

INTERNATIONAL STANDARD

ISO/IEC
13719-2

First edition
1995-06-01

Information technology — Portable Common Tool Environment (PCTE) —

Part 2:

C programming language binding

*Technologies de l'information — Environnement d'outil courant portable
(PCTE) —*

Partie 2: Liant de langage de programmation C



Reference number
ISO/IEC 13719-2:1995
AVA Documents
Document Sharing Hub

Contents

	page
1 Scope	1
2 Conformance	1
3 Normative references	1
4 Definitions	2
5 Formal notations	2
6 Outline of the standard	2
7 Binding strategy	3
7.1 C Programming Language Standard	3
7.2 General Principles	3
7.3 Sets and Sequences	3
7.4 Character Strings	4
7.5 Memory Allocation	4
7.6 References and Names	5
7.7 Operation Return Values	5
7.8 Error Conditions	5
7.9 Identifiers	5
7.10 Implementation Limits.	6
8 Datatype mapping	6
8.1 Mapping of PCTE Datatypes to LI Datatypes	6

8.1.1	Mapping of Predefined PCTE Datatypes	6
8.1.2	Mapping of Private PCTE Datatypes	8
8.1.3	Mapping of Complex PCTE Datatypes	8
8.1.4	New LI Datatype Generators	8
8.2	Mapping of LI Datatypes to C Datatypes	10
8.2.1	LI Datatype: boolean	10
8.2.2	LI Datatype: pcte-integer	11
8.2.3	LI Datatype: pcte-natural	12
8.2.4	LI Datatype: pcte-float	13
8.2.5	LI Datatype: pcte-time	14
8.2.6	LI Datatype: pcte-text	15
8.2.7	LI Datatype: octet	16
8.2.8	LI Enumerated Datatype: pcte-xxx	16
8.2.9	LI Private Datatypes	16
8.2.10	LI Datatype Generator: pcte-sequence	17
8.2.11	LI Datatype: pcte-string	18
8.2.12	LI Datatype Generator: bounded-set	20
8.2.13	LI Datatype Generator: choice	21
8.2.14	LI Datatype: record	22
8.3	Private Datatypes	23
8.4	References and Names	23
8.5	C Private Type: Pcte_sequence	24
8.5.1	Operations on Sequences	27
8.5.2	Error Conditions for Sequence Operators	31
8.6	Deriving C Function Semantics From Abstract Specification.	32
8.7	Headers	32
8.7.1	The Global Pcte Header	33
8.7.2	The Pcte Basic Type Header	33
8.7.3	The Pcte Sequence Header	34
9	Object management	38
9.1	Object Management Datatypes	38
9.2	Link Operations	41
9.3	Object Operations	46
9.4	Version Operations	53
10	Schema management	55
10.1	Schema Management Datatypes	55
10.2	Update Operations	57
10.3	Usage Operations	65

10.4	Working Schema Operations	68
11	Volumes, devices and archives	72
11.1	Volume, Device, and Archive Datatypes	72
11.2	Volume, Device and Archive Operations	72
12	Files, pipes and devices	75
12.1	File, Pipe and Device Datatypes	75
12.2	File, Pipe and Device Operations	75
13	Process execution	78
13.1	Process Execution Datatypes	78
13.2	Process Execution Operations	79
13.3	Security Operations	83
13.4	Profiling Operations	84
13.5	Monitoring Operations	85
14	Message queues	86
14.1	Message Queue Datatypes	86
14.2	Message Queue Operations	87
15	Notification	89
15.1	Notification Datatypes	90
15.2	Notification Operations	90
16	Concurrency and integrity control	90
16.1	Concurrency and Integrity Control Datatypes	91
16.2	Concurrency and Integrity Control Operations	91
17	Replication	92
17.1	Replication Datatypes	92
17.2	Replication Operations	92
18	Network connection	94
18.1	Network Connection Datatypes	94
18.2	Network Connection Operations	95
18.3	Foreign System Operations	97
18.4	Time Operations	97

19	Discretionary security	97
19.1	Discretionary Security Datatypes	98
19.2	Discretionary Access Control Operations	100
19.3	Discretionary Security Administration Operations	100
20	Mandatory security	102
20.1	Mandatory Security Datatypes	102
20.2	Mandatory Security Operations	102
20.3	Mandatory Security Administration Operations	104
20.4	Mandatory Security Operations for Processes	105
21	Auditing	106
21.1	Auditing Datatypes	106
21.2	Auditing Operations	110
22	Accounting	112
22.1	Accounting Datatypes	112
22.2	Accounting Administration Operations	114
22.3	Consumer Identity Operations	116
23	References	116
23.1	Reference Datatypes	116
23.2	Object Reference Operations	117
23.3	Link Reference Operations	119
23.4	Type Reference Operations	121
24	Implementation limits	122
24.1	Implementation Limit Datatypes	122
24.2	Implementation Limit Operations	124
25	Error conditions	124
25.1	Error Condition Datatypes	124
25.2	Error Condition Operations	132
	Index of abstract operations	133
	Index of c subprograms	141
	Index of c datatypes	149

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 13719-2 was prepared by the European Computer Manufacturers Association (ECMA) (as Standard ECMA-158) and was adopted, under a special "fast-track procedure", by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by national bodies of ISO and IEC.

ISO/IEC 13719 consists of the following parts, under the general title *Information technology — Portable Common Tool Environment (PCTE)*:

- Part 1: Abstract specification
- Part 2: C programming language binding
- Part 3: Ada programming language binding

Information technology — Portable Common Tool Environment (PCTE) —

Part 2: C programming language binding

1 Scope

This part of ISO/IEC 13719 defines the standard binding of the Portable Common Tool Environment (PCTE), as specified in ISO/IEC 13719-1, to the C Programming Language.

A number of features are not completely defined in ISO/IEC 13719-1, some freedom being allowed to the implementor. Some of these features are specified as implementation limits. Some constraints are placed on these implementation limits by this C Language Binding Standard. These constraints are specified in clause 24, Implementation Limits.

PCTE is an interface to a set of facilities that forms the basis for constructing environments supporting systems engineering projects. These facilities are designed particularly to provide an infrastructure for programs which may be part of such environments. Such programs, which are used as aids to system development, are often referred to as tools.

2 Conformance

An implementation of PCTE conforms to this part of ISO/IEC 13719 if it conforms to 2.2 of ISO/IEC 13719-1, where the binding referred to there is taken to be the C Language Binding defined in clauses 1 to 5 and 8 to 25 of this part of ISO/IEC 13719. All other clauses in this part of ISO/IEC 13719 are provided as assistance to the reader and are not normative.

The C Language Binding defined in this part of ISO/IEC 13719 conforms to 2.1 of ISO/IEC 13719-1.

3 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 13719. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO/IEC 13719 are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO 8601:1988,	<i>Data elements and interchange formats — Information interchange — Representation of dates and times.</i>
ISO 8859-1:1987,	<i>Information processing — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1.</i>
ISO 9899:1990,	<i>Programming languages — C.</i>
ISO/IEC TR 10182:1993,	<i>Information technology — Programming languages, their environments and system software interfaces — Guidelines for language bindings.</i>
ISO/IEC 11404:— ¹⁾ ,	<i>Information technology — Programming languages their environments and system software interfaces — Language-independent datatypes.</i>
ISO/IEC 13719-1:1995,	<i>Information technology — Portable Common Tool Environment (PCTE) — Part 1: Abstract specification.</i>

4 Definitions

Definitions used in this part of ISO/IEC 13719 are those defined in ISO/IEC 13719-1.

5 Formal notations

Two formal notations are used in this part of ISO/IEC 13719. For each bound operation, the abstract specification is given using a subset of the Vienna Development Method Specification Language (or VDM-SL), exactly as it is specified in ISO/IEC 13719-1. For the C Language binding for each operation, the function prototype syntax is used as defined in ISO 9899.

6 Outline of the standard

Clause 7 describes the strategy used to develop this binding specification.

Clause 8 contains the mapping from the datatypes that are used in the Abstract Specification to C Language datatypes.

Clauses 9 to 22 define the binding of datatypes and operations in the corresponding clauses of ISO/IEC 13719-1.

Clause 23 defines the binding of object, attribute, link, and type references, as specified in 23.1.2 and 23.2 of ISO/IEC 13719-1.

Clause 24 defines the binding of the implementation limit functions described in ISO/IEC 13719-1, clause 24.

Clause 25 defines the binding of the error conditions specified in ISO/IEC 13719-1, appendix C, and defines binding-defined error conditions for the C Binding.

1) To be published

7 Binding strategy

7.1 C Programming Language Standard

This part of ISO/IEC 13719 is a conforming program according to ISO 9899.

7.2 General Principles

The following general principles were applied when generating the binding in this part of ISO/IEC 13719.

ISO/IEC TR 10182 should be followed as far as possible for binding method 1: provide a completely defined procedural interface.

Each operation in ISO/IEC 13719-1 should be represented by exactly one operation in this part of ISO/IEC 13719 except possibly when the abstract operation has distinct functionality depending on the values of one or more parameters.

Each operation in this part of ISO/IEC 13719 should have the same number of parameters as does the corresponding operation in ISO/IEC 13719-1.

All operations in this part of ISO/IEC 13719 should return an integer status value. All other values returned by the operation should be passed back to the caller via an output parameter. The return value of the operation should indicate success or failure only.

Operation and parameter names should be the same in this part of ISO/IEC 13719 as they are in ISO/IEC 13719-1, with the exception that identifiers with file scope should begin 'Pcte_' and otherwise consist of lowercase letters and underscores. The PCTE standard guarantees that there are no ambiguities in names prefixed by 'Pcte_'.

All additional names introduced in this part of ISO/IEC 13719 which are visible across the interface (except header names, see 8.5) should begin 'Pcte_' and otherwise consist of lowercase letters, underscores and digits, or begin 'PCTE_' and otherwise consist of uppercase letters, underscores and digits. The PCTE standard guarantees that there are no ambiguities in names prefixed by 'PCTE_' or 'Pcte_'.

Wherever practical, types introduced for passing complex data entities between caller and operation (and vice versa) should be private types defined by this part of ISO/IEC 13719. The principle should only be ignored for reasons of ease of use and efficiency of implementation.

Each simple datatype in ISO/IEC 13719-1 should be mapped to a corresponding type defined in this part of ISO/IEC 13719. Each implementation of the binding should then be free to map the binding-defined type to an efficient C Language basic type appropriate for the platform of the implementation, within the constraints specified in this part of ISO/IEC 13719.

A general policy of memory allocation should be adopted; see 7.5

7.3 Sets and Sequences

Some Complex data entities to be passed into or retrieved from an operation are defined as sets or sequences of a base type in ISO/IEC 13719-1. Bounded set types are mapped individually to bit-significant natural numbers; unbounded set and sequence types are

mapped to a private type, **Pcte_sequence** with operations for creation, population, retrieval and deletion. These operations allow multiple elements of sets and sequences to be set and read in a single operation, from or to an array object of an appropriate base type. Thus, the data for sets and sequences can be easily manipulated using standard C Language paradigms, while allowing the implementation to choose the best implementation for such sets and sequences.

7.4 Character Strings

In ISO/IEC 13719-1, two different types are used to represent sequences of characters. String is a sequence of Octets allowing all 8-bit values and Text is a sequence of latin 1 graphic characters. Contents, string attributes etc., are of type String; keys, type names etc., are of type Text.

In the C Bindings, String is mapped to **Pcte_string** (see 8.2.11). Text is mapped to the native C language string with a possibly fixed length, i.e. `char *` with operations depending on NUL ('\0') character termination (see 8.2.6).

7.5 Memory Allocation

Communication between caller and operation is effected by the transfer of data into an operation via an input parameter or back from an operation via an output parameter. There are two types of such parameters: public and private.

All instances of a public type are allocated and managed by the caller of the operation. All instances of a private type are allocated and managed by the implementation. However, the extraction of data from a private type is again by data transfer in instances of a public type via an operation on the private type. In these cases also, the caller of the operation is required to allocate and manage the instances of the public type. The caller is further required to allocate sufficient space to contain a handle to the private type, the type of which is always a pointer to an internal data structure of undefined form. Operations on the private type are provided to create and discard these internal data structures.

Data stored in an instance of a private type is owned by the implementation. The implementation is responsible for allocating, managing and deallocating the memory used to store this data. Furthermore, after a handle to an instance of a private type has been returned to the caller of an operation, via an output parameter, the implementation is responsible for maintaining the data stored therein, until the caller explicitly indicates that the data is no longer needed, by invoking the discard operation on that instance.

Data stored in an instance of a public type and passed into an operation via an input parameter, is owned by the caller of the operation. The caller is responsible for allocating, managing and deallocating the memory used to store this data. The caller is further responsible for maintaining the data stored therein for the duration of the operation. If the implementation needs to access the data after the operation has completed, the implementation is responsible for allocating additional memory and storing therein a copy of the data.

All the operations that have sequences as "out" parameters will allocate the sequence and return it as result of operation. The user does not need to allocate the sequence in

advance: the user only needs to declare a `Pcte_sequence` variable and pass the address of that variable.

7.6 References and Names

Objects, attributes, links, and types are referred to in this part of ISO/IEC 13719 using object references, attribute references, link references and type references, respectively. References are mapped to private datatypes encapsulating two ways of designating an object, attribute, link, or type: by an external and by an internal reference (see ISO/IEC 13719-1, clause 23).

Beside these references, in this part of ISO/IEC 13719 also attribute names, link names and type names are used to refer to attributes, links, or types. These names represent external references and they are mapped to the native C language string type.

Therefore two different interfaces are provided in this Binding for clauses 9 to 22:

- one interface using names for attributes, links, and types.
- one interface using references for attributes, links, and types. All operations of this interface begin '`Pcte_h_`'. These operations are defined if an operation of the previous interface uses attributes, links, or types. Whenever new datatypes are necessary, they also begin '`Pcte_h_`' (see 8.7).

7.7 Operation Return Values

All the operations are mapped to functions which return a `Pcte_error_type` value, which indicates success (`PCTE_NO_ERROR` equivalent to `PCTE_OK`) or failure (one of the other enumeration values of `Pcte_error_type`) of the operation. All other information that is passed between the caller and the operation is passed via "out" or "in-out" parameters.

The error code `PCTE_NO_ERROR` is equivalent to `PCTE_OK` (as both have the value zero).

Additionally the global variable `Pcte_error_number` is set to the same return value of the function after each PCTE call, but it should be noted that in case of multi-threading or queue handlers this global variable cannot be relied upon.

7.8 Error Conditions

Error conditions which are defined in ISO/IEC 13719-1 and which can be established and returned by the operations defined in this part of ISO/IEC 13719 are described in clause 25.

All binding defined errors are defined in 25.1(2).

7.9 Identifiers

Many of the identifiers in ISO/IEC 13719-1 are longer than 31 characters, but no two identifiers are exactly the same within the first 31 characters. The C Programming Language Standard requires that an internal name (i.e., a macro name or an identifier that does not have external linkage) be unique within the first 31 characters. Thus there is no need for any identifiers to be abbreviated in this Binding.

ISO 9899 requires all identifiers of enumeration values to be distinct. Where ISO/IEC 13719-1 uses the same identifier for values of enumeration types bound to different C Language enumeration types, new names have been invented.

In a few cases an abstract operation has been bound to more than one C Language function, to cater for optional parameters; in these cases also, new names have been invented.

7.10 Implementation Limits.

ISO/IEC 13719-1 defines a set of limits that must be honored by all implementations of the Language Bindings. Clause 24 describes the binding-defined identifiers for these limit values and the way in which these limits can be retrieved.

8 Datatype mapping

This clause defines the mapping of the parameter and result datatypes of the operations of ISO/IEC 13719-1 (*PCTE datatypes*) to the parameter and result datatypes of the operations of this part of ISO/IEC 13719 (*C datatypes*).

PCTE datatype names are printed in normal characters.

LI Datatypes names are printed in italics.

C datatype names are printed in bold except in displayed fragments of C.

The mapping from PCTE datatypes to C datatypes is done in two stages via LI datatypes defined in CD11404.2.

8.1 Mapping of PCTE Datatypes to LI Datatypes

As far as possible the names of PCTE datatypes are retained for the corresponding LI datatypes, but some new names are introduced.

The general strategy of this mapping is as follows.

- To select for each PCTE datatype a LI datatype definition which matches the requirements of the PCTE datatype defined in ISO/IEC 13719-1. The LI datatype definition is, where possible, a primitive LI datatype or otherwise a generated LI datatype.
- To define new datatype generators where needed.
- To map PCTE datatypes with the same properties to the same LI datatype.

8.1.1 Mapping of Predefined PCTE Datatypes

The mapping of these PCTE datatypes is as defined in ISO/IEC 13719-1, clause 23 and is summarized in table 1.

Table 1 - Mapping of predefined PCTE datatypes

PCTE datatype	LI datatype
Boolean	<i>boolean</i>
Integer	<i>pcte-integer = integer</i> range (<i>MIN_INTEGER_ATTRIBUTE..MAX_INTEGER_ATTRIBUTE</i>)
Natural	<i>pcte-natural = integer</i> range (<i>0..MAX_NATURAL_ATTRIBUTE</i>)
Float	<i>pcte-float =</i> <i>real</i> (<i>10, MAX_DIGITS_FLOAT_ATTRIBUTE</i>) range (<i>MIN_FLOAT_ATTRIBUTE..MAX_FLOAT_ATTRIBUTE</i>)
Time	<i>pcte-time =</i> <i>time</i> (<i>second, 10, Pcte_accuracy_factor</i>) range (<i>MIN_TIME_ATTRIBUTE..MAX_TIME_ATTRIBUTE</i>)
Octet	<i>octet</i>
Text	<i>pcte-text = character string (repertoire)</i>
Enumerated type xxx = VALUE1 VALUE2 ..	<i>pcte-xxx = enumerated</i> (<i>value1, value2, ...</i>)

8.1.2 Mapping of Private PCTE Datatypes

Table 2 - Mapping of private PCTE datatypes

PCTE datatype	LI datatype
Address	<i>address</i>
Attribute_reference	<i>attribute_reference</i>
Contents_handle	<i>contents_handle</i>
Handler	<i>handler</i>
Object_reference	<i>object_reference</i>
Link_reference	<i>link_reference</i>
Position_handle	<i>position_handle</i>
Profile_handle	<i>profile_handle</i>
Type_reference	<i>type_reference</i>

8.1.3 Mapping of Complex PCTE Datatypes

PCTE sequence datatypes are mapped to the new datatype generator *pcte-sequence* (see 8.1.4).

PCTE set datatypes are divided into bounded set types and unbounded set types. Bounded set types have values which are sets of enumeration values with at most 32 possible elements; all others are unbounded set types. Bounded set types are mapped via the new LI datatype generator *Bounded-set*. Unbounded set types are mapped via the new LI datatype generator *Sequence*, the order of elements being irrelevant.

When used as input parameter of an operation in clauses 9 to 22, a sequence which represents a PCTE unbounded set may contain repeated elements. The effect for the operation is as though each element occurred only once.

When returned as the result of an operation in clauses 9 to 22, an unbounded set has no repeated elements.

PCTE map are notionally mapped via a new LI datatype generator *Map*; their mappings to C datatypes are defined directly. The mapping of *Attribute_assignments* is defined in 9.1. The mapping of *Access_rights*, *Acl*, and *Atomic_access_rights* is defined in 19.1.

PCTE union datatypes other than enumerations are mapped via the datatype generator *Choice*.

PCTE composite and bracketed datatypes (except private PCTE datatypes, for which see 8.1.2) are mapped to the datatype generator *Record*.

8.1.4 New LI Datatype Generators

Pcte-Sequence

Description: *Pcte-sequence* is a datatype generator derived from *Sequence* by adding further characterizing operations. In some operations, an index of LI datatype *pcte-*

natural is used to identify elements in the sequence. The first element is always indexed from 0.

The characterizing operations are IsEmpty, Head, Tail, Equal, Empty, and Append from *Sequence* plus Get, Put, Copy, LengthOf, and IndexOf.

Get(s: sequence of *base*, index: Natural): *base* is undefined if InOrder(LengthOf(s), index) is true or is Head(s) if Equal(index, 0) is true; otherwise Get(Tail(s), Add(index, negate(1))).

Put(s: sequence of *base*, e: *base*, index: Natural): sequence of *base* is undefined if InOrder(Add(LengthOf(s), 1), index) is true or is Append(Create(e), s) if Equal(index, 0) is true; otherwise Append(Head(s), Put(Tail(s), e, Add(index, Negate(1)))).

Copy (s: sequence of *base*): sequence of *base* is Create() if IsEmpty(s) is true; otherwise Append(Head(s), Copy(Tail(s))).

LengthOf(s: sequence of *base*): Natural is 0 if IsEmpty(s); otherwise Add(LengthOf(Tail(s)), 1).

IndexOf(s: sequence of *base*, e: *base*): Natural is undefined if IsEmpty(s) is true or is 1 if Equal(Head(s), e) is true; otherwise Add(IndexOf(Tail(s), e), 1).

Bounded-set

Description: Bounded-set is a datatype generator derived from *Set* by restricting the cardinality of the values to 32 or less.

bounded-set of (*base*) = new set of (*base*) : size (0..32);

The characterizing operations are IsIn, Subset, Equal, Difference, Union, Intersection, Empty, SetOf, Select from *Set*.

8.2 Mapping of LI Datatypes to C Datatypes

8.2.1 LI Datatype: boolean

The LI datatype *boolean* is mapped to the C datatype **Pcte_boolean**.

typedef <boolean-type> Pcte_boolean;

where <boolean-type> is a C integer type.

TRUE is represented by PCTE_TRUE, FALSE is represented by PCTE_FALSE:

```
#define PCTE_TRUE (Pcte_boolean) 1
#define PCTE_FALSE (Pcte_boolean) 0
```

In addition, if a value of type **Pcte_boolean** is supplied as or as part of an input parameter, it is taken as TRUE if it is not 0.

Characterizing Operations

Operation	C Operation
Equal(b1,b2):b	if (b1 == b2) b = PCTE_TRUE; else b = PCTE_FALSE;
Not(b1):b	if (b1 == PCTE_FALSE) b = PCTE_TRUE; else b = PCTE_FALSE;
And(b1,b2):b	b = b1 && b2;
Or(b1,b2):b	b = b1 b2;

8.2.2 LI Datatype: *pcte-integer*

The LI datatype *pcte-integer* is mapped to the C datatype **Pcte_integer**.

typedef <integer-type> Pcte_integer;

where <integer-type> is a C integer type including the range MIN_INTEGER_ATTRIBUTE to MAX_INTEGER_ATTRIBUTE inclusive.

Characterizing Operations

Operation	C Operation
Equal(i1,i2):b	if (i1 == i2) b = PCTE_TRUE; else b = PCTE_FALSE;
InOrder(i1,i2):b	if (i1 <= i2) b = PCTE_TRUE; else b = PCTE_FALSE;
NonNegative(i):b	if (i >= 0) b = PCTE_TRUE; else b = PCTE_FALSE;
Negate(i1):i	i = -i1
Add(i1,i2):i	i = i1 + i2
Multiply(i1,i2):i	i = i1 * i2

8.2.3 LI Datatype: pcte-natural

The LI datatype *pcte-natural* is mapped to the C datatype **Pcte_natural**.

typedef <natural-type> Pcte_natural;

where <natural-type> is a C unsigned integer type including the range 0 to MAX_NATURAL_ATTRIBUTE inclusive.

Furthermore, the unsigned integral datatype in C Language operates under modular arithmetic rules; i.e. the result of any arithmetic operation is always reduced modulo the largest representable value. Therefore, ULONG_MAX + 1 = 0, where ULONG_MAX is the largest value of the chosen base unsigned integer value.

Characterizing Operations

Operation	C Operation
Equal(n1,n2):b	if (n1 == n2) b = PCTE_TRUE; else b = PCTE_FALSE;
InOrder(n1,n2):b	if (n1 <= n2) b = PCTE_TRUE; else b = PCTE_FALSE;
Add(n1,n2):n	n = n1 + n2
Multiply(n1,n2):n	n = n1 * n2

8.2.4 LI Datatype: *pcte-float*

The LI datatype *pcte-float* is mapped to the C datatype **Pcte_float**.

typedef <float-type> Pcte_float;

where <float-type> is a C floating type including the range MIN_FLOAT_ATTRIBUTE to MAX_FLOAT_ATTRIBUTE inclusive, with an accuracy of at least MAX_DIGITS_FLOAT_ATTRIBUTE decimal digits, and able to represent SMALLEST_FLOAT_ATTRIBUTE.

Characterizing Operations

Operation	C Operation
Equal(f1,f2):b	if (f1 == f2) b = PCTE_TRUE; else b = PCTE_FALSE;
InOrder(f1,f2):b	if (f1 <= f2) b = PCTE_TRUE; else b = PCTE_FALSE;
NonNegative(i):b	if (f >= 0.0) b = PCTE_TRUE; else b = PCTE_FALSE;
Negate(f1):f	f = -f1
Add(f1,f2):f	f = f1 + f2
Multiply(f1,f2):f	f = f1 * f2
Reciprocal(f1):f	f = 1.0 / f1;

8.2.5 LI Datatype: pcte-time

The LI datatype *pcte-time* is mapped to the C datatype **Pcte_time**, where *Pcte_time_accuracy_factor* is an implementation-defined constant for the resolution of for time attributes.

```
typedef time_t Pcte_time;

#define Pcte_time_accuracy_factor (Pcte_natural) <implementation-defined>

#define Pcte_reference_time (Pcte_time) <implementation defined>
```

The datatype **time_t** is an arithmetic datatype that holds values representing time. The encoding of the calendar time within a value of type **time_t** is undefined. Functions on values of type **time_t** are provided to convert such a value into a meaningful and usable representation of calendar time.

The implementation shall provide a constant defining an optional time parameter in the operation *Pcte_object_set_time_attributes*.

```
#define Pcte_null_time (Pcte_time) <implementation defined>
```

Characterizing Operations

Operation	C Operation
Equal(t1,t2):b	if (difftime(t1,t2) < radix_factor) b = PCTE_TRUE; else b = PCTE_FALSE;
InOrder(t1,t2):b	if (difftime(t1,t2) >= 0) b = PCTE_TRUE; else b = PCTE_FALSE;
Difference(t1,t2):i	i = (Pcte_integer) difftime(t1,t2);
Extend.res1tores2(t1):t2	t2 = t1 * radix_factor;
Round.res1tores2(t1):t2	t2 = floor (t1 + radix_factor);

In this table radix_factor is 10 ^{-Pcte_time_accuracy_factor}

8.2.6 LI Datatype: pcte-text

The LI datatype *pcte-text* is mapped to the native C language null terminated string type with a fixed length. The character repertoire of the characters of such a sequence are the graphic characters of ISO 8859-1. **Pcte_octet** is used to represent this character repertoire. Attribute names, enumerals type images, exact identifier, keys, machine names, node names, link names, names, type names, and type names in SDSs are of type Text.

```
#define PCTE_MAX_XXX_SIZE <implementation-defined>
typedef Pcte_octet Pcte_XXX[PCTE_MAX_XXX_SIZE + 1];
```

XXX is one of the following PCTE datatypes: Attribute_name, Enumeral_type_image, Exact_identifier, Key, Link_name, Machine_name, Name, Node_name, and Type_name, or Type_name_in_sds. All values of PCTE XXX_SIZE defined in this Binding are minimum values, which have to be respected by an implementation.

Characterizing Operations

Operation	C Operation
Equal(s1,s2):b	if (strcmp(s1,s2) == 0) b = PCTE_TRUE; else b = PCTE_FALSE;
Empty():s	s[0] = '\0';
Head(s):c	c = s[0];
Tail(s1):s2	if (s1[0] != '\0') { s2 = malloc(strlen(s1)); (void) strncpy(s2,s1+1,strlen(s1)); }
Append(s1,e):s2	s2 = malloc(strlen(s1)+2); (void) strcpy(s2,s1); (void) strcat(s2,"e");
IsEmpty(s):b	if (s[0] == '\0') b = PCTE_TRUE; else b = PCTE_FALSE;

Note, that in this table memory errors like access at invalid addresses are not recognized.

8.2.7 LI Datatype: *octet*

The LI datatype *octet* is mapped to the C datatype **char**.

typedef char Pcte_octet;

Characterizing Operations

Operation	C Operation
Equal(o1,o2):b	if (o1 == o2) b = PCTE_TRUE; else b = PCTE_FALSE;

8.2.8 LI Enumerated Datatype: *pcte-xxx*

The LI datatype *pcte-xxx*, defined as *enumerated* (value1, value2, ...), corresponds to the PCTE enumeration datatype xxx (where the values of xxx are VALUE1, VALUE2, ...). It is mapped to the C datatype **Pcte_xxx**, defined as follows.

Case (1): if a bounded set of the enumeration type is not required the mapping is:

typedef enum {PCTE_VAL1, PCTE_VAL2, ...} Pcte_xxx;

Case (2): if a bounded set of the enumeration type is required the mapping is as defined in 8.2.12

Characterizing Operations

Operation	C Operation
Equal(e1,e2):b	if (e1 == e2) b = PCTE_TRUE; else b = PCTE_FALSE;
InOrder(e1,e2):b	if (e1 <= e2) b = PCTE_TRUE; else b = PCTE_FALSE;
Successor(e1):e2	e2 = e1+1;

8.2.9 LI Private Datatypes

Each LI private datatype xxx is mapped to an opaque C datatype *Pcte_xxx*.

typedef void *Pcte_xxx;

8.2.10 LI Datatype Generator: pcte-sequence

The LI datatypes of the family *pcte-sequence* are mapped to the C datatype **Pcte_sequence** (except for *pcte-sequence* of *octet*, see 8.2.7).

Since **Pcte_sequence** is an opaque type in the C Language, objects of this type may be manipulated only via the binding defined operations (see 8.5.1).

Characterizing Operations

Operation	C Operation
Equal(s1,s2):b	Pcte_sequences_are_equal (s1,s2,&b);
Empty():s	Pcte_sequence_create(-,NULL,0,&s);
Head(s):e	Pcte_sequence_get(s,0,&e);
Tail(s1):s2	Pcte_sequence_delete(s,0,1);
Append(s1,e):s2	Pcte_natural n; Pcte_sequence_get_length(s1,&n); Pcte_sequence_copy(s1,&s2,0,0,n); Pcte_sequence_put(s2,n+1,&e);
IsEmpty(s):b	Pcte_natural n; Pcte_sequence_get_length(s,&n); if (n == 0) b = PCTE_TRUE; else b = PCTE_FALSE;
Copy(s1):s2	Pcte_natural n; Pcte_sequence_get_length(s1,&n); Pcte_sequence_copy(s1,&s2,0,0,n);
Get(s,n):e	Pcte_sequence_get(s,n,&e);
Put(s1,e,n):s2	Pcte_natural n; Pcte_sequence_get_length(s1,&n); Pcte_sequence_copy(s1,&s2,0,0,n); Pcte_sequence_put(s2,n,&e);
LengthOf(s):n	Pcte_sequence_get_length(s,&n);
IndexOf(s,e):n	Pcte_sequence_get_index(s,&e,&n);

If a sequence is to be created using `Pcte_sequence_create`, a type of **Pcte_sequence_type** has to be supplied. In the table given above, this is indicated using '-'.

The sequence operations, `Pcte_sequence_XXX`, which are used above are defined in 8.5.1.

8.2.11 LI Datatype: *pcte-string*

The LI datatype *pcte-string*, which is a *pcte-sequence* of *octets*, is mapped to the C datatype **Pcte_string**. Attribute values, contents, control data device characteristics, foreign devices, foreign names, foreign parameters, messages, pathnames, process data, relative pathnames, security labels, and volume characteristics are Strings.

```
typedef struct {
    Pcte_natural    size;
    Pcte_octet      *array;
} Pcte_string;
```

If a value of type **Pcte_string** is passed as or as part of an input parameter, **size** defines the valid number of octets given by **array**. If **size** is bigger than the number of octets allocated in **array**, the error **PCTE_ACCESS_AT_INVALID_ADDRESS** is raised.

If a value of type **Pcte_string** is passed as or as part of an output parameter, the space for the string can be either allocated by the caller or by the implementation:

- if the supplied **size** is set to 0, the space is allocated by the implementation; in this case it is in the responsibility of the caller to discard the space when the string is no longer needed, by using the operation **Pcte_string_discard**.
- if the supplied **size** is non-zero, it indicates the number of octets allocated by the user in **array**.

The implementation sets **size** to the number of returned octets, provided by the implementation in **array**. If there is not enough space in **array**, the implementation raises the error **PCTE_STRING_TOO_SHORT** or in case of a wrong set **size** **PCTE_ACCESS_AT_INVALID_ADDRESS**.

In order to improve usability, there are four exceptions to this rule: **CONTENTS_READ**, **CONTENTS_WRITE**, **PROCESS_PEEK**, and **PROCESS_POKE**. In all cases an array of **Pcte_octet** and an additional size parameter has to be supplied. For these two parameters the same rules apply for incoming and outgoing parameters as for *pcte-string*. In addition, pathname and relative pathname are mapped to the native C language string type (see 23.1). For these exceptions no characterizing operations are defined.

The following operations is provided.

Pcte_string_discard

```
Pcte_error_type Pcte_string_discard (
    Pcte_string    string
);
```

Pcte_string_discard discards the space allocated by the implementation for **string.array**

Characterizing Operations

Operation	C Operation
Equal(s1,s2):b	if ((s1.size == s2.size) && (memcmp(s1.array, s2.array, s1.size) == 0)) b = PCTE_TRUE; else b = PCTE_FALSE;
Empty():s	s.size == 0;
Head(s):c	c = s.array[0];
Tail(s1):s2	s2.size = s1.size - 1; s2.array = malloc(s2.size); memcpy(s2.array, s1.array, s2.size);
Append(s1,e):s2	s2.size = s1.size + 1; s2.array = malloc(s2.size); memcpy(s2.array, s1.array, s2.size); s2.array[s2.size] = e;
IsEmpty(s):b	if (s.size == 0) b = PCTE_TRUE; else b = PCTE_FALSE;
Copy(s1):s2	s2.size = s1.size; s2.array = malloc(s2.size); memcpy(s2.array, s1.array, s2.size);
Get(s,n):e	e = s.array[n];
Put(s1,e,n):s2	s2.size = s1.size; s2.array = malloc(s2.size); memcpy(s2.array, s1.array, s2.size); s2.array[n] = e;
LengthOf(s):n	n = s.size;
IndexOf(s,e):n	n = 0; while (s.array[n] != e) n++;

Note, that in this table memory errors like access at invalid addresses are not recognized.

8.2.12 LI Datatype Generator: bounded-set

Each LI datatype *bounded-set of (base)* is mapped to a different C datatype, defined as **Pcte_natural** (see 8.2.3), with an enumeration definition mapping the names of element values to bit positions. A natural value represents a set including an element value if the corresponding bit is 1. Thus a bounded set <bounded-set>, where <bounded-set> is either **Pcte_access_events**, **Pcte_categories**, **Pcte_definition_mode_values**, **Pcte_discretionary_access_modes**, or **Pcte_work_status**, with possible element values VALUE1, VALUE2, etc. is mapped as follows, making use of the corresponding enumerated types <enumerated-type>, where <enumerated-type> is either **Pcte_access_event**, **Pcte_category**, **Pcte_definition_mode_value**, **Pcte_discretionary_access_mode**, or **Pcte_work_status_item**, respectively:

```
enum {
    PCTE_VALUE1    =    1<<0,
    PCTE_VALUE2    =    1<<1,
    PCTE_VALUE3    =    1<<2,
    ...
} <enumerated-type>;

typedef Pcte_natural <bounded-set>;
```

Characterizing Operations

Operation	C Operation
Equal(s1,s2):b	if (s1 == s2) b = PCTE_TRUE; else b = PCTE_FALSE;
Empty ():s	s = 0;
IsIn (s,e):b	if (s & e) b = PCTE_TRUE; else b = PCTE_FALSE;
Subset(s1,s2):b	if (s2 == (s2 & s1)) b = PCTE_TRUE; else b = PCTE_FALSE;
Complement (s1):s2	s2 = ~s1;
Union (s1,s2):s3	s3 = s1 s2;
Intersection (s1,s2):s3	s3 = s1 & s2;
SetOf(e):s	s = e;
Select(s):e	for (e=1;;e<<1) if (s & e) break;

8.2.13 LI Datatype Generator: choice

The LI datatype *pcte-xxx = choice (alt-1, alt-2, ...)* is mapped to a C union datatype as follows:

Each value is discriminated by an enumeration, the discriminator, with its origin type. The position of the enumeration value of the discriminator corresponds to the position of the type of the choice, i.e. if the discriminator has the value PCTE_TYPE_1, the value of the choice is of type Pcte_type_1, etc.

```
typedef struct {
    enum {
        PCTE_TYPE_1,
        PCTE_TYPE_2,
        ...
    } union_type;
    union {
        Pcte_type_1 alt-1;
        Pcte_type_2 alt-2;
        ....;
    } choice;
} Pcte_xxx ;
```

Characterizing Operations

Operation	C Operation
Equal(c1,c2):b	if (c1 == c2) b = PCTE_TRUE; else b = PCTE_FALSE;
Tag.type-n(x,type-n):c	c.union_type = PCTE_TYPE_N; c.choice = x;
IsType.type-n(c):b	if (c.union_type == PCTE_TYPE_N) b = PCTE_TRUE; else b = PCTE_FALSE;
Case.type-n(c):x	x = c.choice;

8.2.14 LI Datatype: record

The LI datatype generator *Record* is mapped to the C language struct type as follows:

```
typedef struct {
    Pcte_type_1    component_1;
    Pcte_type_2    component_2;
    Pcte_type_3    component_3;
    ...;
} Pcte_xxx ;
```

Characterizing Operations

Operation	C Operation
Equal(r1,r2):b	if (r1 == r2) b = PCTE_TRUE; else b = PCTE_FALSE;
FieldSelect.c(r):c	c = r.c;
FieldReplace(r1,c):r2	r2 = r1; r2.c = c;

8.3 Private Datatypes

Private Datatypes are datatypes with a hidden representation, i.e. the implementation-defined representation of such a datatype is not defined in any header. These datatypes are denoted in the C language using **void ***. The table below shows all private types which are defined in this binding with the corresponding operations to create and discards instances of these types.

Datatypes	Create Operations	Discard Operations
Pcte_attribute_reference	Pcte_type_reference_copy Pcte_type_reference_set	Pcte_type_reference_unset
Pcte_contents_handle	Pcte_contents_open Pcte_contents_get_handle_- from_key Pcte_contents_handle_- duplicate Pcte_contents_handle_- duplicate_to_key	Pcte_contents_close
Pcte_link_reference	Pcte_link_reference_copy Pcte_link_reference_set	Pcte_link_reference_unset
Pcte_object_reference	Pcte_reference_copy Pcte_reference_set_absolute Pcte_reference_set_relative	Pcte_reference_unset
Pcte_position_handle	Pcte_contents_get_position	Pcte_position_handle_discard Pcte_contents_close
Pcte_profile_handle	Pcte_process_profiling_on	Pcte_process_profiling_off
Pcte_sequence	Pcte_sequence_create	Pcte_sequence_discard
Pcte_type_reference	Pcte_type_reference_copy Pcte_type_reference_set	Pcte_type_reference_unset

8.4 References and Names

The LI datatypes *attribute_reference*, *link_reference* and *type_reference* are mapped to the C datatypes **Pcte_attribute_name**, **Pcte_link_name**, and **Pcte_type_name**.

If an attribute reference, link reference, or type reference is passed as part of a parameter, the corresponding parts of the LI datatype of that parameter is also mapped to the C datatypes **Pcte_attribute_name**, **Pcte_link_name**, or **Pcte_type_name**.

Whenever a parameter of an operation in clause 9 to 22 is either an attribute reference, link reference or type reference, a second interface is defined beginning 'Pcte_h_' instead of 'Pcte_'. In this case, the LI datatypes *attribute_reference*, *link_reference* and *type_reference* are mapped to the C datatypes **Pcte_attribute_reference**, **Pcte_link_reference**, and **Pcte_type_reference** (see clause 23).

It may be necessary that if an attribute reference, link reference, or type reference is passed as part of a parameter of the C datatype **Pcte_xxx**, a new C datatype **Pcte_h_xxx** is defined using the C datatypes **Pcte_attribute_reference**, **Pcte_link_reference**, and **Pcte_type_reference**.

All definitions concerning operations beginning with 'Pcte_' also hold for 'Pcte_h_'.

8.5 C Private Type: Pcte_sequence

The new LI datatype *Sequence* is mapped to a family of C datatypes defined by the C datatype **Pcte_sequence**, defined as follows:

```
typedef void *Pcte_sequence;
#define Pcte_null_sequence (Pcte_sequence) NULL
```

In addition, there is for each LI datatype *xxx* which is a *sequence*, except *text* and *Pcte_string*, a C datatype with the corresponding name **Pcte_xxx**. Instead of **Pcte_h_attribute_references**, **Pcte_h_link_references**, and **Pcte_h_type_references**, the C datatypes **Pcte_attribute_names**, **Pcte_link_names**, and **Pcte_type_names** are used.

As ACL and attribute assignments are also defined as sequences (see 9.1, 19.1) corresponding C datatypes for these PCTE datatypes are also provided.

```
typedef Pcte_sequence Pcte_accounting_log;
typedef Pcte_sequence Pcte_acl;
typedef Pcte_sequence Pcte_audit_file;
typedef Pcte_sequence Pcte_attribute_assignments;
typedef Pcte_sequence Pcte_h_attribute_assignments;
typedef Pcte_sequence Pcte_attribute_names;
typedef Pcte_sequence Pcte_attribute_references;
typedef Pcte_sequence Pcte_buffer;
typedef Pcte_sequence Pcte_confidentiality_criteria;
typedef Pcte_sequence Pcte_enumeration_value_type;
typedef Pcte_sequence Pcte_h_enumeration_value_type;
typedef Pcte_sequence Pcte_enumeration_value_type_in_sds;
typedef Pcte_sequence Pcte_general_criteria;
typedef Pcte_sequence Pcte_integrity_criteria;
typedef Pcte_sequence Pcte_key_types;
typedef Pcte_sequence Pcte_h_key_types;
```

```

typedef Pcte_sequence Pcte_key_types_in_sds;
typedef Pcte_sequence Pcte_link_set_descriptors;
typedef Pcte_sequence Pcte_h_link_set_descriptors;
typedef Pcte_sequence Pcte_link_names;
typedef Pcte_sequence Pcte_link_references;
typedef Pcte_sequence Pcte_message_types;
typedef Pcte_sequence Pcte_name_sequence;
typedef Pcte_sequence Pcte_object_criteria;
typedef Pcte_sequence Pcte_object_references;
typedef Pcte_sequence Pcte_type_names;
typedef Pcte_sequence Pcte_type_names_in_sds;
typedef Pcte_sequence Pcte_type_references;
typedef Pcte_sequence Pcte_user_criteria;
typedef Pcte_sequence Pcte_volume_infos;

```

The following additional datatypes are used in the operational interfaces to these private types:

```

typedef enum {
    PCTE_ACCOUNTING_FILE, PCTE_ACL,
    PCTE_AUDIT_FILE, PCTE_ATTRIBUTE_ASSIGNMENTS,
    PCTE_H_ATTRIBUTE_ASSIGNMENTS,
    PCTE_ATTRIBUTE_NAMES, PCTE_ATTRIBUTE_REFERENCES,
    PCTE_BUFFER, PCTE_CONFIDENTIALITY_CRITERIA,
    PCTE_ENUMERATION_VALUE_TYPE,
    PCTE_H_ENUMERATION_VALUE_TYPE,
    PCTE_ENUMERATION_VALUE_TYPE_IN_SDS,
    PCTE_GENERAL_CRITERIA, PCTE_INTEGRITY_CRITERIA,
    PCTE_KEY_TYPES, PCTE_H_KEY_TYPES,
    PCTE_KEY_TYPES_IN_SDS, PCTE_LINK_NAMES,
    PCTE_LINK_SET_DESCRIPTOR, PCTE_H_LINK_SET_DESCRIPTOR,
    PCTE_LINK_REFERENCES, PCTE_MESSAGE_TYPES,
    PCTE_NAME_SEQUENCE, PCTE_OBJECT_CRITERIA,
    PCTE_OBJECT_REFERENCES, PCTE_TYPE_NAMES,
    PCTE_TYPE_NAMES_IN_SDS, PCTE_TYPE_REFERENCES,
    PCTE_USER_CRITERIA, PCTE_VOLUME_INFOS
} Pcte_sequence_type;

typedef void *Pcte_sequence_element;

typedef void *Pcte_array_of_sequence_elements;

```

Values of the type **Pcte_sequence_element** are used to set or retrieve a single element of a sequence using the operations **Pcte_put_element** or **Pcte_get_element**. In both cases, a pointer to a single value of the sequence element type has to be provided. For example, if the sequence is of type **Pcte_object_references**, a pointer to a value of type **Pcte_object_reference** has to be provided.

Pcte_array_of_sequence_elements is used to set or retrieve several elements to or from a sequence using the operations **Pcte_sequence_create**, **Pcte_sequence_insert_elements**, and **Pcte_sequence_get_elements**. In this case, a pointer to an array of the sequence element type has to be provided. For example, if the sequence is of type **Pcte_object_references**, a pointer to an array of **Pcte_object_reference** has to be provided.

The enumeration type **Pcte_sequence_type** is used to indicate the C element type of the sequence as follows:

enumeration value	C element type
PCTE_ACCOUNTING_FILE	Pcte_accounting_record
PCTE_ACL	Pcte_acl_entry
PCTE_AUDIT_FILE	Pcte_auditing_record
PCTE_ATTRIBUTE_ASSIGNMENTS	Pcte_attribute_assignment
PCTE_H_ATTRIBUTE_ASSIGNMENTS	Pcte_h_attribute_assignment
PCTE_ATTRIBUTE_NAMES	Pcte_attribute_name
PCTE_ATTRIBUTE_REFERENCES	Pcte_attribute_reference
PCTE_BUFFER	Pcte_natural
PCTE_CONFIDENTIALITY_CRITERIA	Pcte_confidentiality_criterion
PCTE_ENUMERATION_VALUE_TYPE	Pcte_type_name
PCTE_H_ENUMERATION_VALUE_TYPE	Pcte_type_reference
PCTE_ENUMERATION_VALUE_TYPE_IN_SDS	Pcte_type_name_in_sds
PCTE_GENERAL_CRITERIA	Pcte_general_criterion
PCTE_INTEGRITY_CRITERIA	Pcte_integrity_criterion
PCTE_KEY_TYPES	Pcte_type_name
PCTE_H_KEY_TYPES	Pcte_type_reference
PCTE_KEY_TYPES_IN_SDS	Pcte_type_name_in_sds
PCTE_LINK_NAMES	Pcte_link_name
PCTE_LINK_SET_DESCRIPTOR	Pcte_link_set_descriptor
PCTE_H_LINK_SET_DESCRIPTOR	Pcte_h_link_set_descriptor
PCTE_LINK_REFERENCES	Pcte_link_reference
PCTE_MESSAGE_TYPES	Pcte_message_type
PCTE_NAME_SEQUENCE	Pcte_name
PCTE_OBJECT_CRITERIA	Pcte_object_criterion
PCTE_OBJECT_REFERENCES	Pcte_object_reference
PCTE_TYPE_NAMES	Pcte_type_name
PCTE_TYPE_NAMES_IN_SDS	Pcte_type_name_in_sds
PCTE_TYPE_REFERENCES	Pcte_type_reference
PCTE_USER_CRITERIA	Pcte_user_criterion
PCTE_VOLUME_INFOS	Pcte_volume_info

8.5.1 Operations on Sequences

The following operations are defined on objects of type **Pcte_sequence**. Each operation returns a status via its return value, which indicates its success or failure. For an interpretation of returned status values, see clause 25, Error Conditions.

In some operations, elements of a sequence can be referenced using an index according to the positions of elements within the sequence. The first element of a

sequence has the position 0.

Pcte_sequence_create

```
Pcte_error_type Pcte_sequence_create (
    Pcte_sequence_type      type,
    Pcte_array_of_sequence_elements data,
    Pcte_natural             count,
    Pcte_sequence           *sequence
);
```

Pcte_sequence_create creates an instance of the sequence type specified by *type*, with *count* element values specified by *data*, and returns a pointer to it in *sequence*.

Pcte_sequence_discard

```
Pcte_error_type Pcte_sequence_discard (
    Pcte_sequence *sequence
);
```

Pcte_sequence_discard releases the contents of the sequence pointed to by *sequence* and discards any allocated data structures associated with it. The data structures allocated with each element are also discarded, even if they are themselves sequences. A null pointer is returned in *sequence*. When a sequence of types containing an object reference (i.e. *object_references*, *link_set_descriptors* and *object_criterion*) is discarded, the object references contained in the sequence are unset as defined by the operation *object_reference_unset*.

Pcte_sequence_copy

```
Pcte_error_type Pcte_sequence_copy (
    Pcte_sequence source_list,
    Pcte_sequence destination_list,
    Pcte_natural index,
    Pcte_natural source_index,
    Pcte_natural count
);
```

Pcte_sequence_copy copies *count* subsequent elements of data from the sequence *source_list* from the position *source_index* to the sequence *destination_list* before the element at position *index*. If *index* is greater than the number of elements of *destination_list*, these elements are appended at the end of *destination_list*. Copying sequences is a deep copy operation, and object references are copied as they are with the status and evaluation point. Deallocating the source sequence or any of its elements will not affect the destination.

Pcte_sequence_insert_elements

```

Pcte_error_type Pcte_sequence_insert_elements (
    Pcte_sequence      sequence,
    Pcte_natural        index,
    Pcte_array_of_sequence_elements data,
    Pcte_natural        count
);

```

`Pcte_sequence_insert_elements` inserts the *count* elements specified by *data* into the sequence *sequence* before the element at position *index*.

Pcte_sequence_delete

```

Pcte_error_type Pcte_sequence_delete (
    Pcte_sequence      sequence,
    Pcte_natural        index,
    Pcte_natural        count
);

```

`Pcte_sequence_delete` removes the *count* subsequent elements from the sequence *sequence* starting with the element at position *index*.

Pcte_sequences_are_equal

```

Pcte_error_type Pcte_sequences_are_equal (
    Pcte_sequence      first_sequence,
    Pcte_sequence      second_sequence,
    Pcte_boolean        *equality
);

```

`Pcte_sequences_are_equal` returns in *equality* the value `PCTE_TRUE` if the two sequences *first_sequence* and *second_sequence* are equal, and the value `PCTE_FALSE` otherwise. Two sequences are equal if they contain the same number of elements and the values of corresponding elements are equal.

Pcte_sequence_get_index:

```

Pcte_error_type Pcte_sequence_get_index (
    Pcte_sequence      sequence,
    Pcte_sequence_element element,
    Pcte_integer        *index
);

```

`Pcte_sequence_get_index` returns in *index* the index of the first occurrence of the given element *element* in the sequence *sequence* if the element is a member of the sequence; otherwise -1 is returned.

Pcte_sequence_get_length

```

Pcte_error_type Pcte_sequence_get_length (
    Pcte_sequence    sequence,
    Pcte_natural     *length
);

```

Pcte_sequence_get_length returns in *length* the number of elements of the sequence *sequence*.

Pcte_sequence_get_elements

```

Pcte_error_type Pcte_sequence_get_elements (
    Pcte_sequence    sequence,
    Pcte_natural     index,
    Pcte_array_of_sequence_elements data,
    Pcte_natural     count
);

```

Pcte_sequence_get_elements returns *count* elements from the sequence *sequence* into *data*, starting from the element at position *index*.

Pcte_sequence_get

```

Pcte_error_type Pcte_sequence_get (
    Pcte_sequence    sequence,
    Pcte_natural     index,
    Pcte_sequence_element element
);

```

Pcte_sequence_get returns in *element* a copy of the element specified by *index* from the sequence *sequence*.

Pcte_sequence_insert

```

Pcte_error_type Pcte_sequence_insert (
    Pcte_sequence    sequence,
    Pcte_natural     index,
    Pcte_sequence_element element
);

```

Pcte_sequence_insert inserts in the sequence *sequence* the element *element* immediately before the element specified by *index*. If *index* is not less than the length of *sequence*, the element is appended to *sequence*.

Pcte_sequence_replace

```

Pcte_error_type Pcte_sequence_replace (
    Pcte_sequence      sequence,
    Pcte_natural        index,
    Pcte_sequence_element element
);

```

`Pcte_sequence_replace` replaces in the sequence *sequence* the element specified by *index* with the element *element*. If *index* is not less than the length of *sequence*, the element is appended to *sequence*.

Pcte_sequence_append

```

Pcte_error_type Pcte_sequence_append (
    Pcte_sequence      sequence,
    Pcte_sequence_element element
);

```

`Pcte_sequence_append` appends to the sequence *sequence* the element specified by *element*.

Pcte_sequence_normalize

```

Pcte_error_type Pcte_sequence_normalize (
    Pcte_sequence      sequence
);

```

`Pcte_sequence_normalize` reorganizes the sequence *sequence* so that each element value occurs once only, and elements are in an implementation-defined order. Two sequences containing the same element values (i.e. representing the same unbounded set) are equal after normalization.

8.5.2 Error Conditions for Sequence Operators

The following error conditions may be raised by operations on sequences.

- `PCTE_SEQUENCE_INVALID_TYPE`. Raised by any operation, if the type of a specified sequence does not match the required sequence type for that sequence.
- `PCTE_SEQUENCE_BAD_HANDLE`. Raised by any operation, if a specified sequence handle is not a valid sequence handle.
- `PCTE_SEQUENCE_OUT_OF_DATA`. Raised by `Pcte_sequence_get_elements`, if *sequence* does not contain *count* elements from position *index*. Similarly for `Pcte_sequence_copy`.
- `PCTE_SEQUENCE_INVALID_INDEX`. Raised by `Pcte_sequence_get`, if *sequence* does not contain an element at position *index*.

For further information on error conditions leading to an operational failure, see clause 25, Error Conditions.

8.6 Deriving C Function Semantics From Abstract Specification.

Each C function corresponds to the abstract operation in ISO/IEC 13719-1. The semantics of the C function are generally the same as those of the corresponding abstract operation and are derived as follows:

- C function parameters correspond to abstract operation parameters with the same names.
- C function parameter datatypes correspond to the abstract specification datatypes of the corresponding abstract parameters as defined in clause 8.
- Returned values corresponding to results of abstract operations are returned via parametric pointers to objects which contain the returned values. If the result is a mandatory single value of a simple type, the pointer parameter has the same name as the result. In more complicated cases an explanation is given.
- If the abstract operation contains optional parameters, an explanation of the mapping is given.
- If an operation fails, each operation returns `PCTE_ERROR` and `Pcte_error_number` holds the appropriate `Pcte_error_type` value. Otherwise, `PCTE_OK` is returned and `Pcte_error_number` is unchanged.

All exceptions to these general rules are defined in the appropriate clause.

Handlers execute as separate threads, subject to implementation-defined restrictions. Apart from that case, neither the C language, nor the PCTE operations provide mechanisms for a program to execute more than one thread; but an implementation may provide such mechanisms, which may be subject to implementation-defined rules.

8.7 Headers

Each implementation of this part of ISO/IEC 13719 must provide a header `<Pcte/sequences.h>` that contains the definitions of the constant values, datatypes and operation interface bindings for the manipulation of sequences as defined in 8.3.

Each implementation of this part of ISO/IEC 13719 must provide a set of clause-specific headers that contain the definitions of the constant values, data structures, and operation interface bindings for each clause 9 to 22 of ISO/IEC 13719-1.

Each implementation of this part of ISO/IEC 13719 must provide a header `<Pcte/references.h>` that contains the definitions of the constant values, datatypes and operation interface bindings for clause 23 of ISO/IEC 13719-1.

Each implementation of this part of ISO/IEC 13719 must provide a header `<Pcte/limits.h>` that contains the definitions of the constant values defined in clause 24.

Each implementation of this part of ISO/IEC 13719 must provide a header `<Pcte/errors.h>` that contains the definitions of the constant values and datatypes for errors defined in appendix C of ISO/IEC 13719-1.

In addition, each implementation must provide a single header `<Pcte/pcte.h>` which is used to include all other headers.

Each clause-specific header must provide all definitions necessary for the correct functioning of the operations of that clause. This may be achieved by including other headers.

The name of the headers are meant to be interpreted by the C processor as a directory 'Pcte' and files 'xxx.h'. If this does not map appropriately to a file naming convention then the implementation may change the #include directive appropriately.

8.7.1 The Global Pcte Header

```
/* The header <Pcte/pcte.h> */

#ifndef PCTE_INCLUDED
#define PCTE_INCLUDED 1

#include <Pcte/types.h>           /* clause 8.7.2 */
#include <Pcte/sequences.h>       /* clause 8.7.3 */
#include <Pcte/references.h>      /* clause 23 */
#include <Pcte/limits.h>         /* clause 24 */
#include <Pcte/errors.h>         /* clause 25 */

#include <Pcte/oms.h>            /* clause 9 */
#include <Pcte/sms.h>            /* clause 10 */
#include <Pcte/devices.h>        /* clause 11 */
#include <Pcte/contents.h>       /* clause 12 */
#include <Pcte/execution.h>      /* clause 13 */
#include <Pcte/messages.h>       /* clause 14 */
#include <Pcte/notification.h>   /* clause 15 */
#include <Pcte/activities.h>     /* clause 16 */
#include <Pcte/replication.h>    /* clause 17 */
#include <Pcte/network.h>        /* clause 18 */
#include <Pcte/discretionary.h>  /* clause 19 */
#include <Pcte/mandatory.h>     /* clause 20 */
#include <Pcte/auditing.h>       /* clause 21 */
#include <Pcte/accounting.h>     /* clause 22 */

#endif /* !PCTE_INCLUDED */
```

8.7.2 The Pcte Basic Type Header

```
/* The header <Pcte/types.h> */

#ifndef PCTE_TYPES_INCLUDED
#define PCTE_TYPES_INCLUDED 1

#include <time.h>
#include <Pcte/errors.h>

#define PCTE_OK 0
#define PCTE_ERROR 1
```

```

typedef <integer-type> Pcte_boolean;
#define PCTE_TRUE (Pcte_boolean) 1
#define PCTE_FALSE (Pcte_boolean) 0

typedef <integer-type> Pcte_integer;
typedef <natural-type> Pcte_natural;
typedef <float-type> Pcte_float;
typedef time_t Pcte_time;
#define Pcte_time_accuracy_factor (Pcte_natural) <implementation-defined>
#define Pcte_reference_time (Pcte_time) <implementation defined>
#define Pcte_null_time (Pcte_time) <implementation defined>
typedef unsigned char Pcte_octet;
typedef struct {
    Pcte_natural    size;
    Pcte_octet      *array;
} Pcte_string;

/*    Empty string can be used in all the operations that have a string as input    */
/*    parameters, where the user wants to pass a null value for a string.          */
#define PCTE_NULL_STRING "\0"
Pcte_error_type Pcte_string_discard (
    Pcte_string    string
);

#endif /* !PCTE_TYPES_INCLUDED */

```

8.7.3 The Pcte Sequence Header

```

/* The header <Pcte/sequences.h> */
#ifndef PCTE_SEQUENCES_INCLUDED
#define PCTE_SEQUENCES_INCLUDED 1
#include <Pcte/types.h>
typedef void *Pcte_sequence;
#define Pcte_null_sequence (Pcte_sequence) NULL
typedef Pcte_sequence Pcte_accounting_log;
typedef Pcte_sequence Pcte_audit_file;
typedef Pcte_sequence Pcte_attribute_names;

```

```
typedef Pcte_sequence Pcte_attribute_references;
typedef Pcte_sequence Pcte_buffer;
typedef Pcte_sequence Pcte_confidentiality_criteria;
typedef Pcte_sequence Pcte_enumeration_value_type;
typedef Pcte_sequence Pcte_h_enumeration_value_type;
typedef Pcte_sequence Pcte_enumeration_value_type_in_sds;
typedef Pcte_sequence Pcte_general_criteria;
typedef Pcte_sequence Pcte_integrity_criteria;
typedef Pcte_sequence Pcte_key_types;
typedef Pcte_sequence Pcte_h_key_types;
typedef Pcte_sequence Pcte_key_types_in_sds;
typedef Pcte_sequence Pcte_link_set_descriptors;
typedef Pcte_sequence Pcte_h_link_set_descriptors;
typedef Pcte_sequence Pcte_link_names;
typedef Pcte_sequence Pcte_link_references;
typedef Pcte_sequence Pcte_message_types;
typedef Pcte_sequence Pcte_name_sequence;
typedef Pcte_sequence Pcte_object_criteria;
typedef Pcte_sequence Pcte_object_references;
typedef Pcte_sequence Pcte_type_names;
typedef Pcte_sequence Pcte_type_names_in_sds;
typedef Pcte_sequence Pcte_type_references;
typedef Pcte_sequence Pcte_user_criteria;
typedef Pcte_sequence Pcte_volume_infos;
```

```

typedef enum {
    PCTE_ACCOUNTING_FILE, PCTE_ACL,
    PCTE_AUDIT_FILE, PCTE_ATTRIBUTE_ASSIGNMENTS,
    PCTE_H_ATTRIBUTE_ASSIGNMENTS,
    PCTE_ATTRIBUTE_NAMES, PCTE_ATTRIBUTE_REFERENCES,
    PCTE_BUFFER, PCTE_CONFIDENTIALITY_CRITERIA,
    PCTE_ENUMERATION_VALUE_TYPE,
    PCTE_H_ENUMERATION_VALUE_TYPE,
    PCTE_ENUMERATION_VALUE_TYPE_IN_SDS,
    PCTE_GENERAL_CRITERIA, PCTE_INTEGRITY_CRITERIA,
    PCTE_KEY_TYPES, PCTE_H_KEY_TYPES,
    PCTE_KEY_TYPES_IN_SDS, PCTE_LINK_NAMES,
    PCTE_LINK_SET_DESCRIPTOR, PCTE_H_LINK_SET_DESCRIPTOR,
    PCTE_LINK_REFERENCES, PCTE_MESSAGE_TYPES,
    PCTE_NAME_SEQUENCE, PCTE_OBJECT_CRITERIA,
    PCTE_OBJECT_REFERENCES, PCTE_TYPE_NAMES,
    PCTE_TYPE_NAMES_IN_SDS, PCTE_TYPE_REFERENCES,
    PCTE_USER_CRITERIA, PCTE_VOLUME_INFOS
} Pcte_sequence_type;

typedef void *Pcte_sequence_element;

typedef void *Pcte_array_of_sequence_elements;

Pcte_error_type Pcte_sequence_create (
    Pcte_sequence_type          type,
    Pcte_array_of_sequence_elements data,
    Pcte_natural                count,
    Pcte_sequence               *sequence
);

Pcte_error_type Pcte_sequence_discard (
    Pcte_sequence               *sequence
);

Pcte_error_type Pcte_sequence_copy (
    Pcte_sequence               source_list,
    Pcte_sequence               *destination_list,
    Pcte_natural                index,
    Pcte_natural                source_index,
    Pcte_natural                count
);

Pcte_error_type Pcte_sequence_insert_elements (
    Pcte_sequence               sequence,
    Pcte_natural                index,
    Pcte_array_of_sequence_elements data,
    Pcte_natural                count
);

```

```
Pcte_error_type Pcte_sequence_delete (
    Pcte_sequence    sequence,
    Pcte_natural     index,
    Pcte_natural     count
);

Pcte_error_type Pcte_sequences_are_equal (
    Pcte_sequence    first_sequence,
    Pcte_sequence    second_sequence,
    Pcte_boolean     *equality
);

Pcte_error_type Pcte_sequence_get_index (
    Pcte_sequence    sequence,
    Pcte_sequence_element element,
    Pcte_integer     *index
);

Pcte_error_type Pcte_sequence_get_length (
    Pcte_sequence    sequence,
    Pcte_natural     *length
);

Pcte_error_type Pcte_sequence_get_elements (
    Pcte_sequence    sequence,
    Pcte_natural     index,
    Pcte_array_of_sequence_elements data,
    Pcte_natural     count
);

Pcte_error_type Pcte_sequence_get (
    Pcte_sequence    sequence,
    Pcte_natural     index,
    Pcte_sequence_element element
);

Pcte_error_type Pcte_sequence_insert (
    Pcte_sequence    sequence,
    Pcte_natural     index,
    Pcte_sequence_element element
);

Pcte_error_type Pcte_sequence_replace (
    Pcte_sequence    sequence,
    Pcte_natural     index,
    Pcte_sequence_element element
);
```

```

Pcte_error_type Pcte_sequence_append (
    Pcte_sequence      sequence,
    Pcte_sequence_element element
);

Pcte_error_type Pcte_sequence_normalize (
    Pcte_sequence      sequence
);

#endif /* !PCTE_SEQUENCES_INCLUDED */

```

9 Object management

```

/* The header <Pcte/oms.h> */

#ifndef PCTE_OMS_INCLUDED
#define PCTE_OMS_INCLUDED 1

#include <Pcte/types.h>
#include <Pcte/references.h>

```

9.1 Object Management Datatypes

```

typedef enum {
    PCTE_COMPOSITION      = 1<<0,
    PCTE_EXISTENCE        = 1<<1,
    PCTE_REFERENCE        = 1<<2,
    PCTE_DESIGNATION      = 1<<3,
    PCTE_IMPLICIT          = 1<<4
} Pcte_category;

typedef Pcte_natural Pcte_categories;

#define PCTE_ALL_CATEGORIES (Pcte_natural) (PCTE_COMPOSITION | \
PCTE_EXISTENCE | \
PCTE_REFERENCE | \
PCTE_DESIGNATION | \
PCTE_IMPLICIT)

typedef enum {
    PCTE_BOOLEAN_ATTRIBUTE,
    PCTE_INTEGER_ATTRIBUTE,
    PCTE_NATURAL_ATTRIBUTE,
    PCTE_FLOAT_ATTRIBUTE,
    PCTE_STRING_ATTRIBUTE,
    PCTE_TIME_ATTRIBUTE,
    PCTE_ENUMERATION_ATTRIBUTE
} Pcte_value_type;

```

```

/*      The PCTE datatype Value_type is a union which also contains the      */
/*      constituent type Enumeration_value_type, which is a sequence of      */
/*      Enumeral_type_nominator. For convenience, in the operations          */
/*      SDS_GET_ATTRIBUTE_TYPE_PROPERTIES and                                */
/*      WS_GET_ATTRIBUTE_TYPE_PROPERTIES an additional parameter              */
/*      is used to return an enumeration value type if necessary. In all other */
/*      cases it is sufficient to use Pcte_value_type.                        */

```

```

typedef struct {
    Pcte_value_type    type;
    union {
        Pcte_boolean   v_boolean;
        Pcte_integer    v_integer;
        Pcte_natural     v_natural;
        Pcte_float       v_float;
        Pcte_string      v_string;
        Pcte_time        v_time;
        Pcte_natural     v_enumeral_type_position;
    } value;
} Pcte_attribute_value;

```

```

typedef struct {
    Pcte_attribute_name    attribute;
    Pcte_attribute_value    value;
} Pcte_attribute_assignment;

```

```

typedef struct {
    Pcte_attribute_reference    attribute;
    Pcte_attribute_value        value;
} Pcte_h_attribute_assignment;

```

```

#include <Pcte/sequences.h>

```

```

typedef Pcte_sequence Pcte_attribute_assignments;

```

```

typedef Pcte_sequence Pcte_h_attribute_assignments;

```

```

/*      The PCTE datatype attribute assignments, which is a map from attribute */
/*      reference to a value type, is mapped to the sequence                  */
/*      Pcte_attribute_assignments indicated by                               */
/*      PCTE_ATTRIBUTE_ASSIGNMENT with the C element datatype                 */
/*      Pcte_attribute_assignment. The component attribute represents the      */
/*      domain of the map whereas the component value_type indicates the      */
/*      value of the map. If a parameter of type Pcte_attribute_assignments is */
/*      passed as an input parameter to an operation and if there is more than */
/*      one entry in the sequence with the same value of the component         */
/*      attribute, only the first of these entries is recognized in the operation. */

```

```

typedef enum {
    PCTE_INTERNAL_LINKS, PCTE_EXTERNAL_LINKS,
    PCTE_ALL_LINKS
} Pcte_link_scope;

typedef enum {
    PCTE_EQUAL_TYPE, PCTE_ANCESTOR_TYPE,
    PCTE_DESCENDANT_TYPE, PCTE_UNRELATED_TYPE
} Pcte_type_ancestry;

typedef enum {
    PCTE_ANCESTOR_VSN, PCTE_DESCENDANT_VSN,
    PCTE_SAME_VSN, PCTE_RELATED_VSN,
    PCTE_UNRELATED_VSN
} Pcte_version_relation;

typedef enum {
    PCTE_ATOMIC, PCTE_COMPOSITE
} Pcte_object_scope;

typedef enum {
    PCTE_ACCESSIBLE, PCTE_INACCESSIBLE, PCTE_UNKNOWN
} Pcte_volume_accessibility;

#include <Pcte/devices.h>

typedef struct {
    Pcte_volume_identifier    volume;
    Pcte_volume_accessibility mounted;
} Pcte_volume_info;

typedef struct {
    Pcte_object_reference    origin;
    Pcte_link_names          links;
} Pcte_link_set_descriptor;

typedef struct {
    Pcte_object_reference    origin;
    Pcte_link_references     links;
} Pcte_h_link_set_descriptor;

#define PCTE_MAX_EXACT_IDENTIFIER_SIZE PCTE_MAX_KEY_SIZE

typedef Pcte_octet
    Pcte_exact_identifier[PCTE_MAX_EXACT_IDENTIFIER_SIZE + 1];

#include <Pcte/discretionary.h>

```

9.2 Link Operations

/* 9.2.1 LINK_CREATE */

```
Pcte_error_type Pcte_link_create (
    Pcte_object_reference    origin,
    Pcte_link_name          new_link,
    Pcte_object_reference    dest,
    Pcte_key                reverse_key
);
```

```
Pcte_error_type Pcte_h_link_create (
    Pcte_object_reference    origin,
    Pcte_link_reference      new_link,
    Pcte_object_reference    dest,
    Pcte_key                reverse_key
);
```

/* The effect of not providing the optional parameter *reverse_key* to the */
 /* abstract operation is achieved by specifying **reverse_key** as NULL. */

/* 9.2.2 LINK_DELETE */

```
Pcte_error_type Pcte_link_delete (
    Pcte_object_reference    origin,
    Pcte_link_name          link
);
```

```
Pcte_error_type Pcte_h_link_delete (
    Pcte_object_reference    origin,
    Pcte_link_reference      link
);
```

/* 9.2.3 LINK_DELETE_ATTRIBUTE */

```
Pcte_error_type Pcte_link_delete_attribute (
    Pcte_object_reference    origin,
    Pcte_link_name          link,
    Pcte_attribute_reference attribute
);
```

```
Pcte_error_type Pcte_h_link_delete_attribute (
    Pcte_object_reference    origin,
    Pcte_link_reference      link,
    Pcte_attribute_reference attribute
);
```

```
/* 9.2.4 LINK_GET_ATTRIBUTE */
```

```
Pcte_error_type Pcte_link_get_attribute (  
    Pcte_object_reference    origin,  
    Pcte_link_name          link,  
    Pcte_attribute_name      attribute,  
    Pcte_attribute_value     *value  
);
```

```
Pcte_error_type Pcte_h_link_get_attribute (  
    Pcte_object_reference    origin,  
    Pcte_link_reference      link,  
    Pcte_attribute_reference attribute,  
    Pcte_attribute_value     *value  
);
```

```
/* 9.2.5 LINK_GET_DESTINATION_VOLUME */
```

```
Pcte_error_type Pcte_link_get_destination_volume (  
    Pcte_object_reference    origin,  
    Pcte_link_name          link,  
    Pcte_volume_info        *volume_info  
);
```

```
Pcte_error_type Pcte_h_link_get_destination_volume (  
    Pcte_object_reference    origin,  
    Pcte_link_reference      link,  
    Pcte_volume_info        *volume_info  
);
```

```
/* 9.2.6 LINK_GET_KEY */
```

```
Pcte_error_type Pcte_link_get_key (  
    Pcte_object_reference    origin,  
    Pcte_link_name          link,  
    Pcte_key                 key  
);
```

```
Pcte_error_type Pcte_h_link_get_key (  
    Pcte_object_reference    origin,  
    Pcte_link_reference      link,  
    Pcte_key                 key  
);
```

```
/* 9.2.7 LINK_GET_REVERSE */
```

```
Pcte_error_type Pcte_link_get_reverse (  
    Pcte_object_reference    origin,  
    Pcte_link_name          link,  
    Pcte_link_name          reverse_link,  
    Pcte_object_reference    *dest  
);
```