quad \text{vencal}_2 := \text{blank}; \\
& \quad \quad \quad \text{vencal}_3 := \text{vencal}; \\
& \quad \quad \text{Nil}; \\
& \text{Endproc}
\end{align*}
\]

Display_event takes an event, extracts the strings using the names as field extraction operators,
and turns them into simple line uniqucs using ToExistDelSimpleLine. A vertical vector is then
formed from the lines using AsDelVec, and returned.

Review_jurnal takes a journal as parameter and returns a Del. It takes as non-locals the
procedure display_event and del_vent_u, a procedure for creating unicques for verticals. It is
defined as follows:

\[
\begin{align*}
\text{Proc review_jurnal = (journal : Journal) \rightarrow Del:} \\
\text{Let review_jurnal = Use (display_event, del_vent_u) In} \\
& \quad \text{Proc (journal : Journal) \rightarrow Del} \\
& \quad \quad \text{Let po : pers Event = DelVar Journal} \\
& \quad \quad \text{Let last : Event = UnPersist po} \\
& \quad \quad \text{Var vencal : ref DelVec := display_event(fast)} \\
& \quad \quad \text{Var next : ref choice ptr pers Event := Last last} \\
& \quad \quad \text{In} \\
& \quad \quad \quad \text{Loop @label :} \\
& \quad \quad \quad \quad \text{NilVod f @next} \\
& \quad \quad \quad \quad @Op p \\
& \quad \quad \quad \text{Let event : Event := UnPersist D p} \\
\end{align*}
\]

\(^1\) A choice pers Event was really what was required here, but supposing the system would not allow this Thus a
ptr pers Event was used instead which could be chosen! When the system is improved, the pointer aspect will be
removed.
First the pers Event most recently assigned to the journal is extracted using the operator DePvar, and unpersisted. This unpersisted event is passed to the display_event procedure, and the next event in the linked list is referenced. The procedure then enters a loop to extract and display the remaining events in the linked list. The loop is constructed using the NotVoid assertion which tests the choice ptr pers Event is the next Event in the list. If this is not null then the operator nonvoid is obeyed with the next Event as argument. The operator takes the ptr pers Event and retrieves the previous Event using the operators Unpersist and D (de_ptrs a pointer). This event is then passed to the display_event procedure, with the DelVec returned being concatenated onto the vertical vector. The loop is repeated until the null choice is reached and the operator null is obeyed with argument void. Finally the vertical vector is converted into a Del using ToEndsDelVec.

9.2.3. Create_Journal

Before the procedures can be tested a journal has to be created. The procedure create_journal takes two strings, one the user’s name and the other the nature of the action, and returns a journal.

```
Let datastore = [datastore : Module(RoPtr, )]
Let root_pset = [root_pset : Module(RoPtr, )]
Let (, date, time) = [data and time : Module(RoPtr, )]

Let create_journal = Use (datastore, root_pset, date, time) In
  Procl (what_str, who_str : String) In
    Let e = choice ptr pers Event = Null : ptr pers Event()
    Let init_event = (who_str, what_str, @ date Concat time(), e)
    Let init_pers : pers Event = Persist (init_event, datastore)
    In
      CreatePvar (root_pset, init_pers)
  Nl
Endproc
```

An initial event describing the creation of a journal is formed using the who and what strings and the date and time from the date_and_time module, with the pointer to the previous event being a null choice of type ptr pers Event generated using the operator Null. This initial event is then persisted to the datastore, and a journal is created using CreatePvar to deliver a new pvar with parent root_pset. Finally journal is initialised to contain the pers Event initial_event.

---

1 The system operator GetTime gives the number of seconds past midnight, but no date. Thus the date is input as a string at the start of the demo.
9.3. TESTING THE JOURNALLING MODULE

The procedure journal_test1 tests both the add_journal and review_journal procedures.

9.3.1. Journal_test1

Journal_test1 is a procedure which takes void and delivers a persist Del.

Events are added to the journal using the add procedure defined in the journalling module, and reviewed using the review procedure. Review returns a Del, but since the test procedure requires that a persist Del is returned, the result of review is persisted to the datastore.
10. LOGIN AND RELATED PROCEDURES

The following procedures allow a user to login to the Ten15 editor. Once in the editor, the system can be asked to answer questions and perform/simulate operations, these form the basis of the SERCUS demonstration. The questions include Who am I? (who), What's my clearance? (what), and Am I on the trusted path? (where), while the operations include Review my journal (my_journal), Put me on the trusted path (on.to.tp) and Take me off the trusted path (off.tp). All the above are written as ions. An ion is very similar to a procedure except that some of the non-locals are not given values in the construct, only a specification, with the actual values being supplied at a later time. These specifications have the same form as parameters.

10.1. MODES REQUIRED FOR THE LOGIN RELATED PROCEDURES

The following modes are defined for use in the login related ions and procedures:

User mode is defined as a structure of two strings representing the user’s id and password and a Class corresponding to the user’s clearance.

User = struct( Strmg Uid Password, Class Clearance )

ValidUsers is defined as a vector of entries of mode User as follows:

ValidUsers = vec( User, Posint )

UserDetails is a structure of a string representing the user’s id, a Class representing the user’s clearance and a Journal representing the user’s journal.

UserDetails = struct( Strig Uid, Class Uid_Clear, Journal Uid_Journal )

N.B. The modes User and UserDetails will eventually be expanded to include a user’s cupboard and mail box.

The TpFlag allows the user to determine whether they are on or off the trusted path. However, the actions of going on to, or off the trusted path are only simulated at present. The TpFlag is defined as a bool value as follows:

TpFlag = bool
10.2. IONS AND PROCEDURES RELATED TO LOGIN

10.2.1. Login Ion

This module declares an ion for login which when closed with the unique values for user_details and trusted path flag, the valid_users vector and journal becomes a procedure of type void to void.

\[
\text{Let (set.)} = \text{context : Module(RoPr...)} \\
\text{Let (add.)} = \text{[journal : Module(RoPr...)} \\
\text{Let (date.time) = date_and_time} \\
\text{Let stringlength : PostInt = 10} \\
\text{In} \\
\text{Ro Pack Use (stringlength, set, add, date, time) In} \\
\text{ion (user_details : unique UserDetails) \\
(tflag : unique ToFlag) \\
(valid_users : ValidUsers) \\
(journal : Journal) \\
Void -> Void:} \\
\text{Loop @next:} \\
\text{Output "Please Enter Your Name";} \\
\text{Let uid : String = Input Add1; } \\
\text{Let invoc2 = (Output "Please Enter Your Password", Vec(stinglength Of Char") )} \\
\text{Let password : String = Input Invoc2; } \\
\text{In} \\
\text{Forall user In valid_users} \\
\text{Repeat @again:} \\
\text{if (Uid @ user VecEq uid) And (Password @ user VecEq password)} \\
\text{Then} \\
\text{Let ed = (Ch, ed : Module(RoPr...))} \\
\text{Let clearance : Class = Clearance @ user} \\
\text{Let details : UserDetails = (uid, clearance, journal)} \\
\text{Let flag : TFlag = True} \\
\text{In} \\
\text{add(journal, uid, "Logged in", @ date Concat time());} \\
\text{set(user_details, Pack details);} \\
\text{set(tflag, Pack flag);} \\
\text{Trapply(ed, 0);} \\
\text{if Failure InToFlag 11} \\
\text{Output "Bye Bye " Concat uid; } \\
\text{add(journal, uid, "Logged out", @ date Concat time());} \\
\text{Goto @next} \\
\text{else} \\
\text{Goto @again} \\
\text{Fi} \\
\text{Giving} \\
\text{Output "Authorization Failure"} \\
\text{End} \\
\text{Fi} \\
\text{Endloop} \\
\text{Endon} \\
\]

Login is essentially a loop, which prompts for a user name followed by a password (The name or the password is restricted to ten characters) The vector of valid_users is then examined for a match on the user name and the password. If the match is successful, then the users ID, the event...
"Logged in", and the date and time are added to the user's journal. Additionally the user_details (uid, clearance and journal) and tpflag (set to true) are set in the context.

Login then calls the editor using the operator Trapply. Trapply calls the editing procedure edit with the required void parameter, and tests the exit status of the procedure call. If the procedure completes normally then Failure IntToTrap 11 is obeyed. (This will never happen in practice.) It should also be noted that the only way to leave the editor is to make it fail i.e. the editor is successfully exited when it is allowed to fail. If the procedure fails (the only case that will apply with edit) then the logout message is output, the appropriate "Logged out" message and date and time are added to the user's journal, and the procedure again prompts for a user name to be input. The third alternative in the Trapply is for a long jump out of the procedure, but as this will never happen it can be ignored in the documentation.

If the user name was not found in the vector of valid_users, or the passwords did not match then the procedure outputs an authentication failure message and dies. This is sufficient for the moment, but eventually it should prompt for a new user name after an authentication failure instead of requiring that the login procedure is called again.

10.2.2. Login Procedure

The ion for login can be closed with the following non-local values, the unique for user details, the unique for the trusted path flag, the vector of valid users, and the user's journal. Once each ion has been closed, login becomes a procedure of type void to void i.e.

\[
\text{proc login} = () \rightarrow ()
\]

Two unicodes are generated as described previously, unique_ud is a unique for the UserDetails, and unique_tp is a unique for the TpFlag.

Let unique_ud : unique UserDetails = \[ A : \text{Unique String} \]

Let unique_tp : unique TpFlag = \[ A : \text{Unique Bool} \]

A journal is also created using the procedure create_journal.

Let journal : Journal = create_journal("Journal Created", "Administrator")

Sets of user details are formed as a structure of the three strings representing user ID, password and clearance (the procedure class_to_str is used to convert a Class into a String). A vector of valid users is formed from the sets of user details.

The login ion is then closed as follows to give a procedure taking void and returning void

\[
\begin{align*}
\text{let closed1 : ion( unique TpFlag, ion( ValidUsers, ion( Journal, Void -> Void )))} \\
& = \text{unique_ud Close login on} \\
\text{let closed2 : ion( ValidUsers, ion( Journal, Void -> Void ))} = \text{unique_tp Close closed1} \\
\text{let closed3 : ion( Journal, Void -> Void )} = \text{valid_users Close closed2} \\
\text{let login = Void -> Void = journal Close closed3}
\end{align*}
\]
10.2.3. Who Am I Ion

When closed to form a procedure, this ion will determine who is logged in to the system.

```hs
Let (get) = \{context : Module(RoPtr..)\}
Let (....fail_proc) = \{FailureNumbers : Module(RoPtr..)\}
Let bybye = \{bybye : Module(RoPtr..)\}
Let message = \{Message : Module(RoPtr..)\}

In
Ro Pack Use (get, fail_proc, bybye, message) In
ion (user_details : unique UserDetails)
Void => String:
  Trappy( get, user_details
    Op ptr_details:
    let details : UserDetails = D ptr_details
    let me : String = ud_user details
    In
    me
    \!
    Op trap
    let mess = fail_proc(trap)
    In
    message("msg Conc \": Failed No User-details in Context\"");
    Failure into Trap bybye
    N\!
    Op jump : jump()
  
End ion
```

WhoAmI ion is based on the operator Trappy. Get_context is used to extract the user_details from the context. If successful, get returns a pointer to the user_details which is de-pired using D to get the user_details. The users User ID is then extracted from the structure and returned.

If get fails to find the user_details in the context then the procedure will fail using the operator Failure which forms an exception from the trap value created from the integer context_fail defined in the FailureNumbers module.

The other ions described in this section are generally constructed in a manner very similar to that of the WhoAmI ion, using the operator Trappy.

10.2.4. Who Am I Procedure

The WhoAmI ion is closed with the unique for user details to give a procedure that takes void and returns a string.

```hs
Ion( unique UserDetails, Void => String )
```

N.B. The same unique generated for the unique user details in login must be used to close the WhoAmI Ion

```hs
Let whoami : Void => String = unique ud Close whoami_ion
```
10.2.5. What's My Clearance Ion

When closed this ion will allow the clearance of the user to be determined.

```
Let (...class_to_str) = [Classifications : Module(RoPtr...)]
Let (...get) = [Context : Module(RoPtr...)]
Let (...fail_proc) = [FailureNumbers : Module(RoPtr...)]
Let byebye = [byebye : Module(RoPtr...)]
Let message = [Message : Module(RoPtr...)]

In
Ro Pack Use (get, class_to_str, fail_proc, byebye, message) In
Ion (user_details : unique UserDetails)
Void -> String:
  Trapply (get, user_details
    | Op ptr_details:
    | Let details : UserDetails = D ptr_details
    | Let clear : Class = Ud_Clear details
    | In
    | class_to_str (clear)
    | N
    | Op trap:
    | Let mess = fail_proc (trap)
    | In
    | message (mess Concat " : Panel\'l No User Details in Context")
    | Failure InfoToTrap byebye
    | N
    | Op jump : jump()
  )
Endion
```

What's My Clearance extracts the user_details from the context in the same way as the WhoAmI ion. The clearance is converted from a class into a string using the procedure class_to_str from the classifications module. If no user details are found in the context, then the procedure will fail.

10.2.6. What's My Clearance Procedure

The What's My Clearance ion is closed with the unique for user details to give a procedure which takes a void and returns a string:

```
ion (unique UserDetails, Void -> String)
```

N.B. The unique for user details generated in login must be used to close the ion as follows:

```
Let myclear : Void -> String = unique_ud Close myclear_ion
```
10.2.7. Am I On The Trusted Path Ion

This ion will determine whether the user is on the trusted path when it is closed to form a procedure.

\[
\text{Let} \quad (\ldots, \text{get}) = \text{context} : \text{Module}(\text{RoPtr}),
\]
\[
\text{Let} \quad (\ldots, \text{fail} \_\text{proc}) = \text{FailureNumbers} : \text{Module}(\text{RoPtr}),
\]
\[
\text{Let} \quad \text{byebye} = \text{bybye} : \text{Module}(\text{RoPtr}),
\]
\[
\text{Let} \quad \text{message} = \text{Message} : \text{Module}(\text{RoPtr}).
\]

In
\[\text{Ro Pack Uso. (get, fail\_proc, byebye, message) In}
\]
\[\text{Now (tpflag : unique TpFlag)}
\]
\[\text{Void \rightarrow bool;}
\]

Trably: \(\text{get, tpflag}
\]
\[\quad | \text{Op ptr\_flag;}
\]
\[\quad \text{let flag} : \text{TpFlag} = \text{D ptr\_flag}
\]
\[\quad \text{In}
\]
\[\quad \text{flag}
\]
\[\quad \text{N}
\]
\[\quad | \text{Op trap;}
\]
\[\quad \text{let mass} = \text{fail\_proc} (\text{trap})
\]
\[\quad \text{In}
\]
\[\quad \text{message} = \text{Concat}: \text{failure intToTrap byebye}
\]
\[\quad \text{N}
\]
\[\quad | \text{Op jump : jump()}
\]

Endon

\(\text{AmIOnTheTrustedPath}\) extracts the tpflag from the context using the procedure get\_context, in a way similar to the WhoAmI ion. If get\_context succeeds then a pointer is returned which is de- picted to return the value of the flag. If there is no trusted path flag in the context then the procedure fails in the usual way.

10.2.8. Am I On The Trusted Path Procedure

The \(\text{AmIOnTheTrustedPath}\) ion is closed with the unique for the trusted path flag to give a procedure which takes void and returns a bool.

\[\text{ion (unique TpFlag, Void \rightarrow bool)}
\]

The same unique for the trusted path flag as generated in the login procedure is used to close the ion as follows:

\[\text{tpath : Void \rightarrow Bool = unique to \text{Close tpath\_ion}}\]

\(\text{tpath}\) is then used as a non-local in a procedure which takes a void and returns a string. If the tpath procedure returns true then the string "You are on the Trusted Path" is returned. If tpath returns false then the string "You are NOT on the Trusted Path" is returned.
10.2.9. On_toTp and Off_toTp Ions

These ions, when closed to form procedures, will simulate the action of the user going on to or off
the trusted path.

Let (set, get) = \{context : Module(RoPtr, )\}
Let (......, fail_proc) = \{FailureNumbers : Module(RoPtr, )\}
Let byebye = \{byebye : Module(RoPtr, )\}
Let message = \{Message : Module(RoPtr, )\}

In
Ro Pack Use (set, get, fail_proc, byebye, message ) In
ion (tflag : unique TpFlag )
Void \rightarrow Void;
Trap( set, tflag)
| Op ptr_flag;
E(Not) ( O ptr_flag )
Then
set( tflag, Pack False(True))
F1
| Op trap;
Let mess = failproc( trap )
in
message( mess Concat " : Pencil No Trusted Path Flag in Context" ),
Failure in To Trap byebye
N
| Op jump : jump()
Endon

The procedure get_context is used within the operator Trap( set) to extract the trusted path flag from
the context. If get is successful, then a pointer to the flag is returned. This is de-pointed, and the flag
is re-set, as appropriate in the context. If the trusted path flag cannot be extracted from the context
then the procedure fails.

10.2.10. On_toTp and Off_toTp Procedures

The ions are closed with the unique for the trusted path flag to give procedures which take void and
return void.

ion( uniqueTpFlag, Void \rightarrow Void )

The same unique for the trusted path flag as generated in the login procedure must be used to close
the ion eg

Let onpath : Void \rightarrow Void = unique_tpclose onpath on
This ion, when closed to form a procedure, will allow the user to examine their journal as long as they are on the trusted path.

```
Let (.get) = context : Module(RoPtr..)
Let (review) = [Journal: Module(RoPtr..)
Let (...flag, fail, fail_proc) = [FailureNumbers: Module(RoPtr..)
Let byebye = byebye : Module(RoPtr..)
Let message = [Message: Module(RoPtr..)

In Ro Pack Use (review, get, flag, fail, fail_proc, byebye, message) In ion (user_details: unique UserDetails)
(tflag: unique TpFlag)
Void -> Del:
  Trapply(get, user_details
  | Op ptr_details:
  Trapply(get, tflag
  | Op ptr_flag:
  # D ptr_flag
  Then
  Let details : UserDetails = D ptr_details
  Let journal; Journal = Ud_Journal; details
  In review(journal)
  N
  Else
  Failure Into Trap flag,fail
  FI
  | Op trap : Failure trap
  | (Op jump;jump)
| ]
  | Op trap:
  message(fail_proc(trap1 Concat "- " PanclII No User Details in Context") );
  Failure Into Trap byebye
  | Op jump : jump)
  ]
Endon
N
```

ReviewMyJournal uses the procedure get_context within the operator Trapply to retrieve the users details from the context. Get_context is then used again within Trapply to retrieve the trusted path flag from the context. Assuming the trusted path flag is true, then the user_details are de-pitted and the journal field is extracted from the structure. The review journal procedure is then applied to the journal, returning a Del. If get_context fails i.e. there are no user details or no trusted path flag in the context then the procedure ReviewMyJournal will also fail.

10.2.12. Review My Journal Procedure

The ReviewMyJournal ion is closed with the unique for user details and the unique for the trusted path flag to give a procedure that takes void and returns a Del

```
ion( unique UserDetails, ion( unique TpFlag, Void -> Del ) )
```

The same uniques generated for the user details and the trusted path in login are used to close the ion as follows:

32
Let closed : len( unique TpFlag, Void → Del ) = unique_tcl Close reviewmyjournal_on
Let reviewmyjournal : Void → Del = unique_tp Close closed

Since the procedure is actually required to return Text, which is a persist Del, reviewmyjournal is passed as a non-local to a procedure which calls Persist on the result of the reviewmyjournal procedure call so returning Text.
11. REGISTRY MODULE

The registry contains all the documents held in the system, with the CDR number of a document being its position in the registry. The registry will be supplied to the registry procedures as a non-local variable. Documents are only accessible via the procedure to supply their contents, and the clearance of the user issuing the request must dominate the classification of the document.

11.1. MODES FOR REGISTRY MODULE

The modes used in the registry procedures are as follows:

The Document mode is defined as a struct of three fields, the classification of the document, the contents of the document and the document's Journal, as follows:

Document = struct(Class Class, Del Contents, Journal Journal)

The registry is defined as a vector of Document entries as follows:

Registry = vec(Document, PosInt)

11.2. PROCEDURES IN THE REGISTRY MODULE

The procedures in the registry module are written as ions, they are read_document, review_document_journal, document_class and create_document.

11.2.1. Read_Document

Read_document is an ion which can be closed with the unique for user details, the registry, and the trusted path flag to give a procedure which takes the cdr number (where the cdr number of a document is simply its position in the registry), and returns the text of the document.

Let (dom,...) = classifications : Module(RoPtr.)
Let (...get) = Context : Module(RoPtr.)
Let (add,) = Journal : Module(RoPtr.)
Let (date, time) = date_and_time : Module(RoPtr.)
Let (..chars) = InitialChars : Module(RoPtr.)
Let (...,cdt, clear, fail) = FailureNumbers , Module(RoPtr.)
Let datastore = Datastore : Module(RoPtr.)

Let read_document = Use( dom, get, add, date, time, initial_chars, datastore, cdr, fail, clear, fail) in

Ivon(user_details: unique User Details)

(registry: Registry)
(tflag: unique TplFlag)
(cdr_num: Int) -> Text

Try{ get, user_details
| Op ptr_details
| Op ptr_flags
| Try{ get, tflags
| Op ptr flag;
| if((cdr_num < 1) Or (cdr_num > Upb @ registry))
| Then
| Failure InitToTrap cdr fail

34
Else

Let user = Ud_uid (D ptr_details)
Let user_clear = Ud_Clear (D ptr_details)
Let u_jnl = Ud_journal (D ptr_details)

Let document = @ (@ registry Veclid cdr_num)
Let doc_clear = Class document
Let d_jnl = Journal document

Let date_time = @ date Concact time()
In
If dom (user_clear, doc_clear)
Then
Let tp_str = "Opened Document " Concat "CDR"
Concat intcharz (cdr_num)
Let oftp = tp_str Concat "Off TP"
In
If D ptr_flag
Then
add(u_jnl, user, tp_str, date_time)
add(d_jnl, user, tp_str, date_time)
Else
add(u_jnl, user, oftp, date_time)
add(d_jnl, user, oftp, date_time)
End
Persist (Contents document, datastore)
Else
Let mess = "Prevented from Opening Document CDI"
Concat intcharz (cdr_num)
In
add (u_jnl, user, mess, date_time)
No
Failure InfoToTrap clear fail
Fl
| Op trap : Failure trap
| Op jump : jump()
|
| Op trap : Failure trap
| Op jump : jump()
|
Endon

Get _context is used within the operator Traplly to extract the user details from the context, returning a pointer to the details. Get _context is then used again within Traplly to extract the trusted path flag, returning a pointer to it. If the user details and trusted path flag have been successfully extracted, then the cdr number is checked to ensure that it is within range i.e. greater than or equal to one and less than or equal to Upb registry. If either of the get_context procedure calls fail then the read_document procedure will also fail using the Failure operator.

Assuming the cdr number is in range it is used to extract the document from the registry vector. If it is not in range, then the procedure will fail. A check is then done to ensure that the clearance of the user trying to read the document is greater than the clearance of the document itself. This is done using the dominates procedure from the classifications module. If the users clearance dominates the document clearance and the user is on the trusted path, then the event "Opened Document cdr_num" is added to the users journal along with the date and time, and the same event is added to the document journal. If the user is not on the trusted path, then "Off TP" is concatenated onto the event string before adding it to the user and document journals. The contents of the document are then returned. If the users clearance does not dominate the classification of the document then the event "Prevented from Opening Document cdr_num" is added to the users journal, along with the date and time.

35
11.2.2. Review_Document_Journal

Review_doc_journal is an ion which can be closed with the unique for user details, the registry, and the trusted path flag to give a procedure which takes the cdr number of a document and returns the text of the journal.

```
Let (.get) = [context : Module(RoPtr..)]
Let (review) = [Journal : Module(RoPtr..)]
Let (...flag fail, cdr fail...) = [FailureNumbers : Module(RoPtr..)]
Let datastore = [datastore : Module(RoPtr..)]

Let review_doc_journal = Use (get, review, flag fail, cdr fail, datastore) In
  Ion (registry : Registry)
  (tflag : unique_TpFlag)
  (cdr_num : Int) -> Text:
  Trappy[get, tflag,]
    If D ptr_flag;
      Then
        If (((cdr_num < 1) Or (cdr_num > Upb registry))
          Then
            Failure IntToTrap cdr_fail
        Else
          Let document = @ (@ registry Vectors cdr_num )
          Let doc_jnl = Journal document
          In
            Persist { review(doc_jnl), datastore }
      Else
        Failure IntToTrap flag_fail
    FI
    Else
      Failure IntToTrap flag_fail
  FI
  [Op trap : Failure trap]
  [Op jump : jump()]
End
```

Get context is used within the operator Trappy to extract the user details from the context. If this is successful then a pointer to the user details is returned otherwise the procedure fails. The pointer to the flag is determined using D, and if it is set to true then the cdr number of the document is checked to ensure that it is within the range of the registry. If the trusted path flag is false, or the cdr number is out of range, then the procedure fails using the operator Failure.

Assuming all the checks have been successful, the document is extracted from the registry using vector indexing on the cdr number. The journal is then extracted from the Document structure using field extraction, and the procedure review_journal is applied to review the journal. Review_journal returns a Del, but since the procedure is required to return Text, the result of the review_journal procedure call is persisted to the datastore.
11.2.3. Document_Class

Doc_class is an ion which can be closed with the registry to give a procedure which takes the cdr number of a document and returns its classification as a string.

```
Let (...) = classifications : Module(RegPtr,)
Let (...) = FailureNumbers : Module(RegPtr,)

Let doc_class = Use (...) in
  I on (registry : Registry)
  (cdr_num : Int) -> String;

If ((cdr_num < 1) Or (cdr_num > Upos @ registry))
  Then Failure IntToTraps cdr_fail
  Else
    Let document = @ ()(registry Vecdir cdr_num)
    Let class : Class = Class document
    If "Document classification" = "Concat class_to_str(class)
    Then
      Fail
    End
End
```

The cdr number of the document is checked to ensure it is within the range of the registry, and if it is not then the procedure fails. The document is then extracted from the registry using vector indexing with the cdr number, and the classification of the document is extracted from the document structure. The procedure class_to_str from the classifications module is then used to convert the class to a string which is returned with a suitable message.

11.2.4. Create_Document

Create_document is an ion which can be closed with the unique for user details, the registry, and the trusted path flag to give a procedure which takes some text and a classification string and returns void.

```
Let (...) = context : Module(RegPtr,)
Let (...) = Journal : Module(RegPtr,)
Let (...) = classes : Module(RegPtr,)
Let create_journal = create_journal : Module(RegPtr,)
Let (...) = jflag : Module(RegPtr,)
Let (...) = (date, time : Module(RegPtr,)

Let create_document = Use (...) in
  Ion (user_details : Unique UserDetails)
  (jflag : unique ToFlag)
  (registry : Registry)
  (text : Text, class : String) -> Void

Traps[get, user_details 
  | Op ptr_details
  get, jflag
  | Op ptr_flag
  get, ptr_flag
Then
  Let user = Uo wd (D ptr_details)
  Let what = "Document Journal Created" * Concat class
  Let d_jnrd = create_journal(what, user)
```
Let class = str_to_class(class)
Let text = Del.unPersist text
Let doc = Document = \{class, text, d_jnl\}
Var new : Registry := Vec(1 Of doc)
In
registry := @ registry Concat @ new
Else
Let user = Ud_id (D_ptr_details)
Let u_jnl = Ud Journal (D_ptr_details)
Let date_time := @ date Concat time()
Let mess := "Attempt to Create a Document : Not on TP"
In
add(u_jnl, user, mess, date_time)
N:
Failure InToTrap flag fail
Fi
[Op trap : Failure trap
[Op jump : jump()
]
[Op trap : Failure trap
[Op jump : jump()
]
Endon

Get_context is used within the operator Trapply to extract the user details from the context, returning a pointer to the details. Get_context is then used again within Trapply to extract the trusted path flag, returning a pointer to it. If either of the calls of get_context are unsuccessful, then the procedure will fail. If the trusted path flag is set to true, then the user's id is extracted from the details and a journal is created using the procedure create_journal with the user's id and the string "Document Journal Created : " Concat class (where class is the result of applying str_to_class to the class string given as parameter to the create_document procedure) as parameters.

The text supplied to the procedure is unpersisted to give a Del, and a new document is formed using the class, the text, and the newly created document journal. This new document is then used to form a new entry for the registry, which is then concatenated on to the end of the existing registry vector.

If the user is not on the trusted path, then the string "Attempt to Create a Document : Not on TP" is added to the user's journal along with the date and time, then the procedure fails.
11.2.5. Registry procedures

Each of the above four ions are closed within the module registry procedures, and the failures are trapped.

Firstly the variable registry is created as a zero length vector with dummy entries as follows:

```haskell
Let (del_simple_kn_u) = [line uniques : Module(RoPtr, ...)
Let (del_vnt_u) = [vert uniques : Module(RoPtr, ...)
Let some_text = Use (del_simple_kn_u, del_vnt_u)
Proc (message : String) -> Del:
Let t = ToExistsDelSimpleLine (Pack AsLine message, del_simple_kn_u)
Let vncal = AsDelVec Vec 1 Of AsDel t
Let va = ToExistsDelVec (Pack vncal, del_vnt_u)
In
N va
Endproc

Let dummy_journal : Journal = create_journal("dummy", "dummy")
Let dummy_doc : Document = (OrClass 1, some_text("dummy"), dummy_journal)
Var registry : Registry = Vec(0 Of dummy_doc)
```

Each of the four ions are then closed with the appropriate uniques and the registry as follows:

```haskell
Let unique_to : unique TpFlag = [A : Unique Bool]
Let unique_ud : Unique User Details = [A : Unique (...]
Let closed1 : Ion (Registry ion (unique TpFlag, Int -> Text)) = unique_ud Close read_doc ion
Let closed2 : Ion (unique TpFlag, Int -> Text) = registry Close closed1
Let read_doc : Int -> Text = unique_to Close closed2
Let rd_closed : Ion (unique TpFlag, Int -> Text) = registry Close review_doc journal ion
Let review_doc : Int -> Text = unique_to Close rd_closed
Let doc_class : Int -> String = registry Close doc_class ion
Let cd_closed1 : Ion (unique TpFlag, Ion (Registry, struct (Text, String) -> Vod)) = unique_ud Close create_doc ion
Let cd_closed2 : Ion (Registry, struct (Text, String) -> Vod) = unique_to Close cd_closed1
Let create_doc : struct (Text, String) -> Vod = registry Close cd_closed2
```

N.B. The unique for the user details and the unique for the trusted path flag are identical to those created in the login procedure. In fact throughout the whole SERCUS demonstration, wherever these values are required, those created in login are used.

Having closed the four ions, procedures are defined for each ion which catch and deal with exceptions caught in the ions and propagated. The general form of these procedures is the operator Triapply which returns the result of the procedure closed from the relevant ion if the procedure was successful, or a VnuMessage diagnosing the error if the procedure closed from the ion failed, followed by a fatal Failure. eg.
Let $\text{Message} = \text{message : Module(RoPtr...)}$

Let $\text{read} = \text{Use (read_doc, fail_proc, byebye) In}$
$.~\text{Proc (cdr_num : Int) \Rightarrow Text:}$
$.~\text{Trapply(read_doc, cdr_num)}$
$.~\text{Op text : text}$
$.~\text{Op trap.}$
$.~\text{Let message = fail_proc(trap)}$
$.~\text{In}$
$.~\text{Message(message);}$
$.~\text{Failure intToTrap byebye}$
$.~\text{N}$
$.~\text{Op jump; jump()}$

Endproc
12. REFERENCES

[1] (Harrold 90)  An Example Secure System Specified Using the Terry-Wiseman Approach
C. L. Harrold
RSRE Report 90011, July 1990

[2] (Goodenough & Rees 89)  A Notation for Ten15
K. H. Goodenough, S. J. Rees
May 1989
**Title**

TRIAL IMPLEMENTATION OF A SECURE APPLICATION USING Ten15

**Authors**

BILSBY, E R

**Abstract**

This report describes how a subset of SERCUS has been implemented using Ten15. SERCUS is a research implementation of a multi-level secure workstation based on the SMITE approach and running a classified document handling application. SMITE is an approach to the construction of secure systems which uses strong typing. Ten15 is an algebraically defined, strongly typed abstract machine running on a VAX station.

This work was performed while the author was a Vacation Student at RSRE and used the Ten15 Cross Compilation System as it existed in Summer 1990.