Dokument Typ:
Document Type: Standard

Titel:
Title:

User Control Language (UCL) Reference Manual

Lieferbedingungs–Nr.:
DRL/DRD No.:

Klassifikations Nr.:
Classification No.:

Produktgruppe:
Product Group:

Konfigurationsteil–Nr.:
Configuration Item No.:

Schlagwörter:
Headings: UCL
User Control Language
AP
Automated Procedure

Porduktklassifizierungs–Nr.:
Classifying Product Code:

Freigabe Ordnungs–Nr.:
Release Orde No.:

Bisherige Dok.–Nr.:
previous doc.–no.: CGS–RIBRE–STD–0001

Bearbeitet: Prepared by:	Franz Kruse	**Org. Einh.:** Orgin. Unit:	TE 55	**Unternehmen:** Company:	EADS Astrium Bremen
Geprüft: Agreed by:	Stephan Marz	**Org. Einh.:** Orgin. Unit::	TE 55	**Unternehmen:** Company:	EADS Astrium Bremen
Genehmigt: Approved by:	Jürgen Frank	**Org. Einh.:** Orgin. Unit:	TE 55	**Unternehmen:** Company:	EADS Astrium Bremen
Genehmigt: Approved by:		**Org. Einh.:** Orgin. Unit:		**Unternehmen:** Company:	

DOCUMENT CHANGE RECORD

ISSUE/REV.	DATE	Affected Paragraph/Page	DESCRIPTION OF CHANGE
1/–	2002–02–01		First version including: – 32–bit UNSIGNED_INTEGER type. – Database aliases (”nicknames”) – High resolution TIME/DURATION format
1/A	2002–06–03	4.9.4.3 4.3.4, 4.7 4.13.4 4.5.2.1, 4.5.2.5, 4.11, 4.12	String conversions Qualified identifier syntax enhanced to better fit alias usage Derived Values can access their own (old) values. Different editorial changes COL–RIBRE–SPR–10651 New function LOW, generalized HIGH
1/B	2003–01–15	4.12	Generalized Min/Max functions
2/–	2004–05–19	4.14 4.1.3	Privileges, authorization I/O format
2/A	2004–09–01	4.3.4 4.9.4.4 4.13.2	Qualified predefined identifiers (`.xyz`) I/O format Imports in library spec. available in body
2/B	2004–12–14	4.14	Privileges inherited on subprogram level
2/C	2005–02–20	4.9.4.3	Extended string conversions
2/D	2005–06–08	4.8.2, 4.9.4.1	Counting units
2/E	2005–08–30	4.9.4.3	unit string conversion for pathname types
3/–	2005–12–01	4.8 4.8, 4.9.4.1	declaration of new base units unitized integer types
3/A	2008–07–02	4.13.4	declaration of new base units
4/–	2009–02–01	4.5.2.5 4.12 Appendix H	Substrings (slices) Predefined procedure PUT Constraints removed
5/–	2010–01–29	4.1, 4.4 4.16.2 4.1.2.2	Annotations Explicitly numbered system library subprograms New keywords union, entity, void

Table of Contents

1 Introduction

1.1 Identification

This is the Language Reference Manual for the CGS User Control Language (UCL), Document CGS–RIBRE–STD–0001. The first issue is derived from the Columbus document COL–RIBRE–STD–0010, issue 4/A. Changes to this base document are marked with change bars.

1.2 Purpose

This document provides the language definition for UCL. Its primary goal is to establish formal requirements on syntax and semantics of UCL for the development of language processing tools such as compiler and interpreter. It may also be used as a language reference manual.

Please note that this reference manual covers only the UCL language itself, it does not define system libraries for specific target systems. The definition of UCL system libraries is part of the interface documentation of the respective target systems, e. g. the CGS ICD, DMS ICD etc.

1.3 Document Outline

This reference manual is divided into main chapters (3 and 4), and several appendices which, for the most part, summarise information provided elsewhere in the document.

Chapter 3 gives a general overview; it briefly explains the underlying concept, and puts UCL in perspective, showing how it fits into the CGS language scenario (UCL, HLCL).

Chapter 4 deals with the basic language definition. It is organised into sections, each covering a specific language element. This is first explained in narrative form, followed by the formal syntax (using a variant of the Backus–Naur Form) and finally illustrated by one or more examples.

The appendices (A through H) have the following contents:

- Appendix A and B explain the acronyms and terms used.
- Appendix C is empty.
- Appendix D summarizes the syntax of UCL.
- Appendix E shows the ASCII character set.
- Appendix F shows correspondences between MDB item types and UCL types.
- Appendix G shows the ISO 1000 engineering units.
- Appendix H summarizes issues of implementation.

At the end of the document there is an alphabetical index.

2 Applicable and Reference Documents

2.1 Applicable Documents

none

2.2 Reference Documents

2.2.1 High Level Command Language (HLCL) Reference Manual
CGS–RIBRE–STD–0002

2.2.2 UCL Virtual Stack Machine and I–Code Reference Manual
CGS–RIBRE–STD–0003

2.2.3 Mission Database (MDB) User Manual
There are specific MDB user manuals for different target systems.

2.2.4 ISO 1000, SI units and recommendations for the use of their multiples and of certain other units
International Standards Organisation, Geneva, Switzerland

2.2.5 ISO 646, Information Processing ISO 7–Bit Coded Character Set for Information Interchange
International Standards Organisation, Geneva, Switzerland

3 Overview

3.1 Conceptual Overview

Although its basic syntax is that of a general–purpose programming language, UCL is a dedicated test and operations language for monitoring and control of spacecraft subsystems. It is intended for use in both the on–board operational and the ground (checkout) environment.

UCL is a procedural language representing the set of all commands or instructions that can be predefined and stored as so–called *Automated Procedures* (APs) and *User Libraries* in the Mission Database (MDB). Automated procedures play the role of a *main program*. They can be executed like an individual program, but can, typically, also be part of some higher–level *actions* which are stored in the form of hierarchical tree structures.

For better manageability and adaptability to specific target environments, the monitoring and control features of UCL have been removed from the actual language definition and encapsulated in *system libraries*. These libraries are specific for the respective target systems, they are defined in the interface documentation for these systems (e. g. the DMS ICD, the CGS ICD etc.).

The *library* construct of UCL corresponds to the Ada *package* (or the *module* in Modula–2). It supports the information hiding concept which allows the separation of a module's specification from its implementation. The library concept is not restricted to system libraries, but a user may implement his own *user libraries* in UCL.

UCL programs (Automated Procedures, or APs) automatically "inherit" the definitions of objects contained in the MDB. These objects are thus directly visible and may be referenced by their Database path names (see Fig. 1.)

UCL also supports *on–line interactive* commands. These facilities are provided by the *High Level Command Language* (HLCL*)* in the ground SW environment. HLCL is somewhat a modification of UCL (i.e. it extends UCL with respect to interactive commanding, e.g. HLCL allows abbreviations) and shares the same system libraries. In this manner, UCL's monitoring and control facilities also become available *on–line*, as HLCL keyboard commands. HLCL is described in reference document 2.2.1.

UCL programs are edited and compiled *off–line*. This process is depicted in Figure 3. During the compilation process, the UCL code is transformed into a binary intermediate code which is later executed (interpreted) in the target environment (e.g. DMS, EGSE) by a dedicated program (*interprete*r).

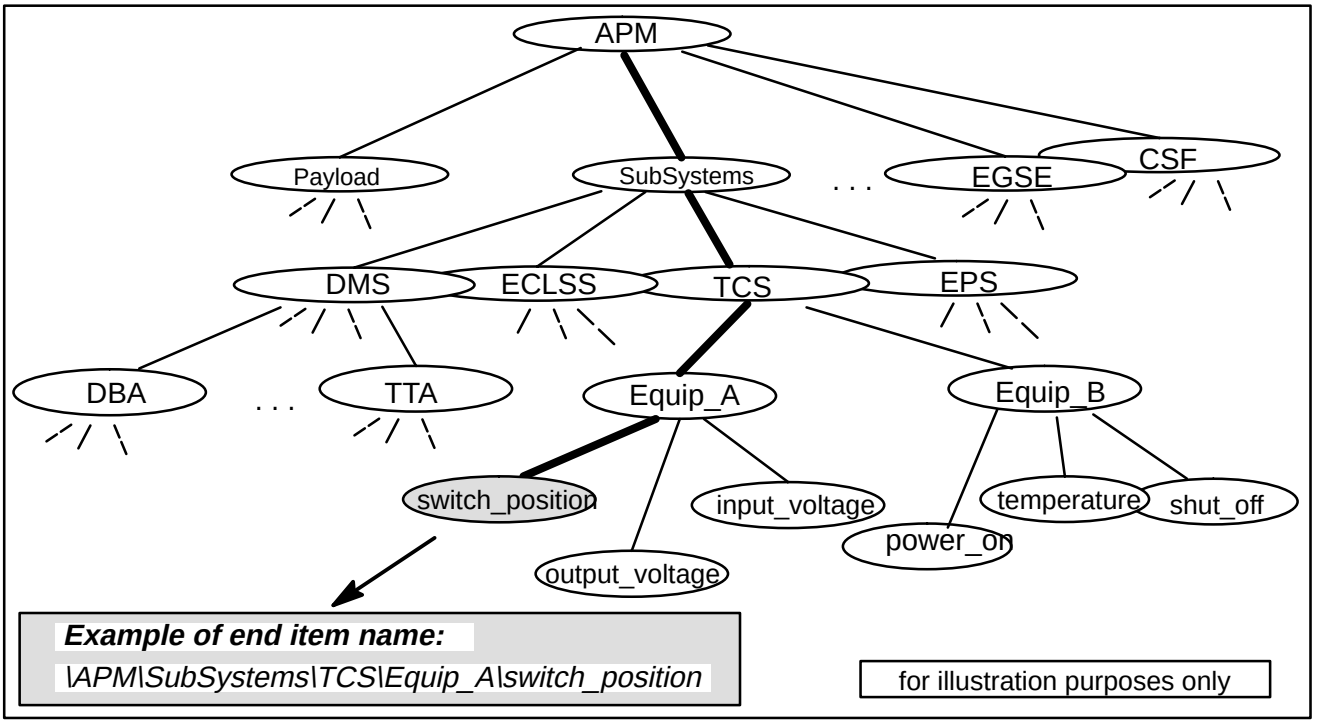

Figure 1. Hierarchical Name Tree (example)

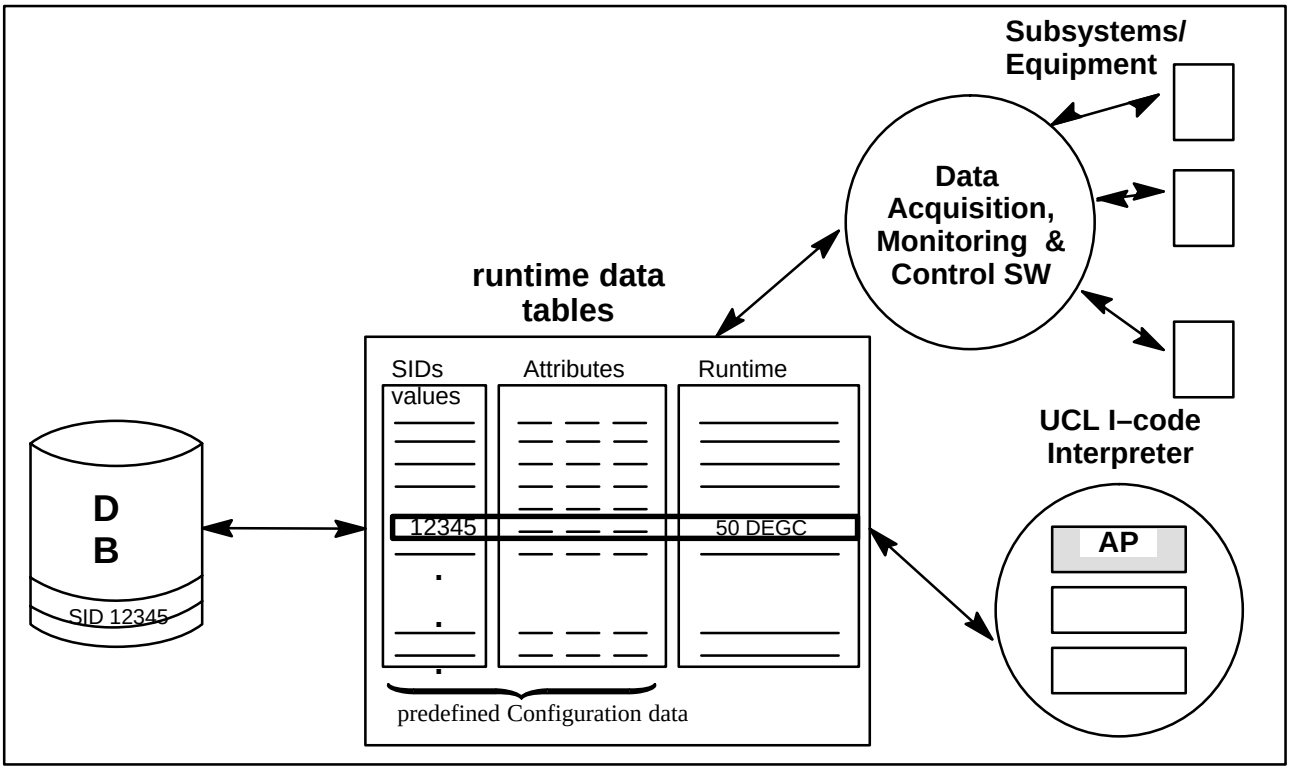

Figure 2. UCL runtime environment

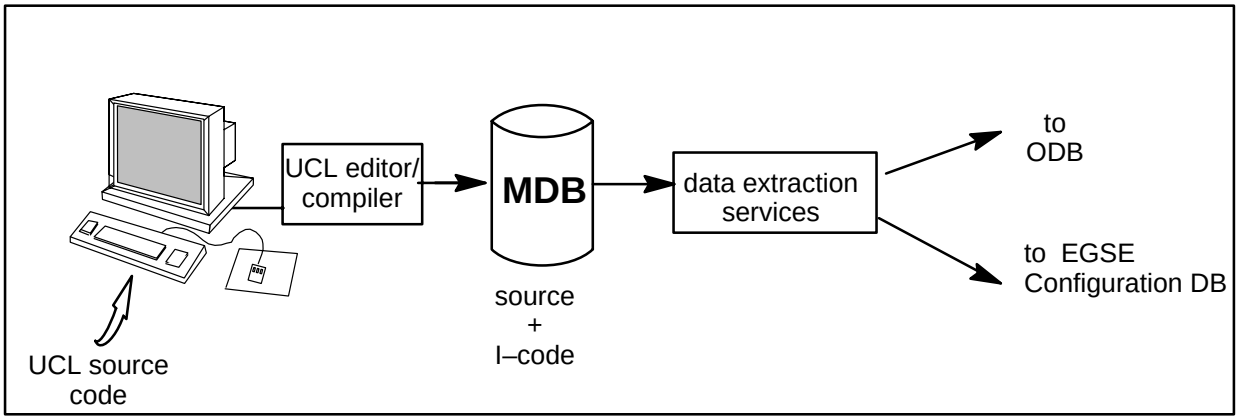

Figure 3. Off–line preparation of an AP

3.2 Language Summary

- Compilation Units
  - Automated Procedures (main program)
  - Libraries
  - Formal parameter list definitions
  - Derived Values
- A Library consists of :
  - Library specification
  - Library body (implementation part)
- Automated Procedures, Library specifications, and Library bodies are compiled separately.
- Two kinds of subprograms:
  - Procedures
  - Functions
- Subprograms must be defined within main programs (APs) or Libraries; they may not be nested inside other subprograms. Their parameters may have default values, and may be ”unbound” arrays or strings. Parameter associations are either named or positional. Subprograms may be called recursively.
- Predefined Subprograms
  - `ABS, MAX, MIN, HIGH, LENGTH, ODD, INC, DEC, INCL, EXCL, ...`
- All I/O (Monitoring & Control operations) via target–specific system library routines.
- Access to database items as global objects via their path name.
- Support for physical measurement units.

- Statements
  - Assignments, Procedure calls, Function calls
  - Conditional statements:
    - **`if`** statement with **`elsif`** and **`else`** clauses
    - **`case`** statement with **`when`** and **`else`** clauses
  - Iteration:
    - general **`loop`** statement with **`exit`**;
    - **`repeat`**, **`while`** and **`for`** loops
  - Transfer of Controls
    - **`halt`**, **`return`**, **`exit`**
- Declarations.
  - **`constant`**, **`variable`**, **`type`**, **`unit`** and **`alias`** declarations
  - Declaration of variables is mandatory.
  - Objects must be declared before they are referenced.
- Certain keywords are reserved and cannot be used as identifiers.
- The semicolon (`;`) is a statement terminator (not a separator).
- Data types
  - predefined:
    Integers, real numbers (with single and double precision), Booleans, enumerated types, state codes, times, durations, sets, character strings, byte strings, arrays, records, pathnames and low level types for bytes, words and long words
  - user–defined
- Arrays and strings may be of arbitrary dimension with arbitrary bounds; array bounds are constants (i.e. no dynamic arrays, except in parameter list of procedures and functions). Strings may vary in length, up to a fixed, user declared upper limit.
- Operators: `+` , `–` , `*` , `/` , `**` , `%` , `&` , `|` , `~` , `<` , `<=` , `=` , `>=` , `>` , `<>` , **`in`**
- Implementation restrictions:
  Several restrictions are indirectly imposed on the language through the I–code and symbol table definition. These restrictions are summarized in Appendix H.

3.3 Syntax Notation

Throughout this manual the syntax of UCL is described in an Extended Backus–Naur Form:

- The symbol = (equal sign) separates a syntactic class from its definition.
- Terminal symbols, i.e. literals, are enclosed in double quotes. If the quote character appears as a literal itself, it is written twice.
- Braces (curly brackets) denote repetition, i.e. the enclosed item may appear zero or more times; e.g. {A} means 0 or more occurrences of A.
- Square brackets enclose optional items, i.e. the enclosed item may appear once or be omitted; e.g. [A] means 0 or 1 occurrence of A.
- A vertical bar separates alternative items; e.g: A | B means either A or B, but not both.
- Parentheses are used for grouping; e.g: (a|b)c stands for ac | bc.
- Each production rule is terminated by a period.

A complete syntax summary is given in appendix D.

3.4 Conventions Used in Examples

UCL source text in examples is written in a mono–spaced font (Courier). Strict conventions are followed for the representation of word classes:

- UCL reserved words are written in bold and all lower–case (**`begin`**, **`if`**, **`case`** etc.).
- Predefined identifiers are written in all upper–case (`INTEGER`, `REAL`, `INCL`, `DECL` etc.).
- Other identifiers are written with upper–case initial letters for each word part (e.g. `File_Name`, `Min_Value`).

Control structures are uniformly indented according to the logical program structure.

4 Language Definition

4.1 Vocabulary & Lexical Elements

4.1.1 Character Set

The text of a UCL program (also called the *source code)* is a sequence of *lexical elements* (or *tokens*), each consisting of one or more *characters*. These characters may be any of the ASCII graphic symbols defined in the ISO standard 646.

The UCL source code can be freely broken into lines whereby the length of the line is restricted to 256 characters. (This is a restriction imposed by the particular implementation of the UCL compiler; see Appendix H.)

Adjacent lexical elements may be separated by one or more special characters (e.g. blank space) called *separators*. Separators are allowed between any two tokens, and also before the first and after the last token; in particular, the end of a line is a separator. Within lexical elements separators are not allowed, except if a separator is part of the element (character and string literals).

A separator is mandatory whenever its absence would result in an ambiguous token sequence. For example, it is not required between identifiers and non–alphanumeric symbols (e.g. the arithmetical operators), but it is required between identifiers and keywords or numeric constants.

Thus, `A=B+C` is equivalent to `A = B + C`. However, `IFA=B THEN ...`
is invalid because of the missing separator between the keyword **`if`** and the identifier `A`.

The UCL separators are:

- Blank space
- Horizontal tabulator
- Vertical tabulation character LF (line feed) and VT (vertical tabulator)
- End of line
- Page separator (ASCII form–feed (FF) character)
- Annotation
- Comment

These ASCII graphic symbols (printable characters) and the separator symbols are the only characters allowed in UCL source code. The other non–graphic symbols (so–called control characters), e.g. backspace or escape, are not allowed except in *comments* (see below), and are rejected by the UCL compiler.

A *comment* begins with two consecutive hyphens (`--`) and terminates with the end of the line. Note: the contents of a comment are not interpreted. Hence, a comment may contain any characters (even those prohibited outside a comment).

An *annotation* is a special comment that can be attached to declared items. It begins with an annotation marker (`<-`) and terminates with the end of the line. Unlike a comment, an annotation is not ignored by the compiler, but kept as a textual description of the item. It can be displayed in an HLCL command window as part of the help information for the item. For a description of annotations see 4.4.

4.1.2 Lexical Elements

The text of a UCL program may consist of the following lexical elements: delimiters, identifiers (predefined and user–defined), and literals (numeric, character, string, state code, time and unit literals).

4.1.2.1 Delimiters

Delimiters are either one character or two consecutive characters, used as punctuation symbols or as operators. They are listed below.

```
(    left parenthesis
)    right parenthesis
*    asterisk (multiplication operator)
**   exponentiation operator
+    plus (addition operator)
,    comma
-    minus (subtraction operator)
.    dot
..   double dot (range symbol)
/    slash (division operator)
%    percent (modulus operator)
&    logical AND
|    logical OR
~    logical NOT
:    colon
:=   assignment operator
;    semicolon (also used as statement terminator)
<    less than
<=   less than or equal
<>   not equal
=    equal
>=   greater than or equal
[    left bracket
]    right bracket
{    left brace
}    right brace
<-   annotation marker
--   comment marker
```

4.1.2.2 Identifiers

Identifiers are the names used in a UCL program to designate various UCL entities, such as constants, variables, types, etc. An identifier begins with a letter which may be followed by any combination of letters, digits and underscore characters "_". Not allowed are consecutive underscore characters, "a__b", and underscore character at the end of a name, "abc_".

UCL is not case sensitive (i.e. does not distinguish between upper– and lower case letters). Thus the three identifiers SENsor_A, SENSOR_A and sensor_a are equivalent.

Since "end of line" is a separator, an identifier must fit on one line. The maximum length of an identifier is thus restricted to 256 characters. All characters in an identifier are significant.

Formal syntax

```
Identifier      = Letter { [ "_" ] Letter_Or_Digit }
Letter_Or_Digit = Letter | Digit
```

Examples

```
Valve_nr_5
Sensor_12
An_example_of_a_very_long_identifier
```

Reserved words

Some identifiers are *reserved words*, having a special meaning in the language. They cannot be used to denote user–defined entities (e.g. variables, constants). The following identifiers are reserved words in UCL:

```
alias     array     begin     body      by        case      constant
do        else      elsif     end       entity    exit      for
function  guarded   halt      if        import    in        library
loop      of        out       pathname  procedure record    repeat
return    sequence  set       statecode string    then      to
type      union     unit      until     variable  void      when
while
```

4.1.2.3 Path Identifiers and Pathnames

A Database object is denoted in UCL by its *pathname* (see also 4.8.3, Pathname Types). Syntactically, a pathname consists of a sequence of *path identifiers* corresponding each to a *level* (or node) in the hierarchical nametree. A path identifier consists of either one backslash (the *root pathname*), or two consecutive backslash characters (*no pathname*), or an identifier prefixed by a backslash character "\". The syntax of a path identifier is less strict than the normal identifier syntax. The nametree design restricts the length of a path identifier to 16 characters (not including the backslash).

Formal syntax

```
Path_Identifier =  "\" ( Letter | "_" | Digit ) { Letter | "_" | Digit }
```

Examples of path identifiers

```
\              (root pathname)
\\             (no pathname)
\APM
\EQUIPMENT_Y
```

Pathnames are formed by a sequence of path identifiers with no spaces between them. The pathname \APM\DMS\ASSEMBLY_X\UNIT_Y consists of 4 consecutive path identifiers.

4.1.2.4 Numeric Literals

In the UCL source code, numeric literals (constants) may take three possible forms: *integer* (simple integer), *based integer* or *real* numbers.

4.1.2.4.1 Integers

An integer is simply a sequence of digits (’0’ .. ’9’). The underscore character (’_’) may be used to logically group digits, it has no effect on the numeric value of the literal. The value of an integer must be in the range: `0 .. MAX(UNSIGNED_INTEGER)`. The compiler issues an error message if this range constraint is violated.

Formal syntax

Simple_Integer = Digits
Digits = Digit { [”_”] Digit }
Digit = ”0” | ”1” | ”2” | ”3” | ”4” | ”5” | ”6” | ”7” | ”8” | ”9”

Examples

```
5, 123, 027, 10_000
```

4.1.2.4.2 Based Integers

A based integer has the form:

Base “#” Value “#”

where *Base* is a decimal number indicating the base of the numbering system, it must be in the range 2 .. 16. Some bases are particularly useful: 2 for binary, 8 for octal, 10 for decimal (the default), or 16 for hexadecimal. *Value* must be a sequence of ”extended digits” (i.e. ’0’ .. ’9’ or the letters ’A’ .. ’F’, which in hexadecimal notation correspond to the decimal values 10 .. 15, respectively). Further, for a given base B, each ”extended digit” must be in the range: 0 to B–1.

The compiler generates an error if the value of the specified number is not in the range `0 .. MAX(UNSIGNED_INTEGER)`.

Formal syntax

Based_Integer = Digits ”#” Hex_Digit { [”_”] Hex_Digit } ”#”
Digits = *see 4.1.2.4.1*
Hex_Digit = Digit | ”A” | ”B” | ”C” | ”D” | ”E” | ”F”

Examples of based integers, all having the value 255:

```
2#1111_1111#
8#377#
16#FF#
10#255#
```

4.1.2.4.3 Real Numbers

A real number may be built as:

Formal syntax

Real = Digits "." Digits ["E" ["+" | "–"] Digits]

Digits = *see 4.1.2.4.1*

The exponent indicates the power of ten by which the value of the decimal literal is to be multiplied. Thus the value of a real number can be obtained as:

```
Decimal_Literal * 10 ** Exponent
```

The compiler generates an error if the value of the specified number is not a valid floating–point number (see predefined type `LONG_REAL`, 4.8.1.4).

Examples of real numbers:

```
2.575   5.8e3   300.6E+05   0.1E23   1.2e–3   0.000_000_1
```

4.1.2.5 Statecode Literals

In the MDB, a statecode is a literal constant identifying one specific *state* (e.g. Open, Closed) of a "discrete–type" MDB object. When such an object is created (like a switch or a valve), its allowable states are also defined and appropriate symbolic names (or *statecodes*) assigned to them.

In UCL, each statecode identifier must be prefixed by a dollar sign ("$"). The following identifier may be up to 8 characters long (not including the $ sign). A special statecode literal is the constant $$, which stands for an undefined statecode value.

Formal syntax

Statecode = "$" Identifier | "$$"

Identifier = *see 4.1.2.2*

Examples of statecode literals:

```
$OFF   $ACTIVE   $OPEN   $$
```

4.1.2.6 Character Literals

A character literal consists of any one of the 95 graphic ASCII characters (including the space character) enclosed between single quotes (apostrophe). The single quote character itself is represented in a similar manner, i.e. also enclosed between single quotes.

Examples of character literals:

```
' '   'a'   '1'   '.'   '''
```

4.1.2.7 String Literals

Strings fall in two classes: *character strings* and *byte strings*.

4.1.2.7.1 Character Strings

A character string is a sequence of zero or more characters from the 95 graphic ASCII characters (including the space) enclosed between quotation marks (double quote characters). The quotation mark itself has to be doubled if it appears within the string. The empty string is denoted by two adjacent quotation marks (`""`). Since end–of–line is a separator, a string must not extend over the end of a line. The maximum length of a string literal, including the quotes, is thus 256 characters.

Character string literals are mapped to the predefined **`string`** type (4.8.2.5). Characters in a string are packed with one character per byte.

Formal syntax

Char_String = """ { ASCII | """" """" } """"

Character = """ ASCII """

ASCII = *any of the ASCII Characters in range 32 .. 126*

Examples of character string literals:

```
"String 1"
"String with ""quoted"" word "
""    (empty string)
```

4.1.2.7.2 Byte Strings

A byte string is a sequence of zero or more bytes. A byte string literal is written in hexadecimal form, enclosed in quotation marks, like a character string, prefixed with #. The number of hexadecimal digits must always be even. The empty byte string is denoted as `#""`. Like for character strings, the maximum length of a byte string literal, including the quotes, is restricted to 256 characters.

Underscore characters (`'_'`) and blank characters (`' '`) may be used to logically group the bytes in a string. They have no effect on the value of the literal.

Byte string literals are mapped to the predefined **`string of`** BYTE type (4.8.2.5). Bytes in a string are packed with four bytes in a 32–bit word.

Formal syntax

Byte_String = "#" """ [Hex_Digit Hex_Digit { ["_" | " "] Hex_Digit Hex_Digit }] """

Hex_Digit = *see 4.1.2.4.2*

Examples of byte string literals:

```
#"0000_0001 0000_0002 0000_0003"
#"FF 0102_AE_F1 0103_E0_00"
#"01 02 03 04 05 06 07 08 09 0A"
#""   (empty string)
```

4.1.2.8 Time Literals

A time literal consists of two separate lexical elements, date and hour specification. If the date is omitted, UCL ignores it in any related time operation. Other omitted parts are assumed as zero. A special time literal is the constant `~:~`, which stands for an undefined time value. It can only be used in assignments and as parameters and tested for equality, no further operations are defined on it.

The date is restricted to the range from year 1901 to year 2099.

Formal syntax

Time_Literal	=	Date [Time] | Time
Date	=	Day ”.” Month ”.” Year
Day	=	[Digit] Digit
Month	=	[Digit] Digit
Year	=	Digit Digit Digit Digit
Time	=	Hours ”:” Minutes [”:” Seconds [”.” Fraction]] | ”~:~”
Hours	=	[Digit] Digit
Minutes	=	Digit Digit
Seconds	=	Digit Digit
Fraction	=	Digits
Digits	=	*see 4.1.2.4.1*
Digit	=	*see 4.1.2.4.1*

Examples

`30.05.1992 12:34:17.48`	date and time
`13:30`	time only (date is ignored)
`24.12.1991`	date only (at 00:00 h)
`~:~`	undefined time

4.1.2.9 Duration Literals

A duration literal is written as a real number with a time unit, e.g. seconds, minutes or hours. Unit literals are described in 4.1.2.10, the unit concept is described in 4.11.

Examples (each representing a duration of one hour):

`3600.0 [s]`	in seconds
`60.0 [min]`	in minutes
`1.0 [h]`	in hours

4.1.2.10 Unit Literals

A unit literal denotes a physical measurement unit. It is enclosed in square brackets. The unit expression follows the ISO 1000 conventions (see section 4.11 and reference document 2.2.4). Exponentiation is denoted by placing the exponent directly behind a unit identifier with no blanks between them, e. g. `[m2]` stands for m^2. Note that, in contrast to ordinary identifiers, unit identifiers used in unit expressions (`kg`, `m`, `A`, `MeV` etc.) are case sensitive.

Formal syntax

Unit	=	”[” Unit_Expression ”]”
Unit_Expression	=	[Numerator [”/” Denominator] [”+” Offset | ”–” Offset]]
Offset	=	Number [”/” Number]
Numerator	=	Unit_Term | ”(” Unit_Term ”)”
Denominator	=	Number | Unit_Factor | ”(” Unit_Term ”)”
Unit_Term	=	[Number] Unit_Factor { Unit_Factor } | Number
Unit_Factor	=	Unit_Identifier { Digit }
Unit_Identifier	=	Letter { Letter }
Number	=	Simple_Integer | Based_Integer | Real
Simple_Integer	=	*see 4.1.2.4.1*
Based_Integer	=	*see 4.1.2.4.2*
Real	=	*see 4.1.2.4.3*

Examples

```
[kg]
[kg m/s]
[A s]
[N m/s2]
```

4.2 Import

Import is used to make objects declared in other compilation units (e.g. libraries) available to the current compilation unit. Importable modules are libraries, automated procedures and parameterized MDB items. The module is given by its *pathname*, see section 4.3.6.

When imported, all identifiers exported by the imported module (units, constants, types, variables, aliases, procedures, and functions) become visible throughout the compilation unit. For the different types of modules, the exported part is

- for libraries: the library specification,
- for automated procedures: the items declared before the AP header, the implicit AP alias and the parameter list, not the items within the AP,
- for parameterized MDB items (formal parameter lists): the items declared together with the parameter list, and the implicit alias, if defined.

An imported identifier will be hidden, if

- the same identifier is imported from more than one module. The conflicting imported identifiers are then hidden throughout the compilation unit.
- the same identifier is declared within the importing compilation unit. The imported identifier is then hidden throughout the scope of the local identifier.
- there is an identical predefined identifier. The imported identifier is then hidden throughout the compilation unit.

An imported identifier can thus never hide a predefined identifier or an identifier declared in the compilation unit. Imported identifiers, even if hidden, can always be accessed with a *qualified identifier,* see section 4.3.4.

Formal Syntax

Import= ”import” Name ”;”

Name = *see 4.3.4*

Example

```
import \APM\ONBOARD\DMS\SYSLIB;
import GROUND_LIBRARY;          -- import via alias (”nickname”)
```

4.3 Declarations, Names and Scopes

4.3.1 Identifiers

All UCL objects (i.e. constant, variable, type, procedure etc.), with the exception of the predefined standard identifiers, Mission Database objects and imported objects, must be explicitly introduced via an object declaration before they can be used in the program. The object declaration associates the object with a name, and at the same time establishes the object's attributes or properties. Declaration of all UCL objects is described in the following sections. Once an object has been declared, it may be referenced by its name or *identifier.* An identifier used in a declaration must have been declared in a previous declaration, i. e. an identifier must not be used in its own declaration.

Within the same *scope*, all identifiers must be unique.

4.3.2 Scope of Identifiers

A UCL program (AP) may be structured, it may contain subprograms (procedures and functions) which are nested within the main program. The main program, as well as each subprogram, form a *scope* which may contain its own set of locally declared objects. Parameters belong to the local scope of the AP or subprogram.

The scope of an identifier is the section of a program or subprogram in which the object denoted by the identifier exists. In general, the scope of an object extends from its declaration to the end of the block in which it is declared (see figure 4. below). Whithin its scope, an identifier is *visible* and can be used:

- An identifier declared in an outer scope (e. g. an AP) may be redeclared in an inner scope (a subprogram). In this case two different objects are denoted by the same identifier, and the identifier declared in the inner scope hides the same identifier declared in the outer scope. Within the inner scope, the hidden variable from the outer scope still exists but is not visible and cannot be accessed.
- The predefined identifiers (`INTEGER`, `REAL`, `BOOLEAN` etc.) belong to a global scope outside the compilation unit. Like any other identifiers, they may be redeclared within inner scopes (main program or subprogram). This will hide the predefined identifier for the rest of the block. Redeclarations of predefined identifiers are legal, but usually they obscure the program and should be avoided.
- The scope of record field names is the record. Since record field names are always written in dot notation, prefixed with the name of the record variable, there is never a name conflict.
- The scope of a **`for`** loop variable is the body of the **`for`** loop (see 4.13.5.4). Since **`for`** loops may be nested, the loop variable of an inner loop may hide a loop variable of an outer loop, and loop variables may hide identifiers declared outside the loop.

Libraries and other importable modules have their own scope. The scope of an identifier declared in a library specification comprises both the specification (up from its declaration) and the body of the library. Identifiers declared in a library body are visible in the body only. When imported, all identifiers declared in the specification of the imported module become visible throughout the compilation unit (see 4.2).

Note that unit identifiers, such as `[kg]`, `[A]`, `[MeV]` etc., are not ordinary identifiers. They are not bound to scopes, but are always global, see 4.11.

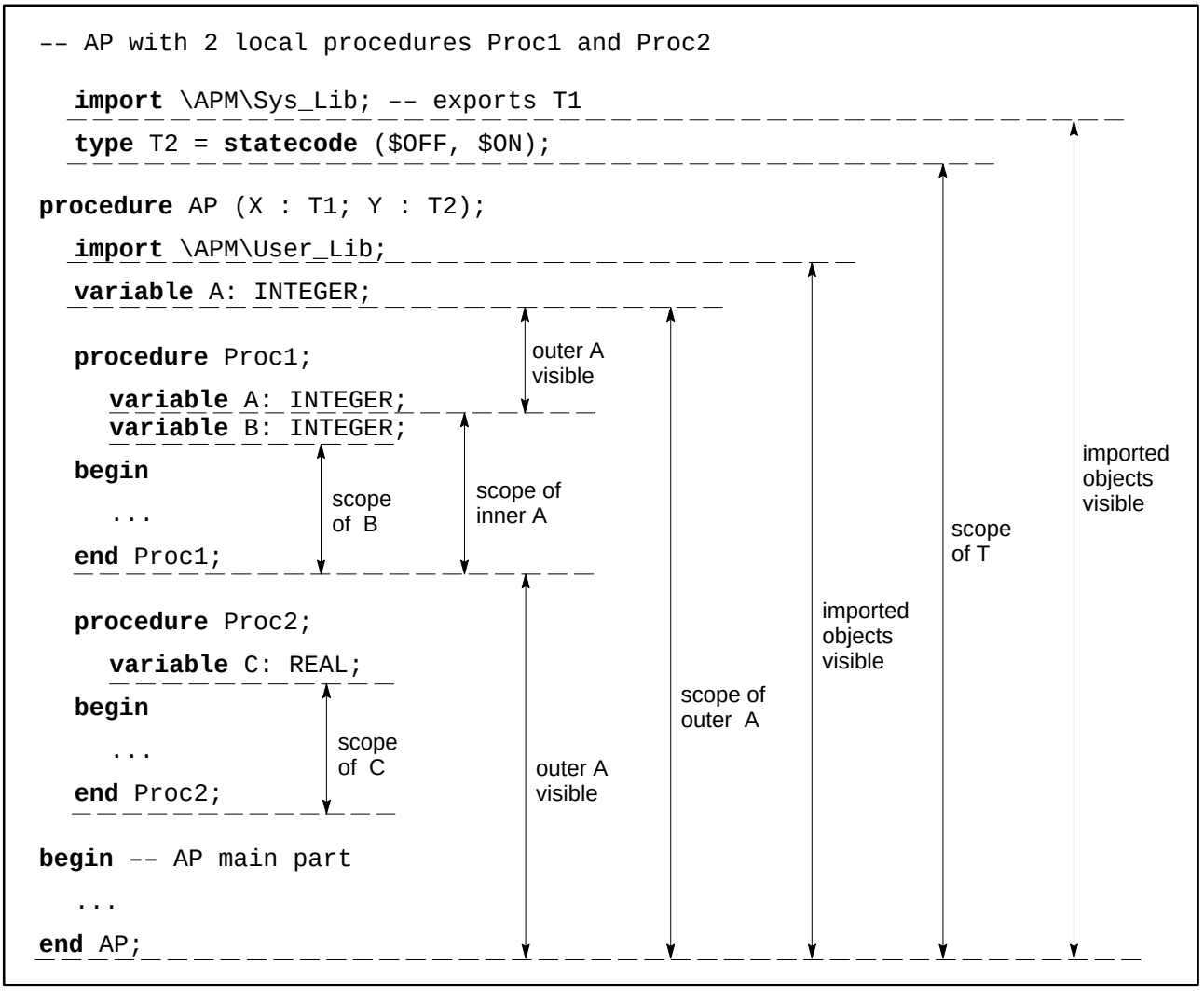

Figure 4. Scope and Visibility of Identifiers

4.3.3 Database Scope and Aliases (”nicknames”)

The database represents a name scope, too. It may contain certain identifiers, e. g. predefined aliases (”nicknames”) for database items. These identifiers can be used like any other identifiers and, like these, they can be hidden by identifiers from other scopes. The database scope is located “behind” all other scopes, i. e. its identifiers will be hidden by equal identifiers declared in any other scope.

4.3.4 Qualified Identifiers

A *qualified identifier* is an identifier prefixed with a pathname or alias designating the library or other importable module in which the qualified identifier is defined. The prefix and the identifier are separated by a period (dot). The qualified form can be used to access imported objects. It is the only way to access imported objects whose identifier is hidden.

Predefined identifiers can be qualified by preceding them with just a single dot, e. g. `.INTEGER`., `.MIN`, `.LENGTH`. This may be used to access a predefined identifier, if it is hidden by a local declaration of the same identifier.

The qualified form can also be used to access identifiers predeclared in the database. The pathname denoting the database scope is the root pathname, a single backslash (\).

Formal syntax

Qualified_Identifier	=	[[Name] "."] Identifier
Simple_Name	=	Identifier { Path_Identifier } | Pathname
Name	=	Simple_Name { "." Identifier }
Pathname	=	*see 4.3.6*
Identifier	=	*see 4.1.2.2*

Examples

```
GROUND_LIBRARY.Issue                    -- a procedure declared in a library
\APM\EGSE\USER_LIBS\MATH_LIB.Matrix     -- a type declared in a library
\.Ground_Library                        -- an alias predefined in the database
```

4.3.5 Lifetime of Objects

Apart from the static scope hierarchy described above, objects in UCL have a dynamic behaviour. An object is created when the block whose scope it is declared in is activated, it is deleted when the block terminates:

- Objects declared in an AP are created when the AP is started, and deleted when the AP terminates.
- Objects declared in a subprogram (procedure or function) are created when the subprogram is called and deleted when the subprogram ends. So in each call of the subprogram its locally declared identifiers will denote different objects. When a subprogram is called recursively, each incarnation of the subprogram will have its own set of local objects. These sets of objects form a stack, according to the subprogram call hierarchy. The identifiers denote the objects in the uppermost incarnation of the subprogram, i.e. the currently active incarnation.
- All objects declared in the specification or body of any modules linked to an AP by direct or indirect import, are created when the AP is started and deleted when it terminates.
- The predefined objects are created when the AP is started and deleted when it terminates.
- A **`for`** loop variable is created when the loop is entered and deleted when the loop ends.

4.3.6 MDB Objects, Pathnames

Mission Database (MDB) *objects* are, by default, visible to the UCL program, i.e. they need not be explicitly declared. Their definitions are ”inherited” from the MDB.

An MDB object (or MDB item) is identified by a *pathname* reflecting its respective position in the hierarchical name tree. Some MDB items may have *subitems*. These are denoted by a *subitem pathname*, i.e. an identifier prefixed with the pathname of the MDB item, separated with a dot. Note that the subitem identifier may be identical to a reserved word.

Formal syntax

Pathname	=	”\” | ”\\” | Path_Identifier { Path_Identifier }
Subitem_Pathname	=	Path_Identifier { Path_Identifier } ”.” Identifier
Identifier	=	*see 4.1.2.2*
Path_Identifier	=	*see 4.1.2.3*

Examples

`\APM\PAYLOAD\EQUIPMENT_UNIT_A`	a pathname
`\APM\PAYLOAD\EQUIPMENT_UNIT_A.INPUT_1`	a subitem pathname

MDB objects may have parameters, e.g. APs, messages, stimuli. It depends on the program context whether or not an actual parameter list must be supplied with the pathname of such items, see 4.8.3 for details. The parameter list is then given together with the pathname, e.g.:

```
\EGSE\TES01\AP_1 (3.14)
```

MDB objects fall into several classes or types. The *item type* of an MDB object (not to be confused with the corresponding UCL type) defines the object’s characteristics or properties and, by implication, the semantic rules governing its usage.

In the UCL environment, several kinds of operations or user interactions may be performed on MDB objects. In particular, all monitoring and control operations (measurement acquisition, stimuli commands, etc.) are performed via dedicated system library procedures/functions. Also, other kinds of interactions are available via a system library, e.g. a procedure that causes a specific MDB object of type ”automated procedure” to be executed.

A pathname designator tells the UCL compiler to retrieve an object’s definition (classification, type, etc.) from the Mission Database to check its semantics. Depending on the context, a pathname designator represents either a value or a reference to an MDB item.

In assignments, expressions, conditions, or when used as parameters of a type other than **`pathname`**, pathname designators refer to objects whose runtime values may change independently of the executing UCL program, e.g. value of sensor data periodically updated by data acquisition software.

In all other cases, they are references to specific object definitions in the MDB. A pathname designator by itself does not refer to any particular occurrence or version of the object. Such information must be provided to the compiler by the user.

Notes:

- The definition of ”analog measurement” objects (in the MDB) includes the physical quantity (voltage, temperature, etc.) and the respective engineering units associated with the particular measurement. Similarly ’discrete measurement’ objects are associated with statecodes (names identifying the states the object may be in, such as: `$ON/$OFF`, `$OPEN/$CLOSED`, `$LOW/$MEDIUM/$HIGH`).

- The runtime values of ”analog measurement” objects are expressed in engineering values, i.e. the acquired raw values are converted to the predefined engineering units using the appropriate calibration curves. For discrete objects the runtime values correspond to the predefined statecodes.
- The runtime values of MDB items have a *software type* according to the UCL type system. (See Type Correspondence Table between MDB and UCL objects in Appendix F).

MDB items are further classified by an *access class*. Some of these access classes define the allowed usage of an item in UCL and HLCL, these are marked with UCL and/or HLCL superscripts:

READ UCL, HLCL

indicates that the item has a runtime value that is read–only, i.e. it may be read but cannot be altered by a UCL program. The corresponding pathnames may be used in an expression (may appear on the right–hand side in an assignment statement) or as an **`in`** parameter. In expression evaluation, the current (i.e. most recent) runtime value of the MDB item is used.

READ/WRITE UCL, HLCL

indicates that the item has a runtime value that may be read and altered by a UCL program. Items of this class may be regarded as global software variables. The corresponding pathnames may be used in expressions, assignments (on either side of the assignment symbol) or in parameter lists (as **`in`**, **`in out`**, or **`out`** parameters).

IMPORT UCL, HLCL

applies to UCL libraries that may be imported by other compilation units.

EXECUTE HLCL

applies to MDB items of type ”automated procedure” which may be invoked by other APs.

PATH SELECT HLCL

applies to parent (virtual) MDB items, i.e. those at non–terminal nodes of the MDB name tree.

NODE SELECT HLCL

applies to MDB items of type ”network node”. The corresponding pathnames designate a specific computer in the network.

SEND

applies to items that represent commands (stimuli, telecommands) that may be sent to specific target systems.

none

applies to all other MDB items.

Note: A compilation unit may reference itself, directly or indirectly, in non–parameterized form. Parameterized references require the referenced unit to be compiled and up to date, this prohibits self–references and cyclic references.

Examples:

```
\APM\DMS\ITEM_A := 3.5;
```

set current runtime value of the global software variable named `\APM\DMS\ITEM_A` to `3.5`. (valid only if the MDB item’s access class is READ/WRITE and its software type is REAL)

`if \APM\XYZ\SWITCH = $OPEN ...;`	check whether current runtime value of MDB item named `\APM\XYZ\SWITCH` is `OPEN` (valid only if the MDB item's access class is READ or READ/WRITE and its software type is **`statecode`**)
`x := \APM\DMS\ITEM_X;`	assign current runtime value of MDB item `\APM\DMS\ITEM_X` to the local variable `X` (valid only if the MDB object's access class is READ or READ/WRITE and its software type is compatible with the type of `X`)
`Issue (\APM\xyz\POWER_ON);`	pass an MDB item named `\APM\xyz\POWER_ON` as a parameter to a system library procedure. Here a reference to the item is passed, not its runtime value. The procedure may retrieve its definition from the MDB, including stimulus characteristics, target equipment, authorization, etc.

4.3.7 Node Names

UCL is designed for use in a networked environment consisting of several *network nodes* (processors). These nodes are referenced by *node names* which correspond to their MDB pathnames. Node names and pathnames are thus syntactically equivalent.

4.4 Annotations

Declared items can have a so–called *annotation* attached. An annotation is a descriptive text that is kept together with the item and can be displayed in an HLCL command window as part of the help output for the item. Annotations will typically be used in library specifications (4.16.2), but can be used in any compilation unit for any type of declaration.

The form of an annotation is very similar to a list of comments, the only syntactic difference is that annotation lines start with '`<–`', while comment lines start with '`––`', both extend to the end of the line. An annotations given directly after a declaration is attached to the item defined by that declaration. For subprogram declarations an annotation is given after the subprogram header (4.5). For AP and library declarations (4.16) it is given after the AP or library header, respectively.

Annotations may be applied to all declared items, including enumeration literals (4.8.1.11), fields of a record (4.8.2.2) and formal parameters (4.6).

Formal Syntax

Annotation	=	Annotation_Line { Annotation_Line }
Annotation_Line	=	"<–" { Printable } *eol*
Printable	=	*any of the printable Characters in the underlying character set*
eol	=	*end of line*

Examples

```
unit [cwt] = [112 lb];  <- hundredweight

variable Default_Width : Integer := 10;
  <- Default output width for numeric values
  <- Changing this value will globally change the output format

type States = (Enabled,   <- Device switched on
               Disabled,  <- Device switched off
               Locked     <- Access to device blocked
              )
  <- Possible states of a device

type Device = record
                Name    : string (20);       <- Device name
                Address : Unsigned_Integer;  <- Device address
                State   : States;            <- Current state of device
              end record;
  <- Description of a device


procedure Enqueue (in out Container : Queue;   <- queue container
                   in     Element   : Object   <- element to be enqueued
                  );

  <- Append an element to the end of the queue.
  <- If queue is full, global Status is set to Failed.

begin
   ...
end Enqueue;
```

4.7 Constant Declarations

A constant declaration associates an identifier (a name) with a value. The associated value must be a constant or a constant expression. The latter is an expression consisting of literal constants and already defined constant identifiers; it can thus be computed at compile time.

A constant declaration is introduced with the reserved word **`constant`**, followed by the name of the constant. A colon (`":"`) associates the constant with a type, the "becomes" symbol (`:=`) associates it with a value.

For string constants, the type may be given as just **`string`**, the maximum length need not be specified, it is then assumed from the assigned string value.

Formal Syntax

Constant_Declaration	=	"constant" Identifier ":" Constant_Type ":=" Constant_Expression ";"
Constant_Type	=	Qualified_Identifier [Unit] | "string" ["of" Identifier] | "statecode" | "pathname" ["." "*"]
Qualified_Identifier	=	*see 4.1.2.2*
Constant_Expression	=	Expression
Expression	=	*see 4.12*
Unit	=	*see 4.11 and 4.1.2.10*

Examples

```
constant Two          : REAL           := 2.0;
constant Lower_Limit  : REAL           := 25.36;
constant Upper_Limit  : REAL           := Two * Lower_Limit;
constant Warning      : string         := "Value exceeds upper limit";
constant Code         : string of Byte := #"FF 0102_AE_F1 0103_E0_00";
constant On           : statecode      := $ON;
constant Max_Voltage  : Voltage        := 100.0 [V]; -- see unitized values
constant High_Value   : INTEGER        := MAX(INTEGER);
constant Set_Value    : BITSET         := {0, 3, 4};
```

4.8 Type Declarations

Every UCL object has a *type* defining the set of valid values that it may take, and the set of operations that may be performed on it. UCL types fall into the following broad categories:

- *Elementary types* comprise all *predefined* types (`INTEGER`,`REAL`, etc.), as well as the *enumeration types* (section 4.8.1.11) and *subrange* types (section 4.8.1.12) which enable users to create new data types. Such user–defined types make code more readable, and let the user exclude illegal values from storage in variables.
- *Structured types* allow the definition of data structures (arrays, records, sets, strings).
- *Unitized types* are for dimensioned quantities, i.e. associated with a unit of measure.
- *Pathname types* and *subitem pathname types* represent references to MDB items or their subitems, resp.
- *Inherited types* are types inherited from the software types of database items.

A type declaration is introduced with the reserved word **`type`**, followed by the type identifier. The ”=” symbol associates the identfier with the type definition.

Note that a type declaration need not introduce a new type, it may define a synonym to an existing type.

Formal Syntax

Type_Declaration	=	”type” Identifier ”=” Type [Unit] ”;”
Type	=	Simple_Type | String_Type | Statecode_Type | Pathname_Type | Array_Type | Set_Type | Record_Type | Pathname_Type | Subitem_Pathname_Type | Inherited_Type
Simple_Type	=	Qualified_Identifier | Enumeration_Type | Subrange_Type
Qualified_Identifier	=	*see 4.3.4*
Enumeration_Type	=	*see 4.8.1.11*
Subrange_Type	=	*see 4.8.1.12*
String_Type	=	*see 4.8.2.5*
Statecode_Type	=	*see 4.8.1.8*
Pathname_Type	=	*see 4.8.3*
Subitem_Pathname_Type	=	*see 4.8.4*
Array_Type	=	*see 4.8.2.1*
Set_Type	=	*see 4.8.2.3*
Record_Type	=	*see 4.8.2.2*
Inherited_Type	=	*see 4.8.5*
Unit	=	*see 4.11*

4.8.1 Elementary Types

4.8.1.1 Type `INTEGER`

Values of type `INTEGER` are whole numbers which may be positive or negative. Its range is given by the predefined constants: `MIN(INTEGER)` and `MAX(INTEGER)` corresponding to -2^{31} and $2^{31}-1$, respectively. Integer literals are defined in 4.1.2.4.1.

The operators `+`, `–`, `*`, `/` , `**` and `%` may be applied to values of this type and represent addition, subtraction, multiplication, division, exponentiation, and modulus, respectively. These operators are *infix* operators, which means that they are written between their operands. The `+` and `–` operators may also be used as *unary* operators to denote the sign of a value.

The relational operators (`<`, `<=`, `=` , `<>` , `>` , `>=`) are also applicable to this type and yield `BOOLEAN` values.

The predefined procedures/functions `MIN`, `MAX`, `ABS`, `ODD`, `INC` and `DEC` may be used for `INTEGER`, see 4.15.

The operator `/` denotes integer division (which always yields an integer result, possibly leaving a remainder). No rounding is performed; e.g. `9/4` = 2, `9/5` = 1. The result of an integer division is zero whenever the divisor is greater than the dividend.

Representation in memory

An Integer value is internally represented as one 32–bit word in 2's complement representation.

4.8.1.2 Type `UNSIGNED_INTEGER`

The type `UNSIGNED_INTEGER` comprises non–negative whole numbers in the range 0 to $2^{32}-1$. These bounds correspond to the predefined constants `MIN(UNSIGNED_INTEGER)` and `MAX(UNSIGNED_INTEGER)`, respectively. The compiler issues an error message if this range constraint is violated.

All operators defined for type `INTEGER` also apply to `UNSIGNED_INTEGER`. The arithmetic operators have cyclic semantics, i. e. the values do not overflow or underflow, but wrap around within the above mentioned range. It is, however, not possible to assign values outside the range.

Values of type `UNSIGNED_INTEGER` and `INTEGER` are mutually compatible. Thus, also mixed expressions, consisting of both `INTEGER` and `UNSIGNED_INTEGER` values, are allowed.

Representation in memory

An unsigned integer value is internally represented as one 32–bit word.

4.8.1.3 Type REAL

Values of the type REAL represent approximations of the mathematical real numbers. Their range is given by the predefined constants MIN(REAL) and MAX(REAL), corresponding approximately to the range: –3.4028 E+38 to 3.4028 E+38. Real values may be written either in decimal notation (with integer part and fractional part separated by a decimal point) or in the scientific (or exponential) notation, as defined in 4.1.2.4.3.

The operators that may be used with REAL operands are +, –, *, / and **, which represent addition, subtraction, multiplication, division and exponentiation, respectively. For the exponentiation operator ** the right operand must be of type INTEGER or UNSIGNED_INTEGER. The relational operators (<, <=, = , <> , > , >=) are also applicable to this type; they yield BOOLEAN values.

The predefined procedures/functions MIN, MAX, and ABS may be used for REAL, see 4.15.

Representation in memory

A REAL value is internally represented using one 32–bit word in IEEE standard single floating–point format.

4.8.1.4 Type LONG_REAL

Values of the type LONG_REAL represent approximations of the mathematical real numbers, like the REAL type, but with a better precision and a larger exponent range. Their range is given by the predefined constants MIN(LONG_REAL) and MAX(LONG_REAL) corresponding approximately to the range –1.7969E+308 to 1.7969E+308.

The syntax of LONG_REAL literals is the same as for REAL values, and the same operators may be applied.

Note that values of type REAL are compatible with those of type LONG_REAL and vice versa. Thus mixed expressions, consisting of both REAL and LONG_REAL values, are allowed.

Representation in memory

A LONG_REAL value is internally represented as two 32–bit words in IEEE standard double floating–point format.

4.8.1.5 Type BOOLEAN

The standard type BOOLEAN is an enumeration type defined as follows:

```
type BOOLEAN = (FALSE, TRUE);
```

All properties and operations defined for enumeration types apply to BOOLEAN as well. For a definition of enumeration types see 4.8.1.11. In addition, there are three special BOOLEAN operators: &, |, ~, corresponding to ”and”, ”or” and ”not”, respectively.

Representation in memory

A BOOLEAN value is internally represented like enumeration types (FALSE = 0, TRUE = 1).

4.8.1.6 Type CHARACTER

The value of a variable of type CHARACTER is an 8–bit *character* whose literals are defined by the ASCII character set. A character literal is written between single quotes (or apostrophes), see 4.1.2.6.

Values of type CHARACTER are ordered according to its 8–bit code, whose 7–bit first half is the ASCII character set (see Appendix E). The relational operators may therefore be applied. CHARACTER values may also be used in string operations, see 4.8.2.5. Type conversions to and from INTEGER can be used to convert between a character value and its ordinal number (its ASCII code), see 4.12.4. The predefined procedures/functions MIN and MAX may be used for CHARACTER, see 4.15.

Representation in memory

A CHARACTER value is internally represented as an unsigned integer in the range 0 .. 255 contained in one 32–bit word, where the subrange 0 .. 127 is the ASCII character set. Within strings, 4 characters are packed in one 32–bit word., see 4.8.2.5.

4.8.1.7 Low Level Types BYTE, WORD, LONG_WORD

The low level types BYTE, WORD and LONG_WORD represent individually accessible untyped storage units of 8, 32 or 64 bits, respectively. No operation is defined on these types, their values can only be assigned and passed as parameters. If the target variable of an assignment or the formal parameter of a subprogram is of a low level type, the assigned value or passed actual parameter, respectively, may be of any scalar type, but the value must not occupy more than the corresponding number of bits. In particular:

BYTE

The type BYTE represents untyped 8–bit objects. BYTE variables and formal BYTE parameters may be assigned/passed values of any scalar type that fit into 8 bits (CHARACTER, BOOLEAN, INTEGER and small enumeration types with up to 256 values), as well as byte strings of length 1. For integer values a check is done that the value will fit in 8 bits (i. e. it is in the range 0 .. 255), for non–static values the check will be performed at runtime. Note, however, that byte values are held in 32–bit words in memory. Only in strings, 4 bytes are packed in a word, see 4.8.2.5.

WORD

The type WORD represents untyped 32–bit objects. WORD variables and formal WORD parameters may be assigned/passed values of any scalar type that are represented in 32–bit words (CHARACTER, BOOLEAN, INTEGER, REAL, BITSET, pathname types, enumeration types), as well as byte strings of length 4.

LONG_WORD

The type LONG_WORD represents untyped 64–bit objects. LONG_WORD variables and formal LONG_WORD parameters may be assigned/passed values of any scalar type that is represented in 64 bits (LONG_REAL, TIME, statecode types, subitem pathname types), as well as byte strings of length 8.

Representation in memory

BYTE and WORD values are represented as one 32–bit word. Within strings, four BYTE values are packed in one 32 bit word. Within strings, four BYTE values are packed in one 32 bit word, see 4.8.2.5.

LONG_WORD values are represented as two consecutive 32–bit words.

4.8.1.8 Statecode Types

The set of legal values for a statecode type comprises the literals indicating the predefined states of *discrete* items (e.g. on/off, open/closed, low/medium/high etc.). Statecodes are defined in the Mission Database as attributes of the respective discrete items. In UCL, statecode constants are denoted by identifiers prefixed with a dollar sign ($), such as `$ACTIVE, $OFF`., see 4.1.2.5.

The special literal $$ means *no statecode value*. It may be used e. g. as a default value for statecode parameters or as an initial or dummy value for statecode variables. $$ is compatible to all statecode types, regardless of constraints.

Statecode types represent *unordered* sets of values, i.e. there is no defined less/greater relationship between the statecode literals, and the relational operators cannot be used. The only allowed operation on statecode values is comparison on equality and inequality (=, <>).

Statecode types may be unconstrained or constrained.

- An *unconstrained statecode type* is denoted by the keyword **`statecode`**, it comprises all possible statecode literals.
- A *constrained statecode type* is restricted to a specific set of statecode literals. This is denoted by the keyword **`statecode`**, followed by the list of allowed literals in parentheses.

All declared statecode types are subtypes of the general predefined type **`statecode`**.

Formal Syntax

Statecode_Type	=	”statecode” [”(” Statecode_List ”)”]
Statecode_List	=	Statecode { ”,” Statecode }
Statecode	=	”$” Identifier
Identifier	=	(see 4.1.2.2)

Examples

```
type Code   = statecode;                -- unconstrained type
type Switch = statecode ($ON, $OFF);    -- constrained type

variable S1 : statecode;                -- unconstrained variable
variable S2 : Code;                     -- unconstrained variable
variable S3 : Switch;                   -- constrained

...

S1 := $HIGH;
S2 := $HIGH;
S3 := $HIGH;       -- not allowed, constraint violation
S2 := $ON;
S3 := $OFF;
```

Representation in memory

Internally, a statecode variable is represented by two 32–bit words containing the corresponding statecode literal as an ASCII character sequence (up to 8 characters) in all upper–case, left–justified and padded with blanks. The literal $$ is represented as 8 blanks.

4.8.1.9 Types TIME and DURATION

UCL distinguishes between absolute and relative time references. Two predefined data types are provided: TIME, whose values are absolute points in time, and DURATION, whose values are time distances.

TIME values range from 1901 to 2099, the lowest and highest allowed values can be obtained with the standard functions MIN(TIME) and MAX(TIME). The resolution of TIME values is 1 nanosecond.

The type DURATION comprises LONG_REAL values expressed in seconds, the lowest and highest allowed values can be obtained with the standard functions MIN(DURATION) and MAX(DURATION). The predefined type DURATION is declared as

```
type DURATION = LONG_REAL [s];
```

The following operations may be performed on TIME/DURATION objects:

```
time     + duration = time
duration + time     = time

time     – duration = time
time     – time     = duration

duration * real     = duration
duration / real     = duration

duration + duration = duration
duration – duration = duration
```

Furthermore, all comparison operations are provided for both types, e.g.:

```
time     < time     = boolean

duration < duration = boolean
```

Time values may be with or without a date. In the second case the date is ignored in any operations. If in an operation an operand is without a date, it is assumed to refer to the same date as the other operand, e.g.

```
01.09.1997 13:00 – 12:00 = 3600.0 [s]
12:00 – 13:00            = –3600.0 [s]
```

The special time literal ~:~ denotes *no time*. This may be used e.g. as a default value for TIME parameters or variables. It can only be compared on equality (=, <>) and must not be used in other operations.

Representation in memory

A TIME value is internally represented using two 32–bit words. They contain the following time components in packed format in this order: year – 1900 (8–bit integer), month (4–bit integer), day (5–bit integer), seconds since midnight (47–bit fixed point value with 17 bits before and 30 bits after the decimal point). Times without a date are represented with the year, month and day fields = 0. The constant ~:~ (no time) is represented with all bits in both words set to 1.

DURATION values are represented like LONG_REAL values (see 4.8.1.4).

4.8.1.10 Type COMPLETION_CODE

The predefined type COMPLETION_CODE is an enumeration type defined as

```
type COMPLETION_CODE = (SUCCESS, FAILURE)
```

All characteristics and operations defined for enumeration types (see 4.8.1.11) apply to COMPLETION_CODE as well.

Representation in memory

COMPLETION_CODE values are represented like other enumeration types, with SUCCESS = 0 and FAILURE = 1.

4.8.1.11 Enumeration Types

An enumeration consists of an ordered sequence of identifiers. These identifiers are constants defining the list of values that may be assumed by a variable of that type.

Enumerated types are ordered by the sequence of values in the enumeration (in the example below, Red is smaller or less than Green). In an enumeration with n elements, the first element has the ordinal number 0 and the last element the ordinal number n–1. All comparison operators can therefore be used on enumeration values, according to this ordering. Type conversions (see 4.12.4) can be used to convert between an enumeration value and its ordinal number.

The predefined procedures/functions MIN, MAX, INC and DEC may be used for enumeration types, see 4.15.

Note that BOOLEAN and COMPLETION_CODE are enumeration types. Whenever an enumeration type may be used, it includes these two types.

Formal Syntax

Enumeration = "(" Identifier_List ")"
Identifier_List = Identifier { "," Identifier }
Identifier = *see 4.1.2.2*

Examples

```
type Color = (White, Red, Green, Blue, Black);
type State = (Ready, Active, Suspended);
```

Representation in memory

Enumeration values are represented like INTEGER, the 32–bit word stores the ordinal value.

4.8.1.12 Subrange Types

A subrange type denotes a restricted range of consecutive values from some other type (called the *base type)*. The latter must be one of the following types: `INTEGER`, `UNSIGNED_INTEGER`, `REAL`, `LONG_REAL`, `CHARACTER` or enumeration type. They are not separate types, but subtypes of other types. All characteristics and operations defined for the base type apply to the subtype as well. However, a value to be assigned to a variable of a subrange type must lie within the specified interval, otherwise a runtime error occurs.

A subrange type is specified by its bounds, i.e. the lowest and highest allowable values. The lower bound must not be greater than the upper bound, further they must be both specified as constants or constant expressions. Both bounds must be compatible with the given base type. The predefined constant functions `MIN` and `MAX` may be used to obtain the lower and upper bound, resp.

It is possible to define a subrange type of another subrange type. Then both subrange types have the same base type, and the bounds of the new type must not lie outside the bounds of the old type.

A subrange type declaration is given by the name of the base type, followed by the range, enclosed in parentheses, as in `INTEGER (10 .. 100)`. The two dots are part of the syntax.

Formal Syntax

Subrange	=	Qualified_Identifier ”(” Constant_Expression ”..” Constant_Expression ”)”
Qualified_Identifier	=	*see 4.3.4*
Constant_Expression	=	*see 4.12*

Examples

```
type Index         = INTEGER (0 .. 25);
type Light_Colors  = Colors (Red .. Blue);
type Digit         = CHARACTER (’0’ .. ’9’);
```

Representation in memory

Subrange values are represented like values of their base type.

4.8.2 Structured Types

4.8.2.1 Array Types

An array is a structure consisting of a fixed number of components which are all of the same type. Each component can be denoted and directly accessed by the name of the array variable followed by the *array index* in parentheses, e.g. `A(i)` indicates the `i`–th element of the array `A`. For matrices, `A(i,j)` or `A(i)(j)` are both valid ways to access an element, where `i` is the row, and `j` the column of the matrix. The number of dimensions of an array is determined by the number of indices, it is not restricted by the language. A runtime error occurs if the array index is outside the defined ranges.

Each index belongs to a specific index type. This must be a discrete type (`INTEGER`, `UNSIGNED_INTEGER`, `CHARACTER`, enumeration type or constrained statecode type). If a constrained statecode type is used as the index type of an array, the order in which the array components are arranged is not defined, since statecode types are unordered. A program must not rely on a specific order.

An array type is specified in the form

```
array (index_range) of element_type
```

The index range may either be given as an explicit range in the form *`min .. max`* or as the name of a discrete type. In the latter case the index range comprises all values of the type:

```
(1)  array (min .. max) of element_type
(2)  array (type_name) of element_type
```

For statecode types, (2) is the only allowed form.

The type of an array element may itself be an array (this leads to multidimensional arrays). In that case, the following two notations are identical:

```
(1)  array (0 .. 10) of array (0 .. 20) of INTEGER
(2)  array (0 .. 10, 0 .. 20) of INTEGER
```

No operations are defined on array objects. They can only be assigned and passed as parameters to subprograms.

The predefined functions `LOW` and `HIGH` can be used to determine the lower or upper bound, respectively, of an array type or variable. For an array with more than one dimension, the bounds of the second, third, etc. dimension can be obtained by specifying the dimension as the second parameter of `LOW` and `HIGH`.

Formal Syntax

Array_Type	=	”array” ”(” Index_Range { ”,” Index_Range } ”)” ”of” Qualified_Identifier [Unit]
Index_Range	=	Constant_Expression ”..” Constant_Expression | Qualified_Identifier
Constant_Expression	=	*see 4.12*
Qualified_Identifier	=	*see 4.3.4*
Unit	=	*see 4.11 and 4.1.2.10*

Examples

```
type Row_Index   = INTEGER (0 .. 10);
type Col_Index   = INTEGER (0 .. 20);

type Vector      = array (1 .. 3) of REAL;
type Matrix      = array (Row_Index, Col_Index) of REAL;
```

```
type Color       = (White, Red, Green, Blue, Yellow, Cyan, Magenta, Black);
type RGB         = Color (Red .. Blue);

type Bool_Array  = array (Red .. Magenta) of BOOLEAN;
type Color_Table = array (Color, RGB) of REAL;

type Level       = statecode ($LOW, $MEDIUM, $HIGH);
type Level_Array = array (Level) of REAL;
```

Examples of LOW and HIGH

```
variable V : Vector;
variable M : Matrix;
variable C : Color_Table;

LOW (Vector)    = 1      HIGH (Vector)    = 3
LOW (V)         = 1      HIGH (V)         = 3

LOW (Matrix, 1) = 0      HIGH (Matrix, 1) = 10
LOW (M, 2)      = 0      HIGH (M, 2)      = 20

LOW (C, 1)      = White  HIGH (C, 1)      = Black
LOW (C, 2)      = Red    HIGH (C, 2)      = Blue
```

Aggregates

An array value may be written in form of an *aggregate* with the array elements enumerated in parentheses:

```
(value, value, ..., value)
```

Aggregates of multidimensional arrays are expressed in nested form. A matrix is written as a vector of line vectors:

```
((value, value, ..., value),     -- line 1
 (value, value, ..., value),     -- line 2
 ...
 (value, value, ..., value))     -- line n
```

A three–dimensional array as a vector of two–dimensional planes, which are written as vectors of line vectors. This concept is recursively applied to any dimensionality:

```
(((value, value, ..., value),    -- plane 1
  (value, value, ..., value),
  ...
  (value, value, ..., value)),

  ...

 ((value, value, ..., value),    -- plane n
  (value, value, ..., value),
  ...
  (value, value, ..., value)))
```

An empty aggregate (for an array with no element) is written as an empty pair of parentheses: ( ).

Aggregates are expressions, see 4.12.

Representation in memory

An array is represented as a sequence of its elements in their respective representation. Arrays with more than one dimension are stored ”column–wise”, i.e. such that for all index positions i and i+1 indices on position i+1 vary faster than on position i.

4.8.2.2 Record Types

Like an array, a record is a structured type. In a record, however, the components are not constrained to be of identical type. In the record declaration, each component (or field) must be specified by a name (field identifier) and a type. To reference a field, the name of the record is followed by a period (dot) and the respective field identifier, e.g. `BIRTHDAY.MONTH`.

Records may be nested. The scope of a field identifier is the innermost record in which it is defined. Hence, field identifiers (i.e. the names of record components) may be reused outside the scope of that record definition.

A record type may have several *variants*, based on the values of a specific field (the *tag field*). These variants may differ in the number and type of components and thus allow for alternate forms of the same record. They are enclosed by the keywords **`case`** and **`end case`**. Each keyword **`when`** introduces a variant that is to be part of the record for the list of tag field values given after the keyword.

Note that changing the value of the tag field, e.g. by an assignment, may select a different variant and thus change the structure of the record variable. Access to fields outside the currently selected variant, i.e. to undefined components, may have unpredictable effects. This is not checked and may be used for *low–level programming* purposes to circumvent UCL's type checking mechanisms, exploiting the fact that all variants occupy the same location in memory and thus "overlay" each other. This is not normal programming practice and should be avoided. If necessary for special and exceptional cases, great care must be taken. For an example see chapter Libraries (4.16.2), example 2.

No operations are defined on record objects. They can only be assigned and passed as parameters to subprograms.

Formal Syntax

Record_Type	=	"record" { Fields } "end" "record"
Fields	=	Identifier_List ":" Qualified_Identifier [Unit] | "case" Identifier ":" Qualified_Identifier Variant_Part "end" "case" ";"
Variant_Part	=	{ "when" Case_Label_List ":" { Fields } } ["else" { Fields }]
Case_Label_List	=	Case_Labels { "," Case_Labels }
Case_Labels	=	Constant_Expression [".." Constant_Expression]
Identifier_List	=	Identifier { "," Identifier }
Qualified_Identifier	=	*see 4.3.4*
Identifier	=	*see 4.1.2.2*
Constant_Expression	=	*see 4.12*
Unit	=	*see 4.11 and 4.1.2.10*

Examples

a) Simple record

```
record
   Day    : INTEGER;
   Month  : INTEGER;
   Year   : INTEGER;
end record;
```

b) Record with variant parts

```
type Measurement_Type =  (Analog, Discrete);
type Eng_Unit_Type    =  array (1..4) of CHARACTER;
...
type Measurement =  record
                      Channel_No : INTEGER;
                      case Measure: Measurement_Type
                        when Analog:
                          Eng_Value :  REAL;
                          Eng_Unit  :  Eng_Unit_Type;
                        when Discrete:
                          State     :  statecode;
                      end case;
                    end record;
```

Representation in memory

A record is represented as the sequence of its fields in their respective representation. All variants of the same variant part are mapped to the same location in memory. The size of a variant part is determined by its longest variant.

4.8.2.3 Set Types

A set is a collection of objects of the same type (the *base type*). The latter must be a discrete type with at most 2^{16} values (i.e. a set may contain as many elements as can be expressed using an unsigned 16–bit integer).

A set type declaration is denoted by the keywords **`set of`** followed by the appropriate type identifier.

A set constant is represented by a list of its members, enclosed in braces (curly brackets), and preceded by the respective type identifier (base type), e.g. `CharSet{'a', 'b', 'c'}`. If the type identifier is omitted, the predefined type `BITSET` is assumed, e.g. `{3,5,15}` is a literal of type `BITSET`.

If the members of a set are consecutive values, they may be expressed as a range: e.g. `{5,6,7,8,9}` may be written `{5..9}`.

A set may have no members at all, in which case it is called the *empty set* and is written `{}` .

The operations applicable to *set* variables are:

Operation	Operator
set union	`+`
set difference	`–`
set intersection	`*`
symmetric set difference	`/`
set inclusion	`<=, >=`
set comparison	`= , <>`
membership tests	**`in`**

The *union* of two sets is a set containing the members of both sets.

The *difference* of two sets is a set containing all the members of the first set that are not members of the second set.

The *intersection* of two sets is the set of objects that are members of both sets.

The *symmetric set difference* of two sets is a set containing those elements that are members of exactly one of the two sets (not of both).

The membership operator **`in`** is used to test for *set membership*. **`in`** is treated as a relational operator. The expression `I` **`in`** `S` is of type `BOOLEAN`, returning `TRUE` if `I` is a member of the set `S`.

The predefined procedures `INCL` and `EXCL` (see 4.15) may be used to include a member in, or exclude a member from, resp., a set variable.

Formal Syntax

Set_Type = ”set” ”of” Simple_Type.

Simple_Type = *see 4.8*

Examples

```
type Id            = INTEGER (0 .. 100);
type Id_Set        = set of Id;
type Traffic_Light = set of (Red, Green, Blue);
type Spectrum      = set of Color;
```

Set Constants

A set constant may be written by enumerating the set elements in braces, with the type name preceding the bracketed expression. Each element can be a constant expression:

```
type_name {value, value, ..., value}
```

Ranges of adjacent values may be written in the form `first .. last`, e. g.

```
Char_Set {'a' .. 'z', 'A' .. 'Z', '_'}
```

An empty set is written as an empty pair of braces:

```
Char_Set {}
```

Representation in memory

A set is represented by a number of bits packed in one or more 32–bit words, one bit for each possible member of the set. A member is present in the set, if its bit is set, it is absent otherwise. The first member is expressed by bit 0, the last member by bit n–1.

4.8.2.4 Type BITSET

The predefined type BITSET is defined as

```
type BITSET = set of INTEGER (0 .. 31)
```

It thus represents a set of 32 elements, corresponding to the width of a storage unit. All operations applicable to the set types equally apply to BITSET (see 4.8.2.3), but BITSET has additional special characteristics. It may be used to access a 32–bit storage word as a sequence of bits: The i–th bit of a bitset B, for example, is 1 if i is a member of B, 0 otherwise. Values of type INTEGER and UNSIGNED_INTEGER may be converted to BITSET and vice versa (see 4.12.4).

Examples

```
variable Flags: BITSET;
variable I    : INTEGER;

...
Flags  := {3,5,12};
I      := INTEGER (Flags);
```

Bitset Constants

Bitset constants are written in braces, like set literals (see 4.8.2.3), but the preceding type name may be omitted, e. g.

```
{1, 3, 5, 7, 9, 20 .. 31}
{0 .. 31}
```

An empty bitset is written as an empty pair of braces:

```
{}
```

Representation in memory

A bitset is represented like a set (see 4.8.2.3) in one 32–bit word.

4.8.2.5 String Types

String types are used to store character or byte sequences of variable lengths. A string type is declared with the keyword **`string`**, followed by the maximum length of the string in parentheses and, optionally, an indication of the component type (`CHARACTER` or `BYTE`). If the component type indication is omitted, `CHARACTER` is assumed, so

```
string (80)
string (80) of CHARACTER
```

have identical meaning and denote character strings, whereas

```
string (80) of BYTE
```

denotes a byte string.

The actual length of a variable of a string type may vary between 0 and the declared maximum length. The actual length may be obtained with the predefined function `LENGTH`. In a string, each 32 bit word stores four consecutive characters or bytes.

One may access individual string elements like in array indexing: if `S` is a string variable, then `S(1)` denotes the first character or byte in the string. The first element has index 1 etc.

Strings of different lengths may be assigned and compared. If a string with an actual length longer than the maximum length of the target string is to be assigned, a run–time error occurs (i.e. there is no implicit truncation). A run–time error also occurs if one tries to assign or retrieve an element outside the actual length of the string.

Character string values are written as string literals of the form `"..."` and may contain any printable ASCII characters. Byte string literals have a # prefix: `#"..."` and may contain only an even number of hexadecimal digits grouped with underscore and blank characters, see 4.1.2.7. An empty string is written as `""` or `#""`, respectively.

Supported operations on string variables are comparison and the concatenation of two strings with the "+" operator. The comparison operators <, <=, >, >= compare strings in lexicographic order, based on the ASCII values of the characters or on the unsigned numerical value of the bytes, respectively, e. g.

```
"aux" > "attention"      (character strings)
#"FF" > #"AA0012FF"      (byte strings)
```

A character or byte is treated as a string of length 1, if it is concatenated with a string or another character or byte, assigned to a string variable or constant, or passed as a string parameter, e.g.

```
"car" + 's' = "cars"
'a' + 'b'   = "ab"

#"AA_BB" + BYTE(255) = #"AA_BB_FF"
BYTE(1) + BYTE(2)    = #"01 02"
```

The predefined functions `LOW` and `HIGH` can be used to determine the lower or upper bound, respectively, of a string type or variable. `LOW` will always return 1, `HIGH` returns the maximum length of the string type or variable.

Substrings

In expressions, a substring (*slice*) may be selected from a string by giving the substring index range in parentheses. The lower and upper bounds may be outside the actual index range of the string.The result is the part of the string that lies within the given bounds:

```
S(5 .. 10)      selects range 5 .. 10
S(-5 .. 3)      selects range 1 .. 3
S(1 .. 0)       selects an empty substring (lower > upper)
```

Formal syntax

String_Type = ”string” ”(” Constant_Expression ”)” [“of” Identifier]

Constant_Expression = *see 4.12*

Identifier = *see 4.1.2.2*

Examples

```
variable S   : string (255);          -- String with max. 255 characters
variable S80 : string (80);           -- String with max.  80 characters

variable B   : string (255) of BYTE;  -- String with max. 255 bytes
variable B80 : string (80) of BYTE;   -- String with max.  80 bytes

...
S   := ”Hello world”;
S80 := S;                         -- implies length check
S   := S80;

B   := #””;
B80 := B;                         -- implies length check
B   := B80;

...
S(1) := ’X’;                      -- replace first character in S
S80  := ”Hello” + ’ ’ + S;        -- string concatenation

B(1) := 16#FF#;                   -- replace first byte in B
B    := B + B80 + BYTE(0);        -- string concatenation
```

Examples of substrings

```
S80  := S(11 .. 20);              -- selects a character substring of length 10
B80  := B(11 .. 20);              -- selects a byte substring of length 10
```

Examples of LOW and HIGH

```
type String_80 = string (80);

variable S_80 : String_80;
variable S    : string (HIGH (String_80) + 20);

LOW (String_80) = 1   HIGH (String_80) = 80
LOW (S_80)      = 1   HIGH (S_80)      = 80
LOW (S)         = 1   HIGH (S)         = 100
```

Representation in memory

A string is represented as one 32–bit word holding the actual length of the string as an unsigned integer, followed by zero or more words holding a byte array with 4 bytes (characters) each, packed in one word. The number of bytes is the maximum length of the string, filled up to a multiple of 4.

4.8.3 Pathname Types

The values comprised by pathname types are references to database objects themselves, not to their runtime value. Pathnames are regarded as literals that denote the values of these types. The special literal \\ means *no pathname*. It may be used e.g. as a default value for pathname parameters or as an initial or dummy value for pathname variables. \\ is compatible to all pathname types, regardless of constraints or parameterisation (see below). Pathname values can be assigned and compared on equality (=, <>).

Pathname types can be unconstrained or constrained:

- An *unconstrained pathname type* is denoted by the keyword **`pathname`**, it comprises all valid pathnames.
- A *constrained pathname type* is restricted to a specific set of MDB item types. This is denoted by the keyword **`pathname`**, followed by the list of allowed item types in parentheses. The item types are represented by a special set of identifiers defined in the database: the item type names. These identifiers may appear only in this specific context, they do not interfere with normal UCL identifiers. The compiler will check that pathnames assigned to a variable, associated with a constant or passed as a parameter are of one of the item types allowed for the target object.

All declared pathname types are subtypes of the general predefined type **`pathname`**.

Formal Syntax

Pathname_Type = ”pathname” [”(” Identifier_List ”)”]
Identifier_List = Identifier { ”,” Identifier }
Identifier = *see 4.1.2.2*

Examples

```
procedure P (X : pathname); begin ... end P;                    -- unconstrained
type Monitor_Item  = pathname  (EGSE_INTEGER_MEASUREMENT,       -- constrained
                                EGSE_FLOAT_MEASUREMENT,
                                EGSE_INTEGER_SW_VARIABLE,
                                EGSE_FLOAT_SW_VARIABLE);
type AP_Item       = pathname  (UCL_AUTOMATED_PROCEDURE);       -- constrained
```

If a formal parameter is of a (possibly constrained) pathname type and followed by the syntax **`“()”`**, the actual parameter must be followed by its own actual parameter list (if one is declared within the database). Such a formal parameter is then known to be of a *parameterized pathname* type. For such a parameter, only pathnames of items that may have parameters can be passed. The property whether or not an item can have parameters is an attribute stored together with the item in the database. Parameters of a parameterized pathname type can only be of mode **`in`** (see 4.14.1). Whithin the subprogram they may be accessed as parameterized items (passed to other subprograms) or non–parameterized items, e.g assigned to a variable or passed to a subprogram that requires a non–parameterized item as parameter.

Examples

```
procedure Acquire (Item : Measurement); ...  -- non-parameterized item
procedure Execute (AP : AP_Item()); ...      -- parameterized item
```

Note that all checks concerning constraints and parameterisation of pathname types are performed at compilation time, not at runtime. Whenever a pathname value is assigned to a variable or passed as a parameter, it is checked that the type of the source value has at least the constraints of the target object.

Example

```
type Measurement = pathname (EGSE_INTEGER_MEASUREMENT,
                             EGSE_FLOAT_MEASUREMENT);

procedure P (X : pathname; Y : Monitor_Item; Z : AP_Item());

  variable P : pathname;
  variable M : Measurement;
  variable A : AP_Item;

begin

  P := X;       -- OK. P may be assigned any type of pathnames ...
  P := Y;
  P := Z;       -- ... even parameterized pathnames

  M := Y;       -- Error: M is more constrained than Y
  M := X;       -- Error: X is not constrained at all, but M is

  A := Z;       -- OK. Non-parameterized access to parameterized item

  Execute (Z);  -- OK. Z is parameterized and thus accepted by Execute
  Execute (A);  -- Error: A is not parameterized, even if Z was assigned.

end P;
```

Representation in memory

Non–parameterized pathname values are represented as one 32–bit word containing the *short identifier* (SID) of the item as an unsigned integer.

Parameterized pathname values can only occur as parameters to a subprogram. When the subprogram is called, a *parameter block* is constructed in memory that describes the item together with its actual parameter list, and the address of the parameter block is passed to the subprogram. The parameter block conforms to the *internal parameter encoding scheme* as described in reference document 2.2.2.

4.8.4 Subitem Pathname Types

The values comprised by subitem pathname types are references to subitems of database objects themselves, not to their value. Subitem pathnames are regarded as literals that denote the values of these types. Subitem pathname values can be assigned and compared on equality (=, <>).

Subitem pathname types can be unconstrained or constrained:

- An *unconstrained subitem pathname type* is denoted by the pattern **`pathname.*`**, it comprises all valid subitem pathnames.
- A *constrained subitem pathname type* is restricted to a specific set of MDB item types. This is denoted by the pattern **`pathname.*`**, followed by the list of allowed subitem types in parentheses. The subitem types are represented by a special set of identifiers defined in the database: the subitem type names. These identifiers may appear only in this specific context, they do not interfer with normal UCL identifiers. The compiler will check that subitem pathnames assigned to a variable, associated with a constant or passed as a parameter are of one of the subitem types allowed for the target object.

All declared subitem pathname types are subtypes of the general predefined type **`pathname.*`**.

Formal Syntax

Pathname_Type = ”pathname” ”.” ”*” [”(” Identifier_List ”)”]
Identifier_List = Identifier { ”,” Identifier }
Identifier = *see 4.1.2.2*

Examples

```
type Subitem  = pathname.*;                    -- unconstrained
type Input    = pathname.* (FB_IO_INPUT);      -- constrained to FB_IO_INPUT
procedure P (X : pathname.*); begin ... end P; -- unconstrained parameter
procedure Q (X : Input); begin ... end Q;      -- constrained parameter
```

Note that all checks concerning constraints of subitem pathname types are performed at compilation time, not at runtime. Whenever a subitem pathname value is assigned to a variable or passed as a parameter, it is checked that the type of the source value has at least the constraints of the target object.

Example

```
procedure P (X : pathname.*; Y : Input);
  variable P : pathname.*;
  variable I : Input;
begin
  P := X;       -- OK. P may be assigned any type of subitem pathnames ...
  P := Y;       -- ...
  I := X;       -- Error: X is not constrained at all, but I is
  I := Y;       -- OK. Same constraints on both sides
  Q (X);        -- Error: Formal parameter is constrained, but X is not
  Q (Y);        -- OK. Formal and actual parameter have same constraints
end P;
```

Representation in memory

Subitem pathname values are represented as two 32–bit words: the first contains the short identifier (SID) of the database item, the second the subitem identifier, both as unsigned integers.

4.8.5 Inherited Types

An *inherited type* inherits the *software type* (= UCL type) of a database item, defined together with the item in the database. Note that only items with access class READ and READ/WRITE have a software type, see 4.3.6 for a description. An object (variable, constant, parameter) declared with an inherited type thus has the same type properties as the corresponding database item, including any constraints and engineering units.

Formal Syntax

Inherited_Type = ”type” ”of” Name (Name is a, possibly aliased, pathname)

Name = *see 4.3.4*

Example

```
type Voltage = type of \APM\FLTSYS\MEAS\Volt1;
variable V : Voltage;  -- V has the same type as \APM\FLTSYS\MEAS\Volt1
...
V := X;  -- implies a check against the constraints of \APM\FLTSYS\MEAS\Volt1
```

Representation in memory

Inherited types are not separate types. They always correspond to one of the above described types and are represented like these.

4.8.6 Compatibility of Types

4.8.6.1 General Rules

Whenever a value is assigned to a variable, passed as a parameter to a subprogram or used as an operand in an operation, it is required that it be *compatible* to the target object or to other operands in the operation. Usually, two objects are compatible if they are of the same type, they are incompatible if they are of different types.

Two types declared in different type declarations are considered different and incompatible types, even if the declarations are identical, except in the following cases:

- The type is declared to be identical to another type. In this case both type identifiers denote the same type.
- The type is declared to constrain some other type. In this case the type is a subtype of the other type. All subtypes of the same base type are compatible to the base type and to each other.

For some closely related types, a value of one type is implicitly converted to the other type, if needed:

- `INTEGER` and `UNSIGNED_INTEGER` types are always compatible.
- `REAL` and `LONG_REAL` types are always compatible.
- `CHARACTER` values are compatible to string values in certain contexts (see 4.8.2.5).

Values of certain types may be explicitly converted to certain other types, without violating the strict UCL typing rules (see 4.12.4).

For formal array and string parameters of subprograms there are ”open” forms that allow for less restrictive compatibility rules (see 4.13.2 and 4.12.3).

For unitized types, in addition to type compatibility, compatibility and commensurability of the involved measurement units must be observed (see 4.11).

Examples

```
type Arr_1  = array (1 .. 10) of REAL;     -- different types,
type Arr_2  = array (1 .. 10) of REAL;     -- although textually identical

type Number = INTEGER;                     -- identical types
type Arr_3  = Arr_1;                       -- ...

type Short  = INTEGER (0 .. 2**16 - 1);    -- subtypes ...
type Switch = statecode ($OFF, $ON);       -- ...
type Node   = pathname (EGSE_NODE);        -- ...
type Input  = pathname.* (FB_IO_INPUT);    -- ...

variable A1 : Arr_1;                       -- A1 and A2 are incompatible
variable A2 : Arr_2;

variable A3 : Arr_1;                       -- A3 and A4 are compatible
variable A4 : Arr_3;
```

Expressions are always evaluated in terms of the base type of the operands (see 4.12), constraints are therefore not relevant. But when a value is assigned to a variable, or passed as a parameter to a subprogram, it is checked that the value does not violate the constraints of the target variable (see 4.13.1) or formal parameter (see 4.13.2 and 4.12.3).

The check whether a value violates a constraint is normally performed at runtime. If the value is a constant, und hence known at compile time, the check will be done by the compiler. Pathname and

subitem pathname constraints are always checked at compile time (see 4.8.3 and 4.8.4), in order to avoid database queries at runtime which would otherwise be necessary to complete the check.

4.8.6.2 Structural Compatibility

When accessing structured runtime values of database items, the strict compatibility rules described above cannot be applied: e.g. a local array value would always be incompatible to the array value of a database item, since their types can never stem from the same type declaration.

For this case a less restrictive compatibility concept, called *structural compatibility*, is applied. Two types are structurally compatible, if they have identical structure, e.g. for array types the index types, index range and element types must be identical.

Example

```
type Matrix = array (1 .. 3, 1 .. 3) of REAL;
variable M : Matrix;
...
M := \APM\EGSE\DATA\M1;
```

The assignment is legal, if `M` and the database item are structurally compatible, i.e. if the type of the database item is structurally identical to the type `Matrix`.

Note that in order to assure equal types, the type of the database item can be *inherited* (see 4.8.5).

Example

```
type Matrix = type of \APM\EGSE\DATA\M1;  -- type inherited from database
variable M : Matrix;
...
M := \APM\EGSE\DATA\M1;  -- M and \APM\EGSE\DATA\M1 have the same type
```

4.9 Variable Declarations

A variable declaration associates a variable with a unique identifier and a data type. The type determines the set of values that the variable may assume and the operators that are applicable; it also defines the structure of the variable (see 4.8, Type Declaration).

In a variable declaration, either a predefined type, as in example (a), or a named user–defined type may be specified, as in example (b). String variables may be declared directly, where the maximum length is specified within the variable declaration, see example (c).

A variable declaration is introduced by the reserved word **`variable`**, followed by the name of the variable (an identifier). A colon (”**`:`**”) introduces the type. Optionally, the variable may be given a constant initial value indicated with the assignment symbol (”**`:=`**”). If no initial value is given, the value of the variable is undefined. Using an uninitialized variable is an error that is not automatically detected.

Formal Syntax

Variable_Declaration	=	”variable” Identifier ”:” Variable_Type [”:=” Constant_Expression] ”;”
Variable_Type	=	Qualified_Identifier [Unit] | String_Type | ”statecode” | ”pathname” [”.” ”*”]
Identifier	=	*see 4.1.2.2*
Constant_Expression	=	*see 4.12*
Qualified_Identifier	=	*see 4.3.4*
Unit	=	*see 4.11 and 4.1.2.10*
String_Type	=	*see 4.8.2.5*

Examples

(a) variable declaration with predefined types:

```
variable Index: INTEGER;
variable State: statecode;
```

(b) variable declaration with user–defined types:

```
type Primary_Color = (Red, Green, Blue);
variable Color : Primary_Color;
```

(c) string variable declaration:

```
variable s: string (80);
```

4.10 Alias Declarations

An alias declaration associates a simple name (an identifier) with a complex name that includes a pathname, and thus provides a shorthand for the long name. The alias name may then be used instead of the corresponding long name. The long name may be one of the following:

- a pathname

```
alias Lib = \EGSE\VICOS\SYSTEM\GROUND_LIBRARY;
```

  The pathname may be incomplete, i.e. denote a virtual node in the database name tree. In this case the alias may be used as a prefix in other pathnames:

```
alias System = \EGSE\VICOS\SYSTEM;
alias Lib    = System\GROUND_LIBRARY;
```

- a subitem pathname

```
alias Input_1 = \APM\SIMULATION\PAYLOAD\EQUIPMENT_UNIT_A.INPUT_1;
```

- a qualified name (an identifier prefixed with the pathname of an imported module, e.g. a library)

```
import \EGSE\VICOS\SYSTEM\GROUND_LIBRARY;
alias Issue = \EGSE\VICOS\SYSTEM\GROUND_LIBRARY.Issue;
```

Note that for any compilation unit the name (identifier) of the unit is implicitly declared as an alias for the pathname of the unit (see 4.16.1, 4.16.2 and 4.16.3). When importing a library or some other module, the name alias of the module is imported together with the other identifiers from the module and may then be used instead of the long pathname:

```
import \EGSE\VICOS\SYSTEM\GROUND_LIBRARY;
alias Issue = Ground_Library.Issue;
```

Formal Syntax

Alias_Declaration	=	”alias” Identifier ”=” Name ”;”
Name	=	*see 4.3.4*
Identifier	=	*see 4.1.2.2*

Please note that the Mission Database may itself predefine a set of aliases, called “nicknames” (see 4.3.3). These predefined aliases may be used like any user declared alias.

4.11 Unitized Values and Types

The types and values introduced in the previous sections were all referring to *unitless values*. UCL also provides the facility for defining and operating on *unitized values*, i.e. real or integer values associated with a *unit of measure*. This facility improves the readability of the source code and at the same time enables additional semantic checking at compilation time.

4.11.1 Units of Measure

Every measurement system defines a number of *base units* each describing a physical quantity. The currently world–wide used unit system is the SI system, which defines 7 base units: *meter* to measure the length, *second* to measure the time, *kilogram* to measure the mass etc. It is standardized under ISO 1000. The units of measure in UCL are based on this International Standard.

All other units can be derived from the base units by multiplication or division of two or more base units or by multiplication and/or addition with constants. Such units are called *derived units*. For example, the unit *volt* is defined as:

`(m² * kg) / (s³ * A)` *or* `m² * kg¹ * s⁻³ * A⁻¹`.

The second expression shows that derived units are products of base units raised to positive or negative integer power.

The same physical quantity can be described by more than one unit. For example, the length can be measured in meters or in inches. Such units are called *commensurable units*. They are assignment compatible. The UCL compiler automatically converts between commensurable units. This is explained later in more detail.

UCL predefines the seven base units defined by the international system of units (Système Internationale d'Unités, SI), but allows the user to define additional base units. It can handle all units derived from these base units by means of the usual arithmetical operations, as shown above. Additional unit names can then be defined for units derived from existing units by means of UCL unit declarations.For a definition of the SI base units, see the ISO 1000 standard, annex B.

Quantity	Unit	SI symbol	UCL symbol
length	meter	m	`m`
mass	kilogram	kg	`kg`
time	second	s	`s`
electric current	ampere	A	`A`
temperature	kelvin	K	**`Kabs`**
temperature difference	kelvin	K	**`K`**
amount of substance	mole	mol	`mol`
luminosity	candela	cd	`cd`

Note the distinction made between *temperature* and *temperature difference*, expressed with different UCL unit symbols `Kabs` (= K absolute) and `K`. This is necessary to allow conversions between e.g. °C and K. The *temperature* relationship 0 °C = 273.15 K is not reflected for *temperature differences*, where 1 °C = 1 K.

4.11.2 Counting Units

Beside the normal physical units, UCL supports *counting units* like *pair* or *dozen*. These are pseudo–units that represent a grouping of non–dimensional quantities. All counting units are commensurable with one another and with pure scalar values. They are resolved to scalar values whenever appropriate.

4.11.3 Predefined Units

Within a target system based on UCL, there are two groups of predefined units:

- **UCL predefined**

  UCL itself predefines only the names of the seven SI base units. These are available and identical in any UCL based system.

- **Project predefined**

  The target system may predefine further, project specific, unit names (both new base units and derived units) by means of the Mission Database. These units must not overwrite SI base units.

Both of these groups of units are automatically available in any UCL compilation unit and need not be declared by the user.

4.11.4 Unit Declaration

New base units can be defined in the form

```
unit [unit_identifier];
```

and new identifiers for derived units can be declared in a unit declaration, which takes the form

```
unit [unit_identifier] = [unit_expression];
```

Units cannot be declared local to a procedure or function. Unit names are always global. For convenience it is, however, possible to redeclare an existing unit name, but only identical to the existing unit.

Formal syntax

Unit_Declaration	=	”unit” ”[” Unit_Identifier ”]” [”=” Unit] ”;”
Unit	=	”[” Unit_Expression ”]”
Unit_Expression	=	[Numerator [”/” Denominator] [”+” Offset | ”–” Offset]]
Offset	=	Number [”/” Number]
Numerator	=	Unit_Term | ”(” Unit_Term ”)”
Denominator	=	Number | Unit_Factor | ”(” Unit_Term ”)”
Unit_Term	=	[Number] Unit_Factor { Unit_Factor } | Number
Unit_Factor	=	Unit_Identifier { Digit }
Unit_Identifier	=	Letter { Letter }

4.11.5 Unit Syntax

A unit is expressed with a syntax production defined somehow like a string with an imposed syntax, enclosed in brackets '[' and ']'.

Note that no space may be used between the Unit_Identifier and the following Digit in the syntax production Unit_Factor. A Unit_Identifier is composed of letters only.

The following conventions are used for unit expressions:

Identifiers are case sensitive:	`mA, MeV`
Multiplication is denoted by a space:	`kg m, N m, 1000 m`
Division is denoted by a slash ("/"):	`m/s, 1/s`
Exponentiation is denoted by an attached digit:	`m3, s2`
For numbers, reals or integers may be used:	`0.0254 m, 10 degC/18 – 320/18`
An expression with + or – creates an *absolute* unit: (it defines the offset of the two zero points)	`Kabs + 273.15`
A single number defines a counting unit:	`12 (for dozen)`

Parentheses may be used to clarify the order of evaluation, but no nested parentheses are supported to keep a unit expression as clear as possible. Also, only one division (both for the unit and the offset) is allowed in a unit expression, thus avoiding ambiguous expressions.

Please note that factors and offsets may only be used within the defining unit expression in a unit declaration, but not as a unit literal otherwise. Thus, while the forms `[1000 m]` and `[12]` are legal as the defining part in the declarations

```
unit [km]  = [1000 m];
unit [doz] = [12];
```

they are not in expressions like `1.0 [1000 m]` and `2.0 [12]`. Instead, use `1.0 [km]` and `2.0 [doz]`.

Examples

```
unit [bit];                              -- new base unit bit
unit [Byte] = [8 bit];                   -- unit Byte derived from bit
unit [N]    = [kg m/s2];                 -- Newton
unit [Pa]   = [N/m2];                    -- Pascal
unit [km]   = [1000 m];                  -- kilometer
unit [degC] = [Kabs + 273.15];           -- degrees Celsius
unit [degF] = [10 degC / 18 – 320 / 18]; -- degrees Fahrenheit
unit [doz]  = [12];                      -- dozen
```

4.11.6 Unitized Types

In type declarations, a unit clause is used to indicate a unitized–value type (or *unitized type*, for short). The associated type is given first, then a unit expression. Variables and constants declared to be of a unitized type are bound to values of the respective physical dimension.

Examples

```
type Furnace_Temperature = REAL [Kabs];
type Coolant_Temperature = REAL [degC];
type Power               = REAL [m2 kg/s3];
type Dozen               = REAL [doz];

constant Freezing_Point : Furnace_Temperature := 0.0 [K];
variable Curr_Power     : Power               := 0.0 [m2 kg/s3];
variable Count          : Dozen;
```

Only `REAL`, `LONG_REAL`, `INTEGER` and `UNSIGNED_INTEGER` types may be ”unitized”, i.e. associated with a unit of measure via a unit expression. A unitized type may be part of a structured type (e.g. array, record); however, the structured type itself must not be unitized. Thus, in the following example:

```
type Rval_Array        = array (Lo .. Hi) of REAL;
type Temperature_Array = Rval_Array [degC];         -- not allowed !!!
type Voltage_Array     = array (Lo .. Hi) of REAL [V];
```

the type declaration for `Temperature_Array` is invalid; the type `Voltage_Array`, on the other hand, is a valid UCL construct.

In type, variable and constant declarations, a unit clause is used to indicate a unitized–value type (or *unitized type*, for short). The associated type is given first, then a unit expression.

4.11.7 Unitized Variables and Constants

UCL also allows variables and constants to be associated directly with a unit of measure. This is syntactically analogous to declaring an unnamed structured type. For example,

```
constant Max_Speed : REAL [m/s] := 1000.0 [m/s];
variable Mass      : REAL [kg];
```

The variable declaration associates the variable `Mass` with the mass unit kilogram. This variable is compatible with other variables of type `REAL` which are associated with a unit commensurable to kilogram. Unitized types are forbidden in such declarations. For example,

```
variable Temp: Furnace_Temperature [degC];
```

is not allowed, because `Furnace_Temperature` is already a unitized type.

4.11.8 Compatibility of Unitized Types

Two unitized types are compatible, if the types are compatible without the unit and the two units are commensurable, i. e. of the same physical dimension. A type conversion both converts the value to the representation of the other type and rescales the value to the other unit, if necessary. Types that are implicitly converted to each other (`INTEGER`/`UNSIGNED_INTEGER` and `REAL`/`LONG_REAL`) will also be implicitly converted, if they are associated with a measurement unit.

4.11.9 Unitized Literals & Constants

In the UCL source code, unitized values may be formed by suffixing a real or integer value with their appropriate unit, which is written in brackets in the syntax described below.

As stated above, it is possible to derive other units using the `*` and `/` operators, offsets and exponentiations with integer numbers. Such *unit expressions* (described below) must be enclosed in square brackets. They follow a syntax which allows unit expressions as close as possible to the normally used scientific syntax, as defined in ISO 1000.

Examples:

```
10.0 [degC],  1.3E+3 [K],  35.5 [m/s],  100.0 [m2 kg/s2], 2.0 [doz]
```

4.11.10 Expressions with Unitized Values

Unitized values can be used in arithmetical expressions according to the following rules:

- In assignments both operands must have commensurable units. The UCL compiler converts units as necessary. Example:

  ```
  variable X : REAL [Kabs];
  X := 25.0 [degC];    -- automatic conversion from Celsius to Kelvin
  ```

- In additions or subtractions both operands must have commensurable units. The UCL compiler converts units as necessary, choosing the “smaller” (finer) of the two units as the resulting unit. The other operand is adapted (converted) to this smaller unit before the operation is performed. Note that counting units are commensurable with unitless values. When different counting units are involved, those operands are converted to scalars.

  Special treatment is needed to handle conversion between temperatures correctly, as there are temperatures and temperature differences. Also, special conversion rules apply to measurements of temperatures because the subtraction of two temperatures yields a temperature difference, for any other unit it results in the unit itself. Examples:

  ```
  100.0 [Kabs] – 90.0 [Kabs] = 10.0 [K]
  100.0 [Kabs] – 10.0 [K]    = 90.0 [Kabs]
  ```

  `100.0 [Kabs] + 10.0 [Kabs]` **is invalid!**

  ```
  100.0 [K] – 10.0 [K] =  90.0 [K]
  100.0 [K] + 10.0 [K] = 110.0 [K]
  ```

- Likewise, for comparison operations, the operands are adapted to the smaller of the involved units. When different counting units are involved, those operands are converted to scalars.

- The same holds for the `MIN` and `MAX` functions (see 4.15): All operands are adapted to the smallest of the involved units, values with counting units are converted to scalars.

- In multiplications the operands can be of different units. The result is another unit or a unitless value.Values with counting units are converted to scalars. Examples:

  (a) force * length → work, or in units: Newton * Meter → m^2 * kg * s^{-2} = Joule
  (b) duration * frequency → unitless value.

  But note that the resulting unit is always normalized to contain base units only.

- In divisions the operands can be of different units. The result is another unit or a unitless value. Operands with commensurable units are adapted to the smaller of the involved units before performing the operation. Values with counting units are converted to scalars. The resulting unit is always normalized to contain base units only.
- Any unitized value can be multiplied with or divided by a unitless value. The resulting unit is the unit of the unitized operand.
- In exponentiations the unitized values can be raised to constant integer powers only. This operation is equivalent to a multiplication or division of values with the same unit. It results in a new unit, which will be normalized to contain base units only. Operands with counting units are converted to scalars.

Examples:

```
unit [bit];   -- base type bit
unit [px];    -- base type pixel

unit [Byte] = [8 bit];

unit [degC] = [Kabs + 273.15];
unit [km]   = [1000 m];
unit [h]    = [3600 s];
unit [mi]   = [1.609 km];

unit [doz]  = [12];

type Temperature  = REAL [degC];
type Voltage      = REAL [V];
type Speed        = REAL [m/s];
type Acceleration = REAL [m/s2];

type Dozen        = REAL [doz];

type Color_Depth  = INTEGER [bit/px];

constant Max_Voltage: Voltage := 100.0 [V];

variable T : Temperature;
variable V : Voltage;
variable S : Speed;
variable A : Acceleration;
variable M : REAL [m];
variable D : Dozen;

variable C : Color_Depth;

...

T := 37.5 [degC];
T := 10.5 [V];          -- invalid because T is of type TEMPERATURE;
D := 24.0;              -- assigns 2.0 [doz]

V := \A\B\C;            -- valid only if \A\B\C refers to an MDB item
                        -- of the appropriate type (’voltage’ measurement)

if V > 5.0 [V] then ...

M := 50.0 [km];         -- commensurable units are converted by the compiler
S := M / 0.5 [h];       -- [m] / [h]   -> [m/s]
A := S / 2.0 [h];       -- [m/s] / [h] -> [m/s2]

C := 24 [bit/px];
C := 3 [Byte/px];       -- implies a conversion [Byte/px] -> [bit/px]
```

4.11.11 Unitized Integer Values

Integer values with units require special care:

- In expressions with unitized integer or unsigned integer values, integer arithmetic applies, like for non–unitized values.

```
5 [m] / 2 [s] = 2 [m/s]
```

- When converting between commensurable units, the resulting value will be rounded to the nearest integer:

```
1 [m/s] (→ 3.6 [km/h])   → 4 [km/h]
3 [mi]  (→ 4.82803 [km]) → 5 [km]
```

- Expressions with mixed commensurable units may imply inadvertent unit conversions that lead to loss of precision and, even worse, to unwanted zero values. The rules given in 4.11.10 try to reduce such loss as much as possible, but cannot avoid it completely:

```
variable K : INTEGER [km];

K := (1 [km] + 1 [mi]) / 2;   -- = (1 [km] + 2 [km]) / 2 = 2 [km]
K := 1 [km] – 400 [m];        -- = 1000 [m] – 400 [m] = 600 [m] = 1 [km]

K := 1 [km/h] * 1 [s]         -- = 1 [1000/3600 m] = 0 [m] = 0 [km]
```

- Unit conversions and normalizations can much more easily lead to overflows or underflows than for unitized real values.
- Divisions whose operands have units that are not commensurable but have common base units like

```
1 [km/h] / 1 [min]
```

  present a specific problem. Due to its internal unit representation, the compiler cannot decide how to adapt these units to each other. Expressions of this kind are therefore not allowed.

  This does not affect division of “unrelated” units like

```
1 [kg m] / 1 [s] = 1 [kg m/s]
```

- The right operand of the % (mod) operator must not be unitized:

```
5 [m] % 2 = 1 [m]   -- OK
5 [m] % 2 [s]       -- not allowed!
```

- The increment parameter of the INC and DEC functions (see 4.15) must not be unitized:

```
X := 2 [m/s];
INC (X);           -- OK:  X = 3 [m/s]
INC (X, 2);        -- OK:  X = 5 [m/s]
INC (X, 2 [m/s]);  -- not allowed!
```

4.12 Expressions

Expressions are syntactical constructs used to calculate values of variables or generate new values. They represent combinations of *operands* and *operators* whereby an operand may itself contain an expression; i.e. expressions can be constructed recursively.

At runtime, when the expression is evaluated, the respective current values denoted by the operands are combined using the specified operators to yield a new value. An error will occur if the computed value cannot be represented on the current target machine (e.g. division by zero).

The arithmetical operators, e.g., applied to `INTEGER`, `UNSIGNED_INTEGER`, `REAL` or `LONG_REAL` operands form *arithmetical expressions*; whereas the logical, relational and some set operators together with their respective operands yield *Boolean expressions*.

If an MDB object is used in an expression (through a *pathname designator*), its meaning depends on the expected target type:

- If the target type is a pathname type, the pathname stands for itself as a reference to the item.
- Otherwise the pathname denotes the runtime value of the item. The item must then be accessible in READ or READ/WRITE mode (see section 4.2).

The same holds, analogously, for subitem pathnames.

Formal Syntax

```
Expression      = Relation { "&" Relation } |
                  Relation { "|" Relation } |
                  Aggregate
Aggregate       = "(" [ Component { "," Component } ] ")"
Component       = [ Identifier ":" ] Expression
Relation        = Simple_Expression
                  [ "="   Simple_Expression |
                  "<>"  Simple_Expression |
                  "<"   Simple_Expression |
                  "<="  Simple_Expression |
                  ">"   Simple_Expression |
                  ">="  Simple_Expression |
                  "in"  Simple_Expression ]
Simple_Expression = [ "+" | "–" ] Term { "+" Term | "–" Term }
Term            = Factor { "*" Factor | "/" Factor | "%" Factor }
Factor          = Primary [ "**" Factor ]
Primary         = Number [ Unit ] |
                  String |
                  Character |
                  Set_Constant |
                  Date [ Time ] | Time
                  Statecode |
                  Designator |
                  Function_Call |
                  Type_Conversion |
                  "(" Expression ")" |
                  "~" Primary
Designator      = Name { "." ( Identifier | "(" Expression_List ")" ) }
```

Slice	=	Designator ”(” Expression “..” Expression ”)”
Name	=	*see 4.3.4*
Expression_List	=	Expression { ”,” Expression }
Number	=	Simple_Integer | Based_Integer | Real
Identifier	=	*see 4.1.2.2*
Simple_Integer	=	*see 4.1.2.4.1*
Based_Integer	=	*see 4.1.2.4.1*
Real	=	*see 4.1.2.4.3*
String	=	*see 4.1.2.7*
Character	=	*see 4.1.2.6*
Set_Constant	=	*see 4.8.2.3*
Date	=	*see 4.1.2.8*
Time	=	*see 4.1.2.8*
Statecode	=	*see 4.1.2.5*
Function_Call	=	*see 4.12.3*
Type_Conversion	=	*see 4.12.4*
Unit	=	*see 4.11*

Constant expressions

An expression may be required to be constant. Constant expressions are expressions whose operands are all constants or constant expressions, such that its value can be determined at compilation time.

Constant_Expression = Expression

4.12.1 Operands

The various components of an expression are called *operands*. Valid operands are

- literal constants (see 4.1)
- declared or imported constants (see 4.4 and 4.2)
- declared or imported variables (see 4.9 and 4.2)
- array components (see 4.8.2.1)
- array aggregates (see 4.8.2.1)
- substrings (*slices*) (see 4.8.2.5)
- record components (see 4.8.2.2)
- MDB items and subitems (see 4.3.6)
- function calls (see 4.12.3)
- type conversions (see 4.12.4)
- expressions

4.12.2 Operators

There exists a *precedence hierarchy* among operators. It determines the order of evaluation of operands in an expression. The "not" operator and the exponentiation operator have the highest precedence, followed by the so–called multiplying operators, the so–called adding operators and the relational operators. Boolean operators have the lowest precedence. Sequences of operators of the same precedence are evaluated from left to right, except for sequences of exponentiation operators which are evaluated from right to left. The order of evaluation may be altered by using parentheses.

The precedence of operators is:

(highest)	1)	`~ + – **`	Boolean negation, unary +/–, exponentiation
	2)	`* / %`	multiplying operators
	3)	`+ –`	adding operators
	4)	`= <> < <= > >= in`	relational operators
(lowest)	5)	`& |`	Boolean operators

Examples

```
(I + J) * K      -- INTEGER/UNSIGNED_INTEGER expression
~A | B           -- same as (~A) | b
I = 1 | J = 1    -- same as (I = 1) | (J = 1)
A**N + B         -- same as (A**N) + B
A**N**M          -- same as A**(N**M);   -- ** associates right to left
A*N*M            -- same as (A*N)*M;     --  * associates left to right
```

4.12.2.1 Arithmetical Operators

Symbol	Operation
+	addition
–	subtraction
*	multiplication
/	division
**	exponentiation
%	modulus

All operators apply to variables of the type `INTEGER/UNSIGNED_INTEGER` and their subranges. The operators `+`, `–`, `*` , `**` and `/` apply to `REAL` and `LONG_REAL` variables. The operators `+`, `–`, `*` and `/` apply to `TIME` and `DURATION` variables as well. For exponentiation, however, the right operand (called the exponent) must be of the type `INTEGER/UNSIGNED_INTEGER` and the result type is the type of the left operand. Further, the operators `+` and `–` may be used as unary operators to denote the sign of a number.

Integer division and modulus are defined by the relation:

```
A = (A/B)*B + (A % B)
```

where (A % B) has the sign of A and an absolute value less than the absolute value of B. During divisions and modulus operations, a runtime error occurs if the right operand is zero.

4.12.2.2 Concatenation Operator

Symbol	Operation
+	string concatenation

This operator joins two strings of the same base type, i.e. string literals, string constants or variables, or string elements (characters, bytes). The result is a string containing the left and right operands concatenated.

4.12.2.3 Logical or Boolean Operators

Symbol	Operation
|	logical or
&	logical and
~	logical negation (unary operator)

Their operands must have the type `BOOLEAN`. The definition of the Boolean operators & and | is as follows:

```
p  &  q  =  if p then q     else FALSE
p  |  q  =  if p then TRUE  else q
```

This definition implies that the second operand will not be evaluated if the result is already known from the evaluation of the first operand (*short–circuit evaluation*).

4.12.2.4 Relational or Comparison Operators

Symbol	Operation
=	equal
<>	unequal
<	less
<=	less or equal
>=	greater or equal
>	greater

These operators are applicable to all ordered types, i.e. `INTEGER`, `UNSIGNED_INTEGER`, `REAL`, `LONG_REAL`, `CHARACTER`, `TIME` and enumeration types. They may also be used for string operands and (except < and >) for set operands. The equality operands = and <> can also be applied to statecode, set, pathname and subitem pathname operands, as well as to `BYTE`, `WORD` and `LONG_WORD` operands.

4.12.2.5 Set Operators

Symbol	Operation
in	contained in
+	set union
–	set difference
*	set intersection
/	symmetric set difference
<=	inclusion (left operand included in right operand)
>=	inclusion (right operand included in left operand)

4.12.3 Function Calls

Like a procedure, a function is invoked by its name followed by the appropriate actual parameters. In contrast to a procedure call, which denotes an action, a function call returns a value and corresponds therefore to an expression. The specification of actual parameters in function calls follow the same syntax rules as in procedure calls (see 4.13.2).

Formal Syntax

Function_Call	=	Qualified_Identifier [Actual_Parameters]
Actual_Parameters	=	”(” [Parameter { ”,” Parameter }] ”)”
Parameter	=	[Identifier ”:”] Expression
Qualified_Identifier	=	*see 4.3.4*
Identifier	=	*see 4.1.2.2*
Expression	=	*see 4.12*

Examples

```
X := Average (First_Val, Last_Val);     -- function ”Average”
if Is_Empty (Buffer) then ...           -- Boolean function ”Is_Empty”
```

4.12.4 Type Conversions

Expressions of one type may be converted to certain other types. UCL provides the following types of conversions:

- high level conversions between closely related types,
- low level conversions to circumvent typing rules,
- string conversions to convert between values and their textual representation.

4.12.4.1 High Level Conversions

High level conversions preserve the strong typing properties of the language, they can only be done between closely related types.

A type conversion is specified by prefixing the expression to be converted (in parentheses) with the name of the target type (one of the above). A type conversion is an expression, syntactically it corresponds to a function call.

Note that type conversions done automatically by the compiler may be written explicitly as well.

Type_Conversion = Qualified_Identifier ”(” Expression ”)” |
String_Conversion

String_Conversion = *see 4.12.4.3*

Qualified_Identifier = *see 4.3.4*

Expression = *see 4.12*

Formal Syntax

Scalar and byte string to BYTE/WORD/LONG_WORD Conversion

```
BYTE     (8-bit_scalar_value or 1_element_byte_string)
WORD     (32-bit_scalar_value or 4_element_byte_string)
LONG_WORD(64-bit_scalar_value or 8_element_byte_string)
```

These conversions are implicitly carried out whenever necessary, see 4.8.1.7. They may, however, also be done explicitly.

Examples:

```
B := BYTE ('x');          -- character to BYTE
W := WORD (#"FF000000");  -- byte string to WORD
L := LONG_WORD ($OFF);    -- statecode to LONG_WORD
```

REAL/LONG_REAL to INTEGER/UNSIGNED_INTEGER Conversion

```
INTEGER (real_value)
INTEGER (long_real_value)
UNSIGNED_INTEGER (real_value)
UNSIGNED_INTEGER (long_real_value)
```

The result of this type conversion is the integer value obtained by truncating the value expressed by *real_value* or *long_real_value* toward zero.

Examples:

```
INTEGER (3.6);   -- value is  3
INTEGER (1.5);   -- value is  1
INTEGER (-1.3)   -- value is -1
```

INTEGER/UNSIGNED_INTEGER to REAL/LONG_REAL Conversion

```
REAL      (integer_value)
LONG_REAL (integer_value)

REAL      (unsigned_integer_value)
LONG_REAL (unsigned_integer_value)
```

The result of this type conversion is the floating point value corresponding to the value of *integer_value* or *unsigned_integer_value*.

Examples:

```
REAL (I + 5);   -- value converted to floating point
```

CHARACTER to INTEGER/UNSIGNED_INTEGER Conversion

```
INTEGER (character_value)

UNSIGNED_INTEGER (character_value)
```

The result of this type conversion is the integer corresponding to the ordinal value of *character_value* (i.e. its sequence number) in the ASCII character set.

Examples:

```
INTEGER ('A');   -- yields 65
INTEGER ('E');   -- yields 69
```

INTEGER/UNSIGNED_INTEGER to CHARACTER Conversion

```
CHARACTER (integer_value)

CHARACTER (unsigned_integer_value)
```

The result of this type conversion is the character whose ordinal number (in the ASCII character set) is *integer_value* or *unsigned_integer_value*, resp.

Examples:

```
CHARACTER (65);   -- returns character 'A'
CHARACTER (69);   -- returns character 'E'
```

Enumeration to INTEGER/UNSIGNED_INTEGER Conversion

```
INTEGER (enumeration_value)

UNSIGNED_INTEGER (enumeration_value)
```

The result of this type conversion is the integer corresponding to the position of *enumeration_value* in the enumeration list. (The first element in the list has the position 0).

Examples:

```
type Color = (Red, Green, Blue);
...
INTEGER (Green);   -- returns 1
INTEGER (Red);     -- returns 0
```

INTEGER/UNSIGNED_INTEGER to Enumeration Conversion

```
enumeration_type (integer_value)
enumeration_type (unsigned_integer_value)
```

The result of this type conversion is the enumerated value (literal) whose position in the enumeration list is equal to *integer_value* or *unsigned_integer_value*. (The first element in a list has the position 0). A runtime error occurs if *integer_value* < 0 or *integer_value* > INTEGER (MAX(*enumeration_type*)).

Examples:

```
type Color = (Red, Green, Blue);
...
Color (1);   -- returns Green
Color (0);   -- returns Red
```

BITSET to INTEGER/UNSIGNED_INTEGER Conversion

```
INTEGER (bitset_value)
UNSIGNED_INTEGER (bitset_value)
```

The result of this type conversion is the 32–bit integer whose internal representation corresponds to the (32 member) *bitset_value*.

Examples:

```
INTEGER ({0, 1, 2});   -- returns 7 (or 2^0 + 2^1 + 2^2)
```

INTEGER/UNSIGNED_INTEGER to BITSET Conversion

```
BITSET (integer_value)
BITSET (unsigned_integer_value)
```

The result of this type conversion is the BITSET variable whose internal representation corresponds to the 32–bit integer value *integer_value* or *unsigned_integer_value*.

Examples:

```
BITSET (7);   -- returns the BITSET constant {0, 1, 2}
```

Unitized Type to Non–Unitized Type Conversion

```
REAL (unitized_value)
LONG_REAL (unitized_value)
INTEGER (unitized_value)
UNSIGNED_INTEGER (unitized_value)
```

The result of type conversion is the raw (unitless) real or integer value of *unitized_value*. A conversion between real and integer values is done, if necessary. For counting units the conversion yields the effective quantity.

Examples:

```
REAL (3.6 [km/h])      -- returns 3.6
INTEGER (3.6 [km/h])   -- returns 3 (conversion REAL to INTEGER truncates)
REAL (2.5 [doz])       -- returns 30.0
INTEGER (2.5 [doz])    -- returns 30
```

Non–Unitized Type to Unitized Type Conversion

```
unitized_type (real_value)
unitized_type (long_real_value)
unitized_type (integer_value)
unitized_type (unsigend_integer_value)
```

The result of type conversion is the unitless value expressed in units of *unitized_type*. A conversion between real and integer values is done, if necessary.

Examples:

```
type Velocity = REAL [m/s];
type Distance = INTEGER [km];
type Dozen    = REAL [doz];

...

Velocity (3.6)   -- returns 3.6 [m/s]
Distance (3.6)   -- returns 3 [km]
Dozen (24.0)     -- returns 2.0 [doz]
```

Unitized Type to Unitized Type Conversion

```
unitized_type (unitized_value)
```

The type of *unitized_value* must be commensurable with *unitized_type*. The result of this type conversion is the value of *unitized_value* expressed in units of *unitized_type*. A conversion between real and integer values is done, if necessary.

Example:

```
unit [mph] = [1609 m/h];    -- miles per hour

type Mph   = REAL [mph];

variable Speed : REAL [m/s];
variable X     : REAL;

...

X := REAL (Mph (Speed));   -- Speed is converted to Mph, then to REAL
```

INTEGER/UNSIGNED_INTEGER Conversion

```
INTEGER (unsigned_integer_value)
UNSIGNED_INTEGER (integer_value)
```

These conversions are implicitly carried out whenever necessary. They may, however, also be done explicitly. A check will be done that the allowed range of the target type is not violated.

Examples:

```
Unsigned_Integer_Value := UNSIGNED_INTEGER (Integer_Value) * 2;
```

REAL/LONG_REAL Conversion

```
REAL (long_real_value)
LONG_REAL (real_value)
```

These conversions are implicitly carried out whenever necessary. They may, however, also be done explicitly.

Examples:

```
Long_Real_Value := LONG_REAL (Real_Value) * 1.0E50;
```

Type to Subtype Conversion

subtype (*value*)

The base type of *value* must be the same as the base type of *subtype*. The effect of this conversion is to interpret the value as belonging to the subtype.

A check will be done to insure that the value does not violate possible constraints of the subtype. If the value is constant, and thus known to the compiler, the compiler will do the check, otherwise it will be carried out at runtime. For pathname and subitem pathname types, the check will always be done at compile time: it makes sure that the target *subtype* is not more constrained than the type of *value*.

Note: A special case is the conversion of a type to itself, which is allowed but has no effect.

Example:

```
type Short  = INTEGER (0 .. 2 ** 16 – 1);
type Switch = statecode ($LOW, $MEDIUM, $HIGH);
type Label  = string (10);

variable I : INTEGER;
variable C : statecode;
variable S : string (80);

...

I := Short (I);                 -- make sure I is a short integer
C := Switch (C);                -- make sure C is a Switch value
Send_Message (Label(S), ...);   -- make sure S is not longer than a Label
```

4.12.4.2 Low Level Conversions

Low level conversions circumvent the UCL strong typing concept. They allow to convert a value of one type into any other type, as long as the sizes fit. A low level conversion does not change the representation in memory, but just regards the same byte/word representation to be of a different type.

Low level conversions are done using the special predefined function UNTYPED. The function call

```
UNTYPED (expression)
```

removes the type from the expression and makes it convertible to any other type. A conversion is implicitly perfomed when assigning the untyped expression to a variable or passing it as a parameter to a subprogram, or by converting it explicitly into a specific target type with the usual conversion notation.

For strings, only the string contents (excluding the length field) is converted. The size of a string value to be converted is the current string length, and the length of a string target is set according to the size of the value.

Example:

```
type Rec = record ... end record;

variable R : Rec;
variable S : string (10) of BYTE;
variable I : INTEGER;
variable P : pathname;

procedure Proc (A : array of WORD); begin ... end Proc;

...
I := UNTYPED (P);    -- get SID of pathname
P := UNTYPED (I);    -- get pathname of SID

S := UNTYPED (R);    -- convert record to string of bytes
R := UNTYPED (S);    -- convert string of bytes to record

Proc (UNTYPED (R));  -- pass record as an array of words

if I = INTEGER (UNTYPED(P)) then ... end if;   -- explicit conversion
```

For a low level conversion, the size of the value to be converted must fit the size of the target type. A check is done to enforce this restriction.

- If the target is a string, the size in bytes of the value must not be greater than the maximum size of the string. For open string parameters there is no such restriction.
- If the target is an open array parameter, there is no upper limit for the size, but the size in bytes must be a multiple of the array element size. A check is performed to enforce this restriction.
- For other target types the size of the value must be identical to the size of the target.

Example:

```
-- declarations as above

S := UNTYPED (R);    -- size of R must not be greater than 10
I := UNTYPED (S);    -- length of S must be 4

Proc (UNTYPED(S));   -- length of S must be a multiple of 4
Proc (UNTYPED(R));   -- no check
```

4.12.4.3 String Conversions

Values of scalar types and string types (both character strings and byte strings) may be converted to their textual representation, according to UCL syntax, by just converting them to some character string type, using the high level conversion syntax. Two syntactical forms are available:

```
string (expression)
typename (expression)
```

The first form creates a general string without any size restrictions. The second form creates a string of the indicated specific type. A check will be performed that the resulting string fits the declared size constraint of the type.

For pathname types the two forms may be preceeded by an additional modifier keyword **pathname** or **alias**:

```
pathname string (expression)
alias string (expression)
pathname typename (expression)
alias typename (expression)
```

With the **pathname** keyword the pathname will be chosen as the textual representation, the **alias** keyword requests the database alias to be generated, instead. If the item does not have a database alias, the result will be an empty string. The pathname will be used as the default text representation, if neither of the two keywords is present.

For unitized types and for pathname types the unit itself may be obtained as a string with the following forms, the unit is not enclosed in brackets:

```
unit string (expression)
unit typename (expression)
```

All forms allow to specify simple string formatting:

`string (expression, width)` `typename (expression, width)`	for all types
`string (expression, width, aft)` `typename (expression, width, aft)`	for REAL, LONG_REAL and TIME
`string (expression, width, aft, exp)` `typename (expression, width, aft, exp)`	for REAL and LONG_REAL

width specifies a minimum total width of the resulting string. Negative values request left justification, positive values right justification. If the string is larger than the specified minimum width, the actual size will be taken.

aft specifies the number of digits after the decimal point. For TIME values *aft* affects the fraction part of the seconds component, 0 suppresses the fraction part. Note that the syntax of REAL and LONG_REAL values requires at least one digit after the decimal point.

exp specifies the width of the exponent in REAL/LONG_REAL floating point representation. For *exp* = 0 fix point notation (no exponent) will be used.

Omitted format parameters default to 0. If all format parameters are 0 (or omitted), UCL standard format will be used.

Formal Syntax

String_Conversion = [“pathname” | “alias” | “unit”] Qualified_Identifier ”(” Expression [“,” Format] ”)” |
[“pathname” | “alias” | “unit”] “string” ”(” Expression [“,” Format] ”)”

Format = Width [”,” Aft [“,” Exp]]

Width, Aft, Exp = Expression

Qualified_Identifier = *see 4.3.4*

Expression = *see 4.12*

Examples:

```
string (123)                          → 123
'[' + string (123, -5) + ']'          → [123  ]
'[' + string (123, 5) + ']'           → [  123]

string (1.23)                         → 1.23
string (1.23, 0, 6)                   → 1.230000
string (1.23, 0, 6, 4)                → 1.230000E+00

string (13:30)                        → 13:30:00
string (13:30, 0, 3)                  → 13:30:00.000

string (#"FFFFFFFF")                  → #"FF FF FF FF"
'[' + string ("abc", 5) + ']'         → [  abc]

pathname string (Path_Variable)       → \some\path\name
alias string (Path_Variable)          → xyz  (if xyz is the database alias of \some\path\name)
unit string (Path_Variable)           → m/s  (if \some\path\name has the unit m/s)

unit string (10.0 [m/s])              → m/s
```

4.12.4.4 Input/Output Format

When converted to string representation (see 4.12.4.3), some lexical elements take a form different from their literal form discribed above:

- Characters are shown without enclosing single quotes.
- Strings are shown without enclosing quotation marks, and quotation marks within the string are not doubled.
- Byte strings are shown without enclosing quotation marks.
- Statecode values are shown without the leading dollar sign. The *no statecode* value remains $$.
- The low level types BYTE, WORD and LONG_WORD are shown as byte strings with the corresponding length.
- Measurement units are not enclosed in square brackets.

The table below gives examples for these different forms:

Type	UCL literal	I/O format
CHARACTER	'a' '''' ' '	a '
string	"hello" "string with ""quotes""" ""	hello string with "quotes"
byte string	#"01FE 0203 AFFE" #""	01 FE 02 03 AF FE
statecode	$OFF $CLOSED $$	OFF CLOSED $$
BYTE WORD LONG_WORD	(no literal)	(like byte strings)
unitized value	1.0 [m/s]	1.0 m/s

4.13 Statements

The following kinds of statements are supported in UCL:

- Assignments
- Procedure calls
- Conditional statements (**`if`**, **`case`**)
- Iteration statements (**`while`**, **`repeat`**, **`loop`**, **`for`**)
- Others (**`exit`**, **`return`**, **`halt`**)

UCL requires each statement to be terminated by a semicolon. In many places a, possibly empty, sequence of statements is allowed. Individual statement descriptions follow.

Formal syntax

```
Statement          = Assignment      | Procedure_Call   | If_Statement     |
                     Case_Statement  | While_Statement  | Repeat_Statement |
                     Loop_Statement  | For_Statement    | Halt_Statement   |
                     Exit_Statement  | Return_Statement

Statement_Sequence = { Statement }
```

4.13.1 Assignment

An assignment serves to replace the current value of a variable (or an array or record element) by a new value indicated by an expression. The assignment operator is "**`:=`**". Variable and expression must be *compatible* (see Section 4.8.6). The designator to the left of the assignment operator must be a variable or an MDB item or subitem given by its pathname; in the latter case, the assignment changes the runtime value of the item or subitem, the corresponding MDB object must therefore be accessible in the READ/WRITE mode. After an assignment is executed, the variable has the value obtained by evaluating the expression (the old value is lost, i.e. overwritten).

The value to be assigned must not violate any constraints imposed on the target variable, e.g. if the variable is of a subrange type, the value must be within the bounds of the subrange. Constraints are usually performed at runtime, and constraint violations yield a runtime error. If the value is constant (und thus known to the compiler), the check is performed at compilation time. Checks on pathname constraints are always performed at compilation time, since a runtime check would require a database access.

Note: In string assignments, left and right side may be of different lengths. If the length of the right side exceeds the maximum length on the left, a runtime error occurs.

Formal Syntax

```
Assignment = Designator ":=" Expression ";"
Designator = see 4.12
Expression = see 4.12
```

Examples

```
type Matrix = array (1 .. 10, 1 .. 20) of Real;

variable S : string (8);
variable C : CHARACTER;
variable A : Matrix;
variable B : Real;
...
S             := ”hello”;
C             := ’X’;
A(I+1, I-1)   := B;
Rec_X.Comp_A  := 100;
\PM\TTC\STAT  := $NOGO;
```

4.13.2 Procedure Call

A procedure call activates the specified procedure. It is denoted by stating the procedure's name, possibly followed by one or more parameters in parentheses. These parameters, the *actual* parameters, are substituted for the *formal* parameters specified in the corresponding procedure declaration.

The association between *actual* and *formal* parameter may be specified either in *positional* or in *named* notation. In the positional form, the correspondence between actual and formal parameters is established by the positions of the respective parameters in the list. In a named parameter association, the parameter must be named explicitly. Named parameters may be given in any order. If both positional and named associations are used in one procedure call, the positional associations must occur first.

For each formal parameter, there must be exactly one actual parameter, (either explicitly specified or by default). If a default value exists, the actual parameter may be omitted. In this case, all remaining actual parameters must be denoted by name. For example, if the procedure P is defined as:

```
procedure P (X : INTEGER := 0; Y : REAL := 0.0; Z : BOOLEAN := FALSE);
```

then the following procedure calls are all valid:

```
P (5, 3.14, TRUE);      -- all parameters positional
P (1, Z : TRUE);        -- X positional, Y omitted, Z named
P (X : 1, Y : 0.0);     -- X and Y named, Z omitted
P ();                   -- all parameters omitted
P;                      -- same as P ();
```

Note that the use of commas to indicate omitted (default) parameters is not allowed. Thus,

```
P (5, , TRUE)
```

is invalid.

In a procedure call, the types of the actual parameters must be *compatible* with those of the formal parameters.

Subprogram parameters belong to one of three classes :

in parameters allow to pass values into a subprogram. Within the subprogram only read access is allowed. The actual parameter for a formal IN parameter may be an expression. The keyword **in** may be omitted in the parameter declaration.

out parameters allow to pass values out of a subprogram. Within the subprogram read access is allowed, but only to obtain a value that was previously assigned within the same subprogram call. The actual parameter for a formal **out** parameter must be a variable or an MDB item or subitem with READ/WRITE access.

`in out` parameters allow to pass values into and out of a subprogram. Both read and write access are allowed within the subprogram. The actual parameter for a formal **`in out`** parameter must be a variable or an MDB item or subitem with READ/WRITE access.

Scalar **`in`** parameter are passed by copying the value into the subprogram, structured **`in`** parameters are passed by reference. **`out`** and **`in out`** parameters are always passed by reference. MDB items and subitems are passed by copy: the input value (**`in`**, **`in out`**) is copied into the procedure, the output value (**`in out`**, **`out`**) is copied back to the location of the runtime value of the item.

If a formal parameter type denotes an array structure, its corresponding actual parameter must be an array of identical type (i.e. same bounds, same component types); unless the array is defined as an *unbound* array or unbound string, in which case only the component types must match (also see unbound array and unbound string in subprogram declarations, 4.6).

A procedure call is inherently synchronous, i.e. control is returned to the caller only after execution of the called procedure is completed.

Note:

- A procedure (or function) may invoke itself (recursive call). Indirect recursion – which occurs, for example, when a procedure A calls a procedure B which, in turn, calls A – is also allowed.
- The order of evaluation of actual parameters is not defined and a program must not depend on the evaluation order.

Formal Syntax

Procedure_Call	=	Qualified_Identifier [Actual_Parameters] ";"
Actual_Parameters	=	"(" [Parameter { "," Parameter }] ")"
Parameter	=	[Identifier ":"] Expression
Qualified_Identifier	=	*see 4.3.4*
Identifier	=	*see 4.1.2.2*
Expression	=	*see 4.12*

Examples

Assuming the following procedure declarations:

```
procedure Calculate_New_Limit  (Threshold    : REAL;
                                Ref_Factor   : REAL;
                                out New_Limit: REAL);

procedure Get_Total  (First    : REAL    := 0.0;
                      How_many : INTEGER := 50;
                      out Total: REAL);

procedure Finish_Up;
```

then these are valid procedure calls:

```
Calculate_New_Limit (Rval1, Rval2, Rlim);
Calculate_New_Limit (100.0, 2.5, Rlim);

Get_Total (5.5, Total: T);
Get_Total (First: 5.5, How_Many: 100, Total: T);
Get_Total (Total: T);

Finish_Up;
```

4.13.3 `if` Statement

The **if** statement denotes conditional execution of one or more statements. It takes the form:

```
if condition_1 then
   statement_sequence_1
elsif condition_2  then
   statement_sequence_2
elsif ...
   ...
else
   statement_sequence_n+1
end if ;
```

where *condition_1* and *condition_2* are Boolean expressions.

These expressions are evaluated one after the other, and as soon as one yields the value TRUE, the corresponding statement sequence is executed, and no further conditions are tested. The **else** part, if present, is executed, if none of the conditions was true. The **elsif** and **else** parts are optional.

Formal Syntax

If_Statement = ”if” Expression ”then”
Statement_Sequence
{ ”elsif” Expression ”then”
Statement_Sequence }
[”else”
Statement_Sequence]
”end” ”if” ”;”

Statement_Sequence = *see 4.13*

Expression = *see 4.12*

Examples

```
if \PM\TTC\TRANSMITTER\STATUS = 1 then ... end if;
```

```
if X < Y then
  statements ...;
elsif X > Y then
  statements ...;
else
  ...;
end if;
```

4.13.4 `case` Statement

The **case** statement is used when a choice from among several mutually exclusive alternatives must be made on the basis of the value of an expression. It thus represents a generalization of the **if** statement. It has the form

```
case expression
   when labels_1: statement_sequence_1
   when labels_2: statement_sequence_2
   ...
   else           statement_sequence_n
end case;
```

The **case** statement causes one of its statement sequences to be selected according to the value of an expression (the *case selector*) which must be of a discrete type (INTEGER, UNSIGNED_INTEGER,

CHARACTER, enumeration type or constrained statecode type). The different values that may be taken by this selector appear in the **case** statement as case labels or case label ranges.

The branches in a **case** statement must be uniquely identified by its labels, i.e. a label may appear only once in a **case** statement. The union of labels and label ranges in a **case** statements must cover all possible values of the selector type, either explicitly or implicitly by an **else** branch that collects all label values not explicitly specified in previous branches. If there is an **else** branch, it must be the last branch in the **case** statement.

The case labels must be constants or constant expressions; and they must be unique within a particular **case** statement (i.e. no label may occur more than once). When the same action is required for several different values of the case selector, these values may be written in a list, separated by commas. Ranges of contiguous values may be written in range notation (lower .. upper). Statecode labels cannot form ranges, because statecode types are unordered. The type of all case labels must be compatible with the type of the case selector.

When the **case** statement is executed, the selector is first evaluated. If there is a case label with the same value as the case selector, the corresponding statement sequence is executed. Otherwise the statement sequence specified in the **else** clause, if present, is executed. The **else** clause is optional; if omitted, however, the specified case labels must cover the entire value range of the case selector type.

Formal Syntax

Case_Statement	=	”case” Expression Case { Case } [”else” Statement_Sequence] ”end” ”case” ”;”
Case	=	”when” Case_Label_List ”:” Statement_Sequence
Case_Label_List	=	Case_Labels { ”,” Case_Labels } | Statecode_List
Case_Labels	=	Constant_Expression [”..” Constant_Expression]
Statecode_List	=	Statecode { ”,” Statecode }
Statecode	=	*see 4.8.1.8*
Constant_Expression	=	*see 4.12*
Expression	=	*see 4.12*
Statement_Sequence	=	*see 4.13*

Examples

```
case Traffic_Light
  when Red   : Stop_Car;
  when Green : Go_Ahead; Turn_Right;
  else         No_Traffic_Light;
end case;
```

```
case Code
  when $LOW    : ... ;
  when $MEDIUM : ...;
  when $HIGH   : ...;
end case;
```

4.13.5 Loop Statements

A loop statement specifies that a sequence of statements is to be repeatedly executed until a termination condition, if any, becomes true, or until an **exit** statement is executed. UCL has four different forms of loop statements.

The sequence of statements within a loop statement may itself contain loop statements; that is, loops may be nested. When a termination condition has become true, or when a loop is left via an **exit** statement, control is passed to the first statement following the innermost enclosing loop (after the **end loop** keywords), and the loop statement is considered to have finished its execution. Note that loops need not necessarily terminate, endless loops are valid constructs.

A loop is, of course, implicitly terminated when a **return** or **halt** statement is executed, see 4.13.6 and 4.13.7.

4.13.5.1 Simple `loop` Statement

The basic form of iteration is the **loop** statement. It specifies that a statement sequence is to be repeatedly executed until an **exit** statement is executed.

A typical **loop** statement with an **exit** statement is shown below.

```
loop
   statement_sequence_1
   if condition then exit; end if;
   statement_sequence_1
end loop;
```

Formal Syntax

Loop_Statement = ”loop”
Statement_Sequence
”end” ”loop” ”;”

Statement_Sequence = *see 4.13*

Example

```
loop
    ...
   A := A + 1;
   if A > 10 then exit; end if;
    ...
end loop;
```

4.13.5.2 `while` Statement

The **while** statement specifies the repeated execution of a statement sequence depending on the value of a Boolean expression. The expression is evaluated before each subsequent execution of the statement sequence, hence the statement sequence may not be executed at all. The repetition stops as soon as the evaluation of the expression yields the value FALSE.

A **while** statement has the following form and is equivalent to the shown **loop** statement:

```
while expression do
   statement_sequence
end while;
```

```
loop
   if ~ expression then exit; end if;
   statement_sequence
end loop;
```

Formal Syntax

While_Statement = ”while” Expression ”do”
Statement_Sequence
”end” ”while” ”;”

Expression = *see 4.12*

Statement_Sequence = *see 4.13*

Examples

```
while \PM\TTC\MODE = 0 do
  Wait (1.0 [s]);
end while;
```

4.13.5.3 `repeat` Statement

A **`repeat`** statement specifies the repeated execution of a statement sequence depending on the value of a Boolean expression. In contrast to the **`while`** loop, the expression is evaluated after each subsequent execution of the statement sequence; hence the statement sequence is executed at least once. The repetition stops as soon as the evaluation of the expression yields the value TRUE.

A **`repeat`** statement has the following form and is equivalent to the shown **`loop`** statement:

```
repeat                                  loop
   statement_sequence                      statement_sequence
until expression;                          if expression then exit; end if;
                                        end loop;
```

Formal Syntax

Repeat_Statement = ”repeat”
Statement_Sequence
”until” Expression ”;”

Statement_Sequence = *see 4.13*

Expression = *see 4.12*

Examples

```
repeat
  INC (Index);
  if ~ Table(Index).Used then exit; end if;
until Index = Max_Index;
```

4.13.5.4 `for` Statement

The **`for`** statement is used when a statement sequence is to be executed a known number of times while a progression of values is assigned to a control variable (the loop index). It takes two alternative forms:

1. Iterative `for` statement

```
for loop_index := start_value to limit [ by increment ] do
  statement sequence
end for;
```

The iterative **`for`** statement is used to loop over a range of values, whose *start_value* and *limit* are explicitly specified. The iterative **`for`** statement cannot be used for statecode types, since statecode types are not ordered. The type of *limit* must be compatible with the type of *start_value*, and the *increment* must be an `INTEGER` constant or constant expression.

2. Collective `for` statement

```
for loop_index in type_name [ by increment ] do
  statement_sequence
end for;
```

The collective **`for`** statement is used to loop over all values of a discrete type, including constrained statecode types. For non–statecode types the *start_value* and *limit* are implicitly determined by the min. and max. value of the type. Since statecode types are unordered, the order in which the values are processed is undefined and the **`by`** clause cannot be used. The *increment* must be an `INTEGER` constant or constant expression.

Note that *start_value*, *limit* and *increment* are evaluated only once, and in that order, before the loop body. Further, the value of the *loop_index* cannot be altered by the statements inside the **`for`** loop.

The ordinal value of *start_value* and *limit*, for both forms, must always lie within the bounds of type `INTEGER`.

The **`for`** statement causes the statement sequence enclosed in the **`for ... end for`** bracket to be repeatedly executed until the value of the *loop_index* has covered all values of the range (*start_value* to *limit)* or type. After each iteration, the *loop_index* is incremented by the specified *increment* value (which may be negative). If no increment is specified, 1 is assumed. For a negative *increment* the values are processed in descending order. In contrast to other forms of loop, **`for`** loops always terminate.

A **`for`** statement (with a positive increment) is equivalent to the following loop:

```
Loop_Index      := start_value;
Saved_Limit     := limit;
Saved_Increment := increment;       -- (1 by default)

while Loop_Index <= Saved_Limit do
  statement_sequence
  Loop_Index := Loop_Index + Saved_Increment;
end while;
```

With a negative increment, the relational operator in the **`while`** statement would be >=.

The *loop index* need not be declared in the UCL program, it is implicitly (that is, automatically) created; its type is derived from the type of *start_value* and *limit* or from the specified type, resp., and its scope is limited to the body of the loop only. Should an object of the same name exist, it is hidden inside the **`for`** loop by the loop index and thus become inaccessible, as shown in the following example. For a discussion of scoping rules see 4.3.2.

```
variable Index: INTEGER;        -- declaration of a variable named Index
...
begin
   ...                          -- The variable Index is accessible here
   ...
   for Index := 1 to 100 do     -- loop index also called Index
      ...
      ...                       -- only the loop index is visible here
      ...                       -- the variable Index is hidden by the
                                -- loop index
      ...
   end for;
   ...                          -- variable Index is now accessible again
   ...                          -- and it still has its original value.
end;
```

Formal Syntax

For_Statement = Iterative_For_Statement | Collective_For_Statement

Iterative_For_Statement = ”for” Identifier ”:=” Expression ”to” Expression [”by” Const._Expression] ”do” Statement_Sequence ”end” ”for” ”;”

Collective_For_Statement= ”for” Identifier ”in” Qualified_Identifier [”by” Constant_Expression] ”do” Statement_Sequence ”end” ”for” ”;”

Identifier = *see 4.1.2.2*

Qualified_Identifier = *see 4.3.4*

Expression = *see 4.12*

Constant_Expression = *see 4.12*

Statement_Sequence = *see 4.13*

Examples

```
for I := 1 to 1000 do
   Sum := Sum + I;
end for;

for i := 10 to –10 by –2 do
   statement sequence
end for;

for C := Red to Blue do
   statement sequence
end for;

type Level = statecode ($LOW, $MEDIUM, $HIGH)
...
for L in Level do
   Process (L);
end for;
```

4.13.6 `return` Statement

A **`return`** statement consists of the keyword **`return`**, possibly followed by an expression. It indicates the end of a procedure or function execution. There may be several **`return`** statements in one procedure or function, although only one will be executed.

In a *function*, the **`return`** statement is mandatory and it must be followed by an expression which specifies the value returned by the function. In a *procedure*, the **`return`** statement is optional. It is implied by the end of the procedure body. When a **`return`** statement is executed, control returns to the caller of the procedure or function.

In a Derived Value the **`return`** statement terminates execution and returns the computed value. It may be used with or without an expression, see 4.16.4 for details.

Formal Syntax

Return_statement = ”return” [Expression] ”;”

Expression = *see 4.12*

Examples:

```
return I + J;    -- in a function
return;          -- in a procedure
```

4.13.7 `halt` Statement

The **`halt`** statement is used within an AP to terminate the AP itself. It further provides an optional facility for reporting the AP completion status (successful/non–successful) to the runtime system.

A **`halt`** statement takes the form:

```
halt [ completion_status ];
```

The expression is optional. It is of the type `COMPLETION_CODE`. By default, `SUCCESS` is assumed.

At runtime, depending on the AP activation method, the completion code may be sent either:

- to the *calling AP* on request
- to the *supervisory (higher–level) software* responsible for AP activation
- to an interactive user who started the AP via an interactive High Level Command Language (HLCL) command, as e.g. in an EGSE environment, on request.

Note that unlike a **`return`** statement, the **`halt`** statement causes termination of the AP, i.e. of the ”main program”. A call to the **`halt`** statement itself is optional; an AP may terminate without such a statement, in which case **`halt`** `SUCCESS` is assumed by default.

Formal Syntax

Halt_Statement = ”halt” [Expression] ”;”

Expression = *see 4.12*

4.13.8 `exit` Statement

The **`exit`** statement consists of the keyword **`exit`**. It causes termination of the innermost enclosing loop, execution is resumed with the statement following the loop.

Any iterative statement (i.e. **`loop`**, **`while`**, **`repeat`** or **`for`**) may be left via an **`exit`** statement.

Formal Syntax

Exit_Statement = ”exit” ”;”

Examples

```
loop
  A := A + 1;
  if A > 10 then
    exit;
  end if;
  B := B + A;
end loop;
...       <-- execution continues here when A > 10
```

4.14 Subprogram Declarations

A subprogram (or subroutine) is a code segment or sequence of statements which can be invoked by name. UCL supports two kinds of subprograms: *procedures* and *functions*.

4.14.1 Procedure Declaration

A procedure declaration comprises:

- a *procedure header* consisting of the reserved word **`procedure`** followed by the procedure name and an optional parameter list in which the procedure's *formal parameters* are defined.
- a *procedure body* which may contain declarations and statements. It must not, however, contain procedure or function declarations. In other words, procedures and functions cannot be nested inside other procedures and functions.

Formal Parameters

Formal parameters are identifiers which denote values or objects to be passed with the call. There are three *modes* of formal parameters:

- **`in`** parameters are treated as place holders to which the result of the evaluation of the corresponding actual parameter is assigned as an initial value. They may be assigned *default values* (via the assignment operator "`:=`"). These default values are used as actual parameters if the corresponding parameters are omitted from the procedure call.
- **`in out`** and **`out`** parameters correspond to actual parameters that are variables, and they stand for these variables.

Each formal parameter must be specified by its name and its type. In the parameter list, the reserved words **`in`** and **`out`** are used to preceed the parameter in order to specify a parameter mode. If no mode is specified, then **`in`** is implicitly assumed.

Formal parameters are local to the procedure, i.e. their scope is the program text which constitutes the procedure declaration. They can also be used within the actual parameter list to associate a value with a specific parameter, see 4.13.2. In this context they do not conflict with identifiers declared outside the subprogram.

The type of a formal parameter may be an *open type*:

- If the parameter is a string, the type may be specified as just

  ```
  string
  string of type_identifier
  ```

  where the specification of the maximum length is omitted. The parameter is then said to be an *open* or *unbound string* parameter, and actual parameters of any string types are accepted. Their first index and maximum length can be obtained through the standard functions `LOW` and `HIGH`.

- If the parameter is an array, the type may be specified as just

  ```
  array of type_identifier
  ```

  where the specification of the actual index bounds is omitted, e.g. **`array of`** `INTEGER`. The parameter is then said to be an *open* or *unbound array* parameter, and array parameters of any size, but with the given element type, are accepted. Within the subprogram, the actual index range and its type is always mapped to a corresponding `INTEGER` range with a lower bound of 0; its lower and upper bounds can be obtained through the standard functions `LOW` and `HIGH`.

- The parameter type may be one of the open forms

  `pathname` any pathname types

  `pathname.*` any subitem pathname types

  `statecode` any statecode types

If the parameter is of a pathname type, it may require actual parameters to be given together with their actual parameter list. For parameterized MDB items see 4.3.6. The pathname type specification is then suffixed with an empty pair of parentheses:

`pathname``()`

`typename``()`

Subitem pathnames cannot be parameterized.

Procedure body

The procedure body contains a statement sequence bracketed by the reserved words **`begin`** and **`end`** possibly preceded by local constant, type, variable or alias declarations. These local declarations effectively hide any identically named objects in the enclosing main program (AP), as well as imported and predefined identifiers. A procedure or function declaration may appear within an AP or a library, but not within another procedure or function. After the end keyword the name of the procedure may be repeated.

Also note that parameters of mode **`in`** may not be modified within a procedure body; i.e. they are not allowed as targets in an assignment statement and must not be passed as an **`out`** or **`in out`** parameter to another subprogram.

Formal Syntax

Procedure_Declaration = Procedure_Heading Block

Procedure_Heading = ”PROCEDURE” Identifier [Formal_Parameters] ”;”

Block = { Declaration }
”begin”
Statement_Sequence
”end” [Identifier] ”;”

Formal_Parameters = ”(” [Parameter_List { ”;” Parameter_List }] ”)”

Parameter_List = [”in”] [”out”] Identifier_List ”:” Formal_Type [”:=” Constant_Expression]

Identifier_List = Identifier { ”,” Identifier }

Formal_Type = Formal_Simple_Type |
”array” ”of” Formal_Simple_Type |
”string” [”of” Identifier]

Formal_Simple_Type = Qualified_Identifier [”(” ”)”] |
”statecode” |
”pathname” [”(” ”)”] |
”pathname” ”.” ”*”

Declaration = *see 4.3*

Statement_Sequence = *see 4.13*

Identifier = *see 4.1.2.2*

Qualified_Identifier = *see 4.3.4*

Constant_Expression = *see 4.12*

Example

```
procedure Accumulate (First: INTEGER; Last: INTEGER; out Result: INTEGER);
   variable Sum: INTEGER;
begin
   Sum := First;
   for I := First + 1 to Last do
      Sum := Sum + I;
   end for;
   Result := Sum;
end Accumulate;
```

4.14.2 Function Declaration

A function is a subprogram that returns a value. A function declaration starts with the reserved word **function**, followed by the function name, an optional list of formal parameters, and the function result type. Formal parameters and body of functions are specified in the same way as procedures (see 4.14.1).

The type of the function return value is restricted to scalar types and pathname types.

The statements of the function body must include one or more **return** statements (see 4.13.6) specifying the returned value (but only one **return** statement is executed). A runtime error occurs if a function is left otherwise than by a **return** statement. Note that the expression following the **return** keyword must yield a value of the function result type stated in the function header.

Neither **out** nor **in out** parameters are allowed for functions.

Formal Syntax

Function_Declaration	=	Function_Heading Block
Function_Heading	=	”function” Identifier [Formal_Parameters] ”:” Qualified_Identifier [Unit] ”;”
Block	=	{ Declaration } ”begin” Statement_Sequence ”end” [Identifier] ”;”
Formal_Parameters	=	*see 4.14.1*
Declaration	=	*see 4.3*
Statement_Sequence	=	*see 4.13*
Qualified_Identifier	=	*see 4.3.4*
Identifier	=	*see 4.1.2.2*
Unit	=	*see 4.11 and 4.1.2.10*

Examples

```
function Square (N : INTEGER) : INTEGER;
begin
   return N*N;
end Square;

function Is_Printable (C : CHARACTER) : BOOLEAN;
begin
   return C > CHARACTER(31) & C < CHARACTER(127);
end Is_Printable;
```

```
function Sum_Of (A : array of INTEGER) : INTEGER;
  variable Sum : INTEGER := 0;
begin
  for I := 0 to HIGH (A) do
    Sum := Sum + A(I);
  end for;
  return Sum;
end Sum_Of;
```

4.14.3 Guarded Procedures, Functions and Parameters

Procedures and functions may be marked as *guarded* in the form

```
guarded procedure Name (...);
guarded function Name (...) : Type;
```

Also the formal parameters of a procedure or function may be marked as *guarded*:

```
procedure Name (guarded Name : Type; ...);
function Name (guarded Name : Type; ...) : Type;
```

A subprogram that calls a guarded subprogram or a subprogram with a guarded parameter will inherit the privileges attached to the accessed entity, and calls to this subprogram will themselves be guarded. If the compilation unit itself in its body or initialization part, resp., calls such a subprogram, the compilation unit will inherit the privileges attached to the guarded items. For a description of privileges and authorization see 4.17.

4.15 Standard Functions and Procedures

The following functions and procedures are predefined. Some of them are ”generic” in the sense that they may have several possible parameter list forms. For example, the procedures `INC/DEC` may have one or two actual parameters and the first parameter may be an `INTEGER`, `UNSIGNED_INTEGER` or an enumerated–type value.

ABS Function

`ABS(`*x*`)` returns the absolute value of the expression or variable *x*.
x must be of type `INTEGER, UNSIGNED_INTEGER, REAL` or `LONG_REAL`. The result has the type of *x*.

MAX Function

`MAX(`*t*`)` denotes a constant equal to the maximum (highest) value of the type *t*, where *t* may be one of the basic types (or subtypes thereof)

`INTEGER`, `UNSIGNED_INTEGER`, `REAL`, `LONG_REAL`, `TIME`, `DURATION`, enumeration types

`MAX(`x_1`, `x_2`, ..., `x_n`)`
determines the maximum of a group of values. The values must be of one of the above mentioned types, and all must be of the same type or implicitly convertible to each other.

`MAX(`*a*`)` determines the maximum element of the array *a*. The array may be of any dimension, and the elements must be of one the above mentioned types.

MIN Function

`MIN(`*t*`)` denotes a constant equal to the minimum (lowest) value of the type *t*, where *t* may be one of the basic types

`INTEGER, UNSIGNED_INTEGER, REAL, LONG_REAL, TIME, DURATION`, enumeration types

`MIN(`x_1`, `x_2`, ..., `x_n`)`
determines the minimum of a group of values. The values must be of one of the above mentioned types, and all must be of the same type or implicitly convertible to each other.

`MIN(`*a*`)` determines the minimum element of the array *a*. The array may be of any dimension, and the elements must be of one the above mentioned types.

LOW Function

`LOW(`*a*`)` returns the lowest index of the array or string type or variable a. For strings it is always 1. The return value has the type of the corresponding index, for open arrays and strings it is always `INTEGER`, 0 for open arrays, 1 for open strings.

`LOW(`*a, n*`)` returns the lowest index of the `n`–th dimension of the (multidimensional) array type or variable `a`. The return value has the type of the corresponding index.

HIGH Function

`HIGH(a)` returns the uppermost index of the array or string type or variable `a`. The return value has the type of the corresponding index, for open arrays and strings it is always `INTEGER`.

`HIGH(a, n)` returns the uppermost index of the `n`–th dimension of the (multidimensional) array type or variable `a`. The return value has the type of the corresponding index.

LENGTH Function

`LENGTH(a)` returns the current (actual) length of string *a*. For an empty string, `LENGTH` returns 0. The return value is of type `INTEGER`.

ODD Function

`ODD(x)` returns the Boolean value `TRUE` if *x* is odd (i.e. `x % 2 <> 0`), else `FALSE`. *x* must be of type `INTEGER` or `UNSIGNED_INTEGER`.

INC Procedure

This procedure increments a variable. The parameter can be of any discrete type, except statecode types.

`INC(x)` This procedure increments the variable *x* by 1. If *x* is an enumeration type or `CHARACTER` value, it is replaced by its successor.

`INC(x,n)` This procedure increments the variable *x* by *n*. Regardless of the type of *x*, *n* must be an expression of type `INTEGER` or `UNSIGNED_INTEGER`. If *x* is an enumeration type or `CHARACTER` value, it is replaced by its *n*–th successor.

DEC Procedure

This procedure decrements a variable. The parameter can be of any discrete type, except statecode types.

`DEC(x)` This procedure decrements the variable *x* by 1.
If *x* is an enumeration type or `CHARACTER` value, it is replaced by its predecessor.

`DEC(x,n)` This procedure decrements the variable *x* by *n*. Regardless of the type of *x*, *n* must be an expression of type `INTEGER` or `UNSIGNED_INTEGER`. If *x* is an enumeration type or `CHARACTER` value, it is replaced by its *n*–th predecessor.

INCL Procedure

`INCL(s,x)` This procedure includes the element *x* into the set variable *s*.
x must be an expression of the element type of *s*.

EXCL Procedure

`EXCL(s,x)` This procedure excludes the element x from the set variable s.
x must be an expression of the element type of *s*.

PUT Procedure

`PUT(S)` This procedure prints the string `S` on the standard output channel.

UNTYPED Function

`UNTYPED(X)` This function supports low level conversion. It removes the type from the expression `X` and makes it convertible to any other type. For details see 4.12.4.2.

4.16 Compilation Units

A text which is accepted by the compiler as a unit is called a *compilation unit*. There are four kinds of compilation units in UCL:

- Main program (called *Automated Procedure* in the COLUMBUS UCL environment)
- Library specification
- Library body
- Formal parameter list definition
- Derived value

Automated Procedures

An Automated Procedure (AP) constitutes a parameterized UCL program or main procedure. It may import objects defined in separately compiled units, called *libraries*, and it can also be imported by other objects, see 4.2.

Libraries

A UCL *library* (like a *package* in Ada, or a *module* in Modula–2) is a collection of computational resources (procedures, functions, data types, variables etc.) that may be used by the library's clients by *importing* the provided objects or services. In the UCL context, these clients may be any kind of compilation unit.

A UCL library consists of two parts: a *library specification* and a *library body* (implementation part). Each part is compiled separately. The library *specification* identifies the resources that are visible to its clients, whereas the *body* contains the corresponding implementation details and is hidden from the clients.

All objects declared in a library specification are available in the corresponding body without explicit import. Also note that all libraries (including system libraries) must be explicitly imported by their clients.

A library specification and its corresponding body form two separate compilation units. The library body may be modified without affecting the library's clients, whereas a modification of the library specification implies recompilation of the corresponding body and of all clients.

UCL requires compilation units to be compiled in the following order:

1. A library specification must be compiled *before* its clients.
2. A library specification must be compiled *before* its body (if one exists).

Note that the body of a user library must have been compiled before its services can actually be used (executed) by a client.

Formal Parameter List Definitions

Database items may have formal parameter lists. When using such an item in a context that requires a parameterized item (see 4.14.1), the item name must be given together with its actual parameter list. The UCL Compiler can be used to create the formal parameter lists of such database items. Items with formal parameter lists can be imported by other objects, see 4.2.

Derived Values

A *derived value* is a database item that returns a value computed from the values of other database items. Its compilation unit describes the algorithm for the value computation.

4.16.1 Automated Procedures

An *Automated Procedure* (AP) corresponds to a UCL *main program*. An AP declaration consists of the following:

- an optional import and declarations section before the AP header
- an AP header
- an optional list of imported libraries or other modules
- an optional declarations section
- a statement sequence bracketed by **`begin`** and **`end`**.

The *AP header* comprises the keyword **`procedure`** followed by the AP's name and an optional *parameter list*. The AP name is implicitly declared as an alias for the AP's pathname in the database. The parameter list is similar to the formal parameter list of a procedure or function (see 4.14.1), except that no **`in out`** or **`out`** parameters are allowed.

The optional *import list* contains import directives of the form:

`import` *`pathname`*;

where *`pathname`* denotes an importable item, e.g. a library from which objects are imported. The **`import`** clause makes all objects declared in the imported module's specification directly visible. See section 4.2 for details.

The optional declaration part of an AP may declare constants, types, variables, aliases and units, as well as procedures and functions. See sections 4.3 and 4.6 for details.

The AP's name may be repeated at the end, after the **`end`** keyword; in this case, it obviously must match the name following the **`procedure`** keyword in the header.

The AP definition may be preceded by unit, constant, type and alias declarations. This allows to have parameter types other than the predefined types, and named default values in the AP parameter list. Units, constants, types and aliases may also be obtained by importing them from libraries or other parameterized items.

Note that items declared outside the AP belong to the same name scope as the global declarations within the AP. When importing an AP in another module, only identifiers declared before the AP header, the implicit AP alias and the parameters become visible, see 4.2.

Formal Syntax

```
Main_Procedure      =  { Import }
                       { Unit_Declaration | Constant_Declaration |
                         Type_Declaration | Alias_Declaration }
                       procedure" Identifier [ Formal_Parameters ] ";"
                           { Import }
                           { Declaration }
                       "begin"
                           Statement_Sequence
                       "end" [ Identifier ] ";"
Formal_Parameters   =  see 4.14.1
Import              =  see 4.2
Declaration         =  see 4.3
```

Unit_Declaration = *see 4.11.4*

Constant_Declaration = *see 4.7*

Type_Declaration = *see 4.8*

Alias_Declaration = *see 4.10*

Statement_Sequence = *see 4.13*

Identifier = *see 4.1.2.2*

Example

```
-- This is the definition of an Automated Procedure named
-- \PM\PAYLOAD\EQUIPMENT_UNIT_A\COMMAND\MODE_OFF.
-- Procedures like Issue, Execute_AP, Enable_EVL etc. are imported from a
-- system library named \APM\UCL\GROUND_LIBRARY.

procedure Mode_Off;

  import \APM\UCL\GROUND_LIBRARY;

  constant Limit   : REAL := 15.0;

  variable AP_Id   : AP_Identifier;
  variable AP_Code : INTEGER;
  variable Status  : INTEGER;

  alias Equipment = \APM\PAYLOAD\EQUIPMENT_UNIT_A;

begin
  if Equipment\NOSHARE\FLAG = $ON then
    Write_Message_To_User ("Equipment error", Status);
    halt FAILURE;

  elsif Equipment\MODE = $Off then
    if Equipment\SENSOR_1 > Limit then      -- check sensor output
      Write_Message_To_User ("Limit exceeded", Status);
      halt FAILURE;

    else
      Issue (Equipment\POWER_OFF, Status: Status);
    end if;
  end if;

  Delay (10.0 [s]);

  if Equipment\SENSOR_1 <= Limit then
    Execute_AP (Equipment\COMMAND\MODE_ON, AP_Id, Status);
    Synchronize_With_AP (AP_Id, \\, AP_Code, Status);
  end if;

  Equipment\Mode := $OFF;                -- update the status flag
  Enable_EVL (Equipment\MODE, Status);   -- write flag to memory
end Mode_Off;
```

Example

```
-- This AP has an import and declaration section before its header.
-- It uses identifiers from both the imported library and the declaration
-- in its parameter list.

import \APM\UCL\ONBOARD_LIBRARY;

type Switch = statecode ($OFF, $ON);

procedure Check (Measurement : Discrete_Value_Item;  -- imported type
                 State       : Switch);              -- type declared above

   variable Value : statecode;

begin
   Get_Discrete (Measurement, Value, ...);

   ...

   if Value <> State then
      Submit_Event (...);
      halt Failure;
   end if;
end Check;
```

4.16.2 Libraries

In UCL, a *library* is an encapsulation mechanism or envelope for data structures and operations (similar to a *package* in Ada, or a *module* in Modula–2). There are two types of UCL libraries:

- *user libraries* which have both their specification and implementation (body) written in UCL,
- *system libraries* which are special, predefined libraries with a UCL specification, but no corresponding UCL implementation. Such libraries will be provided by the target–specific runtime environment (presumably as part of the interpreter).

From the user point of view the library type is transparent. Both types of libraries are imported and used in exactly the same way.

A library consists of two parts belonging to two separate compilation units:

- library specification
- library body (missing for system libraries).

The *library specification* is the list of all entities which are visible to APs or other libraries (i.e., they may be **`import`**ed). The *library body*, on the other hand, contains the actual implementations of these entities, as well as other entities that are invisible from the outside. For system libraries the body is missing, since their implementation is part of the runtime system.

This partitioning of libraries (into specification and implementation part) allows:

- to restrict the number of entities visible to users, and
- to hide implementation details (information hiding principle)

It also improves maintainability by allowing changes to be made to a library implementation without affecting its clients.

UCL allows a library to be composed of a specification part only, i.e. without a corresponding body. For example, a library could contain constant and type declarations only.

Note that the specification and body parts logically belong to the same library and must share the same name. The specification part of a library represents the interface between the library on one side and the library's clients on the other side. Any change to the specification will require changes to the body. On the other hand, the body may change without affecting the definition part and, hence, its clients.

Library specification

A library specification starts with the keyword **`library`**, followed by the library's name, an optional **`import`** list (see 4.2), and an optional declaration part (see 4.3). It is terminated by the keyword **`end`**. The library's name may be repeated at the end, after the **`end`** keyword, in which case it obviously must match the name specified in the header. The library name is implicitly declared as an alias for the library's pathname in the database. This alias is visible to client units.

Only the entities declared in a library specification can be imported by other compilation units. The declaration part may contain any kind of declarations, but *system libraries* cannot declare variables. For procedures and functions only the subprogram header is given, the subprogram body must be given in the library body.

Subprograms declared in system libraries are identified in the system by a unique number. This number is assigned implicitly, starting at 1, but may also be given explicitly by prefixing the subprogram header with the number followed by a semicolon. Implicit and explicit numbering may be mixed, but as soon as explicit numbering is used, all following subprograms must be explicitly numbered. The numbers must be unique but need not be contiguous or in any order.

Library body

A library body starts with the keywords **`library body`**, followed by the library's name, an optional **`import`** list (see 4.2), an optional declaration part (see 4.3), and finally by the actual body containing the respective implementations for all functions and procedures defined in the corresponding library specification. Note that identifiers declared in the specification part of a library are visible in the corresponding body, i.e. may be used in the body without declarations. Items imported into a library specification are available to the library body, as well.

A library body may optionally contain an *initialization* section (consisting of a statement sequence delimited by **`begin`** and **`end`**), which is used to initialize its local objects. This initialization section will be automatically executed before the client (AP or library) is executed. A library is initialized only once, even if it is imported many times by different imported units. If multiple libraries are imported they are initialized in the order in which they textually appear. Circular references (for example, A imports B, which imports C, which imports A) are not allowed and result in error messages.

At runtime, all APs share the same system libraries, but user libraries are not shared between APs, each AP has its own copy of user libraries.

Formal Syntax

```
Library_Specification = "library" Identifier ";"
                          { Import }
                          { Definition }
                        "end" [ Identifier ] ";"

Library_Body          = "library" "body" Identifier ";"
                          { Import }
                          { Declaration }
                        [ "begin"
                          Statement_Sequence ]
                        "end" [ Identifier ] ";"

Definition            = Unit_Declaration |
                        Constant_Declaration |
                        Type_Declaration |
                        Variable_Declaration |
                        Alias_Declaration |
                        [ Simple_Integer ":" ] Procedure_Heading |
                        [ Simple_Integer ":" ] Function_Heading

Import                = see 4.2
Declaration           = see 4.3
Statement_Sequence    = see 4.13
Unit_Declaration      = see 4.11
Constant_Declaration  = see 4.7
Type_Declaration      = see 4.8
Variable_Declaration  = see 4.9
Alias_Declaration     = see 4.10
Procedure_Heading     = see 4.14.1
Function_Heading      = see 4.14.2
Identifier            = see 4.1.2.2
Simple_Integer        = see 4.1.2.4.1
```

Example 1

This library keeps a buffer of pathnames. The buffer implementation is invisible to the user, the library provides operations to put/get items in/from the buffer, and to inquire the number of items in the buffer.

Library specification:

```
library Pathname_Buffer;

   constant Buffer_Size : INTEGER := 100;

   type Buffer_Error = (No_Error, Buffer_Full, Buffer_Empty);

   procedure Put_Buffer (in  Item  : pathname;
                         out Error : Buffer_Error);

   procedure Get_Buffer (out Item  : pathname;
                         out Error : Buffer_Error);

   function Buffer_Level : UNSIGNED_INTEGER;

end Pathname_Buffer;
```

Library body:

```
library body Pathname_Buffer;

   type Buffer_Type = array (1 .. Buffer_Size) of pathname;

   variable Buffer : Buffer_Type;
   variable Index  : UNSIGNED_INTEGER := 0;

   procedure Put_Buffer (in  Item  : pathname;
                         out Error : Buffer_Error);
   begin
      if Index < Buffer_Size then
        INC (Index);
        Buffer (Index) := Item;
        Error := No_Error;
      else
        Error := Buffer_Full;
      end if;
   end Put_Buffer;

   procedure Get_Buffer (out Item  : pathname;
                         out Error : Buffer_Error);
   begin
      if Index > 0 then
        Item := Buffer (Index);
        DEC (Index);
        Error := No_Error;
      else
        Error := Buffer_Empty;
      end if;
   end Get_Buffer;

   function Buffer_Level : UNSIGNED_INTEGER;
   begin
      return Index;
   end Buffer_Level;

end Pathname_Buffer;
```

Example 2

This user library provides operations to split TIME values into their components and to construct TIME values from components. This is an example of how to use low level conversions with overlayed record variants in a safe and controlled way (see 4.12.4.2). For the TIME representation in memory see 4.8.1.9.

Library specification:

```
library Time_Library;
  procedure Split_Time (in  Item    : TIME;
                        out Year    : UNSIGNED_INTEGER;
                        out Month   : UNSIGNED_INTEGER;
                        out Day     : UNSIGNED_INTEGER;
                        out Seconds : DURATION);
  function Time_Of (Year    : UNSIGNED_INTEGER;
                    Month   : UNSIGNED_INTEGER;
                    Day     : UNSIGNED_INTEGER;
                    Seconds : DURATION) : TIME;
end Time_Library;
```

Library body:

```
library body Time_Library;
   -- Use a record with variants to overlay time values, packed components:
   -- year – 1900 (8 bits), month (4 bits), day (5 bits),
   -- seconds since midnight (fix point, 47 bits, 17 bits fore, 30 bits aft).

  type Time_Overlay = record
                        case Tag: BOOLEAN
                           when False: Word_1     : UNSIGNED_INTEGER;
                                       Word_2     : UNSIGNED_INTEGER;
                           when True:  Time_Field : Time;
                         end case;
                       end record;

  procedure Split_Time (in  Item    : TIME;
                        out Year    : UNSIGNED_INTEGER;
                        out Month   : UNSIGNED_INTEGER;
                        out Day     : UNSIGNED_INTEGER;
                        out Seconds : DURATION);
    variable Overlay: Time_Overlay;
  begin
    -- Handling of ~:~ and dateless times omitted for simplicity
    Overlay.Time_Field := Item;
    Year  := Overlay.Word_1 / 2 ** 24 + 1900;
    Month := UNSIGNED_INTEGER (BITSET (Overlay.Word_1) * {20 .. 23}) / 2**20;
    Day   := UNSIGNED_INTEGER (BITSET (Overlay.Word_1) * {15 .. 19}) / 2**15;

    Seconds :=
          DURATION (LONG_REAL (Overlay.Word_1 % 2**15 * 2**2)) +
          DURATION (LONG_REAL (Overlay.Word_2 / 2**30)) +
          DURATION (LONG_REAL (Overlay.Word_2 % 2**30)) / LONG_REAL (2**30);
  end Split_Time;
```

```
   function Time_Of (Year    : UNSIGNED_INTEGER;
                     Month   : UNSIGNED_INTEGER;
                     Day     : UNSIGNED_INTEGER;
                     Seconds : DURATION) : TIME;

     variable Overlay: Time_Overlay;

   begin
     -- Check for validity and handling of ~:~ and dateless times
     -- omitted for simplicity

    Int      := UNSIGNED_INTEGER (LONG_REAL (Seconds));  -- truncate
    Fraction := Seconds - DURATION (LONG_REAL (Int));

    Overlay.Word_1
             := (Year - 1900) * 2**24 +
                 Month * 2**20 +
                 Day * 2**15 +
                 Int / 2**2;

    Overlay.Word_2
             := Int % 2 ** 2 * 2 ** 30 +
                UNSIGNED_INTEGER (LONG_REAL (Fraction) * LONG_REAL (2**30));

      return Overlay.Time_Field;
   end Time_Of;

end Time_Library;
```

Example 3

The following library does not need a body. It contains no executable parts, but only passive declarations. It might be implemented both as a user library or as a system library.

Library specification:

```
-- This library defines a set of useful non-SI engineering units.

library Non_SI_Units;
   unit [in]   = [0.0254 m];           <- inch
   unit [ft]   = [12 in];              <- foot
   unit [yd]   = [3 ft];               <- yard
   unit [mi]   = [1760 yd];            <- mile
   unit [oz]   = [0.282495];           <- ounce
   unit [lb]   = [16 oz];              <- pound
   unit [gr]   = [lb/7000];            <- grain
   unit [cwt]  = [112 lb];             <- hundredweight
   unit [ton]  = [20 cwt];             <- ton
   unit [degC] = [273.15 Kabs];        <- degree Celsius
   unit [degF] = [10 degC/18 - 320/18]; <- degree Fahrenheit
end Non_SI_Units;
```

Example 4

The following system library declares explicitly numbered subprograms. The numbers are non–contiguous, in order to group related subprograms.

The subprograms are annotated.

Library specification:

```
library IO_Library;
   <- Simple input/output operations.
   <- The subprograms operate on standard input and standard output.

-- Input operations:

   1: procedure Get_Char (out C: CHARACTER);
        <- Read character from standard input

   2: procedure Get_String (out S: string);
        <- Read string from standard input

   3: procedure Get_Integer (out I: INTEGER);
        <- Read an Integer literal from standard input

   4: procedure Get_Real (out R: REAL);
        <- Read a Real literal from standard input

-- Output operations

  11: procedure Put_Char (C: CHARACTER);
        <- Write a character on standard ouput

  12: procedure Put_String (S: string;    Width: INTEGER := 0);
        <- Write a character on standard ouput

  13: procedure Put_Integer (I: INTEGER;   Width: INTEGER := 1);
        <- Write a character on standard ouput

  14: procedure Put_Real (R: REAL;      Width: INTEGER := 0);
        <- Write a character on standard ouput

-- Line and column control

  21: procedure New_Line;
        <- Terminate current output line

  30: function Column : Unsigned_Integer;
        <- Return the current output column

  31: procedure Set_Column (Column : Unsigned_Integer);
        <- Set a new output column

end IO_Library;
```

4.16.3 Formal Parameter List Definitions

Items in the database may have formal parameter lists. When such an item is used in a context that requires parameters, the item name must be given together with actual parameters conforming to the formal parameter list.

The formal parameter list can be defined in UCL syntax and compiled with the UCL compiler. The parameter list definition is given in the same form as for an AP or a subprogram:

```
identifier (formal_parameter, ..., formal_parameter);
```

The *identifier* is optional. If present, it will be implicitly declared as an alias for the item's path name. For specific item types there may be restrictions on the allowed number and types of parameters. Such operational restrictions must be observed by the programmer and will not be checked by the compiler.

The parameter list definition may be preceded by unit, constant, type and alias declarations. This allows to have parameter types other than the predefined types, and named default values. Units, constants, types and aliases may also be obtained by importing them from libraries or other parameterized items.

Database items with formal parameter lists may be imported like libraries in any compilation unit. An import makes all objects declared in the formal parameter list definition visible in the importing compilation unit. This comprises explicitly declared units, constants, types and aliases, as well as the implicitly declared alias for the item's path name. Without an import, a parameterized item can be used without any restrictions, but any additional identifiers declared together with the formal parameter list will not be visible.

Formal Syntax

Formal_Parameter_List_Definition =
{ Import }
{ Unit_Declaration | Constant_Declaration |
Type_Declaration | Alias_Declaration }
[Identifier] Formal_Parameters ";"

Import = *see 4.2*

Unit_Declaration = *see 4.11*

Constant_Declaration = *see 4.7*

Type_Declaration = *see 4.8*

Alias_Declaration = *see 4.10*

Formal_Parameters = *see 4.14.1*

Identifier = *see 4.1.2.2*

Example

```
import \APM\GLOBAL\COMMON_TYPES;          -- import common type declarations

constant Lower : INTEGER := 0;
constant Upper : INTEGER := 255;

type Range = INTEGER (Lower .. Upper);

Message (Name : string;                   -- the parameter list definition
         Value : Range := Lower);
```

4.16.4 Derived Values

A *derived value* is a database item that returns a value computed from the values of other database items. Its UCL source text describes the algorithm for the value computation.

A derived value definition consists of the following:

- an optional list of imported libraries (system libraries only) or other modules
- an optional declaration section with constant, unit, type, variable and alias declarations.
- a statement sequence containing at least a **`return`** statement and, optionally, **`if`** and **`case`** statements. Other statements are not allowed.

The main part of the derived value definition is a **`return`** statement that returns the value computed via an expression. It can be used in two forms:

`return` *`expression`*`;`

This statement computes a value, returns it as the new value of the item and stops execution. The type of the returned value must be compatible to the software type of the item.

`return;`

When the expression is omitted, execution is stopped without computing a new value. The item keeps its old value.

The expression will typically involve the runtime value of other database items, but it can also use constants declared locally, as well as constants and variables imported from libraries, and it may call imported functions.

A derived value may access itself, in which case its old value will be used. Indirect cyclic references through other items are not allowed.

Termination of execution must always be explicitly done through a **`return`** statement, otherwise a runtime error will occur. There may be more than one **`return`** statement.

Derived value items may be used in other compilation units via their pathname, in order to obtain their current value, like any other database item that has a value.

Formal Syntax

Derived_Value	=	{ Import }
		{ Unit_Declaration | Constant_Declaration | Type_Declaration | Variable_Declaration | Alias_Declaration }
		{ If_Statement | Case_Statement | Return_Statement }
Import	=	*see 4.2*
Unit_Declaration	=	*see 4.11*
Constant_Declaration	=	*see 4.4*
Type_Declaration	=	*see 4.8*
Alias_Declaration	=	*see 4.10*
If_Statement	=	*see 4.13.3*
Case_Statement	=	*see 4.13.4*
Return_Statement	=	*see 4.13.6*

Example

```
return (\EGSE\HK_TM\HK_PFE\DC_POWER_SUPPLY\XTP0104 +
        \EGSE\HK_TM\HK_PFE\DC_POWER_SUPPLY\XTP0804);
```

Example

```
alias FNY1016 = \PLM\INSTR\SARR_TM_FLIGHT\RTRF\FNY1016;
alias FNY1017 = \PLM\INSTR\SARR_TM_FLIGHT\RTRF\FNY1017;
alias FNY1019 = \PLM\INSTR\SARR_TM_FLIGHT\RTRF\FNY1019;
alias FNY1020 = \PLM\INSTR\SARR_TM_FLIGHT\RTRF\FNY1020;

if FNY1016 > 0.1 [A] & FNY1019 > 10.0 [V] then
  return 2;
elsif FNY1017 > 0.1 [A] & FNY1019 > 10.0 [V] then
  return 3;
elsif FNY1016 > 0.1 [A] & FNY1020 > 10.0 [V] then
  return 4;
elsif FNY1017 > 0.1 [A] & FNY1020 > 10.0 [V] then
  return 5;
else
  return 1;
end if;
```

4.17 Privileges and Authorization

Systems using UCL may assign user privileges to users and to single entities managed by the system, such as commands, measurements, variables, APs, libraries and other items maintained in the Mission Database. Furthermore, individual library procedures and functions can be assigned privileges. A privilege is named with a single identifier that corresponds to UCL identifier syntax. Any number of privileges may be defined, and any subset of privileges may be assigned to users and items. The assignment of privileges is done outside UCL with a system tool.

Required privileges are configured in the database with each single item. UCL compilation units and subprograms that reference other privileged items, recursively inherit the privileges of all items referenced directly or indirectly. UCL does not assume a specific meaning of the privileges, it just keeps track of all privileges inherited by each item.

When using a privileged item in an HLCL command window, e. g. starting an AP, HLCL will deny access to the item, if the user lacks any privileges required by the item. For details see the HLCL Reference Manual (2.2.1).

4.17.1 Determining the Privileges of a Subprogram or Compilation Unit

When compiling a UCL item, the compiler determines the full set of required privileges, i. e. the assigned privileges of the item itself plus all privileges inherited through references to other items, and stores them together with the compiled item in the database. The privileges of items used within a compilation unit are evaluated only if used in a *guarded* position. In non–guarded positions, the privileges are ignored:

- Imports are always guarded.
- Assignments to software variables are always guarded, as well as passing a software variable as an out or in out parameter to a procedure.
- Procedure and function calls are guarded either if the call is to a library procedure or function explicitly marked with the keyword **`guarded`** in its declaration, or if the called subprogram directly or indirectly accesses any guarded entities.
- Passing a pathname as a parameter is guarded only, if the formal parameter was explicitly marked with the keyword **`guarded`** in the procedure/function declaration.

A compilation unit accessing any entities that are guarded in the above described way will itself inherit the privilegs of the accessed entities.

The privileges themselves do not appear in the UCL source text, they are exclusively configured in the database. The source text just marks the guarded positions. Therefore, no UCL items need to be rewritten and recompiled, if privileges change.

4.17.2 Guarded Library Procedures and Functions

Library procedures and functions can be marked as guarded as shown in the example.

```
library Ground_Library;
   ...
   procedure Write_Message_To_User (...);       -- not guarded
   procedure Read_Message_From_User (...);      -- not guarded
   ...
   guarded procedure Enable_Monitoring (...);   -- guarded
   guarded procedure Disable_Monitoring (...);  -- guarded
   ...
   guarded procedure Issue (...);               -- guarded
   ...
   procedure Pathname_To_String (...);          -- not guarded
   ...
end Ground_Library;
```

4.17.3 Guarded Parameters

Parameters of procedures and functions can be marked as guarded as shown in the example.

```
library Ground_Library;
   ...
   guarded procedure Enable_Monitoring  (guarded Item : Monitor_Item; ...);
   guarded procedure Disable_Monitoring (guarded Item : Monitor_Item; ...);
   ...
   guarded procedure Issue (guarded Command : Command_Item(); ...);
   ...
end Ground_Library;
```

Guarded parameters must be of a pathname or parameterized pathname type. When the procedure or function is called, the calling subprogram or the compilation unit, resp., inherits the privileges of the items that are passed as parameters:

```
procedure AP;
   ...
   import \some\path\Ground_Library;  -- Import is always guarded,
   ...                               -- inherit the library's privileges
begin
   ...
   Issue                             -- Issue is guarded,
      (\some\path\Cmd_1 (...),       -- command parameter is guarded,
       ...;                         -- other parameters are not guarded,
       ...);                        -- so inherit the privileges of Issue
   ...                               -- and \some\path\Cmd_1.
end AP;
```

4.17.4 Dependencies Imposed by Privileges

Please note that inheriting privileges from another item imposes a dependency on that item. This may create a hierarchy of dependencies between a group of items. When the privileges of an item are changed, the item becomes outdated and must be recompiled. Likewise, when an item is changed or recompiled, its privileges may have changed, and so may the privileges of the depending item. Hence, all items depending on this item directly or indirectly must be recompiled. The compiler will keep track of all dependencies and enforce recompilations where necessary.

5 Compilation

5.1 References and Dependencies

When a compilation unit references another unit, by importing it or just by using its pathname or alias in the source text, this reference creates a dependency between the two compilation units. All references are kept in the database and checked by the compiler. The kind of reference determines the strength of the dependency and possibly consequences for the compilation process, e. g.

- Simple references, like declaring an alias for a pathname or using the pathname as a non–parameterized subprogram parameter, just require the referenced unit to exist.
- References to the value of an item, e. g. a sensor or a software variable, imply a dependency on the item's software type properties, thus changing the referenced item will invalidate the referencing compilation unit. It is no longer *up to date* and will have to be recompiled.
- A library body implies a dependency on its specification. So before a body can be compiled, its specification must be compiled and *up to date*.
- Importing an item implies a dependency on its specification and hence requires the specification to be compiled and *up to date* before usage. The body is not concerned, it need not even exist.
- Passing an item as a parameterized parameter implies a dependency on its formal parameter list and hence requires the item to be compiled and *up to date* before usage.
- Using an item in a guarded position implies a dependency on that item's privileges (see 4.17). Changing the privileges of referenced items requires the referencing compilation unit to be recompiled.
- In order to execute an AP, it must of course be *complete*, i. e. it must be compiled and *up to date* (which implies that all its references meet the above mentioned conditions) and all its constituents, including library bodies, must be compiled and *up to date* as well.

The compiler will check all these different kinds of dependencies and enforce all mentioned conditions. If any of its references fails to meet the conditions, the unit cannot be compiled.

5.2 Compilation Order

A consequence of the conditions stated in 5.1 is that compilation units have to be compiled in a specific order:

- A library specification must be compiled before its body.
- A library specification must be compiled before any compilation unit that imports the library.
- An item must be compiled before any compilation unit that passes it as a parameterized parameter.
- Before executing an AP, all its constituents, including bodies, must be compiled.

The compiler provides both a single compilation mode to compile one specific compilation unit, and a special *make* mode that recompiles not only the unit itself but also all constituents of all units referenced directly or indirectly in a valid order, and leaves the compilation unit in *complete* state. Moreover, in *make* mode the compiler performs only the necessary steps, i. e. constituents that are already *up to date* will not be compiled. So, in case the unit is already *complete*, the compiler will do nothing.

APPENDICES

Appendix A: Acronyms

AP	**A**utomated **P**rocedure
APM	**A**ttached (**P**ressurized **M**odule) Laboratory
ASCII	**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
BNF	**B**ackus–**N**aur **F**orm (syntax)
CSS	**C**ore **S**imulation **S**oftware
DB	**D**ata**B**ase
DMS	**D**ata **M**anagement **S**ubsystem
EBNF	**E**xtended **B**ackus–**N**aur **F**orm
EGSE	**E**lectrical **G**round **S**upport **E**quipment
FLAP	**FL**ight **A**utomated **P**rocedure (onboard)
GMT	**G**reenwich **M**ean **T**ime
HCI	**H**uman **C**omputer **I**nterface
HLCL	**H**igh–**L**evel **C**ommand **L**anguage
HW	**H**ard**W**are
ICD	**I**nterface **C**ontrol **D**ocument
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
ISO	**I**nternational **O**rganisation for **S**tandardization
I/O	**I**nput/**O**utput
MDB	**M**ission **D**ata**B**ase
N/A	**N**ot **A**pplicable
NASA	**N**ational **A**eronautics and **S**pace **A**dministration (USA)
ODB	**O**nboard **D**ata**B**ase
OS	**O**perating **S**ystem
SI	**S**ystème **I**nternationale (metric unit standards)
SID	**S**hort **ID**entifier
SW	**S**oft**W**are
TBC	**T**o **B**e **C**onfirmed
TBD	**T**o **B**e **D**efined
UCL	**U**ser **C**ontrol **L**anguage
UIL	**U**ser **I**nterface **L**anguage
VICOS	**V**erification **I**ntegration and **C**heck–**O**ut **S**oftware

Appendix B: Definitions

Access rights	permission users or applications have to access objects or entities.
Application	program or set of programs performing some specialized user–oriented function (as opposed to general–purpose programs like an **operating system**)
Automated Procedure	program (main procedure) written in **User Control Language (UCL)**.
Child	in a hierarchical structure, denotes an immediate descendant of a network or tree component. A child is thus located one hierarchical level below its **parent.**
Compilation unit	smallest unit of code that is accepted by the compiler. In UCL, there are different types of Compilation Units: Automated Procedure (AP), Library Specification, Library Body, Formal Parameter List Definition, Derived Value.
Database	common or integrated collection of interrelated data whose purpose is to serve one or more applications.
Default	a value supplied by the system when a user does not specify a required parameter, qualifier, or attribute.
Derived Value	a database item whose value is computed from the values of other database items.
End item	MDB item located at the lowest hierarchical level (leaf or terminal node), and hence cannot be further decomposed.
Intermediate code (I–code)	binary code generated by the UCL compiler, interpreted at run time.
Item type	type of an MDB item defines the properties (attributes) and the set operations that may be performed on that item.
MDB item, MDB object	*used interchangeably*; uniquely identifiable entity defined in the Mission Database (and corresponds to a real–world HW or SW entity). An MDB Object or Item may be decomposed into lower–level items according to the hierarchical name tree conventions, see **Name tree**.
Mission Database (MDB)	the central repository for all HW/SW configuration information about Flight Elements, Payloads and associated Ground Support Equipment.
Name tree	hierarchical (tree) structure within the MDB which decomposes the system into subsystems, equipment, etc. The topmost node of the name tree is called the root node, whereas terminal nodes (leaf nodes) represent the items that cannot (or need not) be further decomposed, the **end items**. Each MDB object is identifiable by a **pathname** indicating the succession of nodes to be traversed to reach that particular object in the name tree.
Network	group of computers (workstations) and/or terminals that are linked together to allow the sharing of resources (data and peripherals).
Nickname	an alias for a database item, that may be used instead of a pathname. Nicknames are predefined in the database.
Node	component of a network or tree structure.
Operating system (OS)	system software that controls the computer and its parts, performing the basic tasks such as allocating memory, and allowing computer components to communicate.

Parent	in a hierarchical structure, denotes an immediate ancestor of a network or tree component.
Pathname	**see Name tree**
Semantics	rules dealing with the meaning of the language elements (symbols, constants, variables, statements etc.)
Short identifier (SID)	software–generated unique number assigned to each MDB item (allows faster retrieval)
Statecode	identifier (character string) denoting one of many possible states of a discrete end item. (e.g. `$OPEN`, `$CLOSED`, `$HIGH`, `$MEDIUM`, `$LOW`)
Unit	any lower level item in the SW architecture e.g. module, object
User Control Language	**T**est and operations language (used for real–time control & monitoring purposes in both the onboard and ground environment)

Appendix C: *DELETED*

Appendix D: UCL Syntax

The following syntax definition uses the variant of BNF described in section 3.3.

Within the syntax productions, between any terminal symbols, blanks, tabs or line and page breaks may be arbitrarily inserted. Within terminal symbols, no blanks, tabs, line or page breaks are allowed. A comment starts with two hyphens "––" and extends to the end of a line.

In identifiers and reserved words, as well as for the letter e/E in real literals, lower– and upper–case letters are not distinghuished. These letters are shown here in lower–case only, they are to be undestood as representing both the lower–case and upper–case variant.

Please note: Annotations are not explicitly shown in the syntax. They are syntactically treated like comments, but may appear only at defined positions in a compilation unit (see 4.4).

Compilation Units

```
Compilation_Unit =      Main_Procedure |
                        Library_Specification |
                        Library_Body |
                        Formal_Parameter_List_Definition
                        Derived_Value

Main_Procedure =        { Import }
                        { Unit_Declaration | Constant_Declaration | Type_Declaration | Alias_Declaration }

                        ”procedure” Identifier [ Formal_Parameters ] ”;”
                            { Import }
                            { Declaration }
                        ”begin”
                            Statement_Sequence
                        ”end” [ Identifier ] ”;”

Library_Specification = ”library” Identifier ”;”
                            { Import }
                            { Definition }
                        ”end” [ Identifier ] ”;”

Library_Body =          ”library” ”body” Identifier ”;”
                            { Import }
                            { Declaration }
                        [ ”begin”
                            Statement_Sequence ]
                        ”end” [ Identifier ] ”;”

Formal_Parameter_List_Definition =
                        { Import }
                        { Unit_Declaration | Constant_Declaration | Type_Declaration | Alias_Declaration }

                        [ Identifier ] Formal_Parameters ”;”

Derived_Value =         { Import }
                        { Unit_Declaration | Constant_Declaration | Type_Declaration |
                        Variable_Declaration | Alias_Declaration }

                        { If_Statement | Case_Statement | Return_Statement }
```

Declarations

Import = ”import” Name ”;”

Definition = Unit_Declaration |
Constant_Declaration |
Type_Declaration |
Variable_Declaration |
Alias_Declaration |
Procedure_Heading |
Function_Heading

Declaration = Unit_Declaration |
Constant_Declaration |
Type_Declaration |
Variable_Declaration |
Alias_Declaration |
Procedure_Declaration |
Function_Declaration

Unit_Declaration = ”unit” ”[” Unit_Identifier ”]” [”=” Unit] ”;”

Constant_Declaration = ”constant” Identifier ”:” Constant_Type ”:=” Constant_Expression ”;”

Constant_Type = Qualified_Identifier [Unit] |
”string” |
”statecode” |
”pathname” [”.” ”*”]

Type_Declaration = ”type” Identifier ”=” Type ”;”

Variable_Declaration = ”variable” Identifier ”:” Variable_Type [”:=” Constant_Expression] ”;”

Variable_Type = Qualified_Identifier [Unit] |
String_Type |
”statecode” |
”pathname” [”.” ”*”]

Alias_Declaration = ”alias” Identifier ”=” Name [”.” Identifier] ”;”

Procedure_Heading = [”guarded”] ”procedure” Identifier [Formal_Parameters] ”;”

Procedure_Declaration = Procedure_Heading
Block

Function_Heading = [”guarded”] ”function” Identifier [Formal_Parameters] ”:”
Qualified_Identifier [Unit] ”;”

Function_Declaration = Function_Heading
Block

Block = { Declaration }
”begin”
Statement_Sequence
”end” [Identifier] ”;”

Formal_Parameters = ”(” [Parameter_List { ”;” Parameter_List }] ”)”

Parameter_List = [”guarded”] [”in”] [”out”] Identifier_List ”:” Formal_Type
[”:=” Constant_Expression]

Types

Formal_Type = Formal_Simple_Type |
”array” ”of” Formal_Simple_Type |
”string” [”of” Identifier]

Formal_Simple_Type = Qualified_Identifier [”(” ”)”] [Unit] |
”statecode” |
”pathname” [”(” ”)”] |
”pathname” ”.” ”*”

Type = Simple_Type |
String_Type |
Statecode_Type |
Pathname_Type |
Subitem_Pathname_Type |
Array_Type |
Set_Type |
Record_Type |
Inherited_Type

Simple_Type = Qualified_Identifier [Unit] |
Enumeration |
Subrange

Enumeration = ”(” Identifier_List ”)”

Subrange = Qualified_Identifier ”(” Constant_Expression ”..” Constant_Expression ”)”

String_Type = ”string” ”(” Constant_Expression ”)” [”of” Identifier]

Statecode_Type = ”statecode” [”(” Statecode_List ”)”]

Statecode_List = Statecode { ”,” Statecode }

Pathname_Type = ”pathname” [”(” Identifier_List ”)”]

Subitem_Pathname_Type=
”pathname” ”.” ”*” [”(” Identifier_List ”)”]

Array_Type = ”array” ”(” Index_Range { ”,” Index_Range } ”)” ”of” Qualified_Identifier [Unit]

Index_Range = Constant_Expression ”..” Constant_Expression |
Qualified_Identifier

Set_Type = ”set” ”of” Simple_Type

Record_Type = ”record”
{ Fields }
”end” ”record”

Fields = Identifier_List ”:” Qualified_Identifier [Unit] |
”case” Identifier ”:” Qualified_Identifier
Variant_Part
”end” ”case” ”;”

Variant_Part = { ”when” Case_Label_List ”:” { Fields } }
[”else” { Fields }]

Case_Label_List = Case_Labels {”,” Case_Labels} |
Statecode_List

Case_Labels = Constant_Expression [”..” Constant_Expression]

Identifier_List = Identifier { ”,” Identifier }

Inherited_Type = ”type” “of” Name

Statements

Statement = Assignment | Procedure_Call | If_Statement | Case_Statement | While_Statement | Repeat_Statement | Loop_Statement | For_Statement | Halt_Statement | Exit_Statement | Return_Statement

Assignment = Designator ":=" Expression ";"

Procedure_Call = Qualified_Identifier [Actual_Parameters] ";"

Actual_Parameters = "(" [Parameter { "," Parameter }] ")"

Parameter = [Identifier ":"] Expression

If_Statement =
"if" Expression "then"
    Statement_Sequence
{ "elsif" Expression "then"
    Statement_Sequence }
["else"
    Statement_Sequence]
"end" "if" ";"

Case_Statement =
"case" Expression
    Case
    { Case }
    ["else" Statement_Sequence]
"end" "case" ";"

Case =
"when" Case_Label_List ":"
    Statement_Sequence

While_Statement =
"while" Expression "do"
    Statement_Sequence
"end" "while" ";"

Repeat_Statement =
"repeat"
    Statement_Sequence
"until" Expression ";"

Loop_Statement =
"loop"
    Statement_Sequence
"end" "loop" ";"

For_Statement =
Iterative_For_Statement |
Collective_For_Statement

Iterative_For_Statement =
"for" Identifier ":=" Expression "to" Expression ["by" Constant_Expression] "do"
        Statement_Sequence
    "end" "for" ";"

Collective_For_Statement =
"for" Identifier "in" Qualified_Identifier ["by" Constant_Expression] "do"
        Statement_Sequence
    "end" "for" ";"

Halt_Statement = "halt" [Expression] ";"

Exit_Statement = "exit" ";"

Return_Statement = "return" [Expression] ";"

Statement_Sequence = { Statement }

Expressions

Constant_Expression =	Expression
Expression =	Relation { ”&” Relation } | Relation { ”|” Relation } | Aggregate
Aggregate =	”(” [Expression{ ”,” Expression}] ”)”
Relation =	Simple_Expression [”=” Simple_Expression | ”<>” Simple_Expression | ”<” Simple_Expression | ”<=” Simple_Expression | ”>” Simple_Expression | ”>=” Simple_Expression | ”in” Simple_Expression]
Simple_Expression =	[”+” | ”–”] Term { ”+” Term | ”–” Term }
Term =	Factor { ”*” Factor | ”/” Factor | ”%” Factor }
Factor =	Primary [”**” Factor]
Primary =	Number [Unit] | String | Character | Set_Constant | Date [Time] | Time | Statecode | Designator | Function_Call | Type_Conversion | ”(” Expression ”)” | ”~” Primary
Set_Constant =	[Qualified_Identifier] ”{” [Element { ”,” Element }] ”}”
Element =	Constant_Expression [”..” Constant_Expression]
Function_Call =	Qualified_Identifier [Actual_Parameters]
Type_Conversion =	Qualified_Identifier ”(” Expression ”)” | String_Conversion
String_Conversion =	[“pathname” | “alias” | “unit”] Qualified_Identifier ”(” Expression [“,” Format] ”)” | [“pathname” | “alias” | “unit”] “string” ”(” Expression [“,” Format] ”)”
Format =	Width [”,” Aft [“,” Exp]]
Width, Aft, Exp =	Expression
Qualified_Identifier =	[[Name] ”.”] Identifier
Designator =	Name { ”.” (Identifier | ”(” Expression_List ”)”) }
Slice =	Designator ”(” Expression “..” Expression ”)”
Expression_List =	Expression { ”,” Expression }
Simple_Name =	Identifier { Path_Identifier } | Pathname
Name =	Simple_Name { “.” Identifier }
Pathname =	”\” | ”\\” | Path_Identifier { Path_Identifier }
Subitem_Pathname =	Name ”.” Identifier

Unit Expressions

Unit =	”[” Unit_Expression ”]”
Unit_Expression =	[Numerator [”/” Denominator] [”+” Offset | ”–” Offset]]
Offset =	Number [”/” Number]
Numerator =	Unit_Term | ”(” Unit_Term ”)”
Denominator =	Number | Unit_Factor | ”(” Unit_Term ”)”
Unit_Term =	[Number] Unit_Factor { Unit_Factor } | Number
Unit_Factor =	Unit_Identifier { Digit }
Unit_Identifier =	Letter { Letter }

Terminal Symbols

Unit_Factor =	Unit_Identifier { Digit }
Path_Identifier =	”\” (Letter | ”_” | Digit) { Letter | ”_” | Digit }
Statecode =	”$” Identifier | ”$$”
Identifier =	Letter { [”_”] Letter_Or_Digit }
Unit_Identifier =	Letter { Letter }
Letter_Or_Digit =	Letter | Digit
Letter =	”a” | ”b” | ”c” | ”d” | ”e” | ”f” | ”g” | ”h” | ”i” | ”j” | ”k” | ”l” | ”m” | ”n” | ”o” | ”p” | ”q” | ”r” | ”s” | ”t” | ”u” | ”v” | ”w” | ”x” | ”y” | ”z”
Number =	Simple_Integer | Based_Integer | Real
Simple_Integer =	Digits
Based_Integer =	Digits ”#” Hex_Digit { Hex_Digit } ”#”
Real =	Digits ”.” Digits [”e” [”+” | ”–”] Digits]
Date =	Day ”.” Month ”.” Year
Day =	[Digit] Digit
Month =	[Digit] Digit
Year =	Digit Digit Digit Digit
Time =	Hours ”:” Minutes [”:” Seconds [”.” Fraction]] | ”~:~”
Hours =	[Digit] Digit
Minutes =	Digit Digit
Seconds =	Digit Digit
Fraction =	Digits
Digits =	Digit { [”_”] Digit }
Hex_Digit =	Digit | ”a” | ”b” | ”c” | ”d” | ”e” | ”f”
Digit =	”0” | ”1” | ”2” | ”3” | ”4” | ”5” | ”6” | ”7” | ”8” | ”9”
String =	Char_String | Byte_String
Char_String =	”””” { ASCII | ”””” ”””” } ””””
Byte_String=	”#” ”””” [Hex_Digit Hex_Digit { [”_” | ” ”] Hex_Digit Hex_Digit }] ””””
Character =	”””” Printable ””””
Printable =	*any of the printable characters of the underlying character set*

Appendix E: ASCII Character Set

(see ISO standard 646, Ref.doc. 2.2.5, for details)

	0	1	2	3	4	5	6	7
0	NUL,	SOH,	STX,	ETX,	EOT,	ENQ,	ACK,	BEL,
8	BS,	HT,	LF,	VT,	FF,	CR,	SO,	SI,
16	DLE,	DC1,	DC2,	DC3,	DC4,	NAK,	SYN,	ETB,
24	CAN,	EM,	SUB,	ESC,	FS,	GS,	RS,	US,
32	' ',	'!',	'"',	'#',	'$',	'%',	'&',	''',
40	'(',	')',	'*',	'+',	',',	'-',	'.',	'/',
48	'0',	'1',	'2',	'3',	'4',	'5',	'6',	'7',
56	'8',	'9',	':',	';',	'<',	'=',	'>',	'?',
64	'@',	'A',	'B',	'C',	'D',	'E',	'F',	'G',
72	'H',	'I',	'J',	'K',	'L',	'M',	'N',	'O',
80	'P',	'Q',	'R',	'S',	'T',	'U',	'V',	'W',
88	'X',	'Y',	'Z',	'[',	'\',	']',	'^',	'_',
96	'`',	'a',	'b',	'c',	'd',	'e',	'f',	'g',
104	'h',	'i',	'j',	'k',	'l',	'm',	'n',	'o',
112	'p',	'q',	'r',	's',	't',	'u',	'v',	'w',
120	'x',	'y',	'z',	'{',	'|',	'}',	'~',	DEL

Appendix F: UCL/MDB Type Correspondence Table

The following table shows for a few examples the principle, how MDB item types map on UCL access classes and software types. This is to be understood as an example only, the actual MDB item types, the corresponding access classes and software types are defined in the database documentation for the respective target system.

MDB object type		UCL object type
MDB item type class	**access class**	
analog measurement	READ	REAL
discrete measurement	READ	*statecode*
analog stimulus	SEND	–
discrete stimulus	SEND	–
SW variable	READ/WRITE	INTEGER REAL BOOLEAN ...
UCL library	IMPORT	–
network node	NODE SELECT	–
SW unit	*none* (*)	–
nametree node	PATH SELECT	–
AP	EXECUTE	–

* ”none” means: MDB objects of this type can be accessed only via System Library procedures/functions

Appendix G: Engineering Units

This section is based on the International Standard ISO 1000 “SI units and recommendations for the use of their multiples and of certain other units”.

G–1 Base Units

The international system of units (Système International d’Unités, SI) defines seven *base units*. (For a definition of the SI base units, see the ISO 1000 standard, annex B.)

Quantity	Unit	SI symbol	UCL symbol
1. length	meter	m	m
2. mass	kilogram	kg	kg
3. time	second	s	s
4. electric current	ampere	A	A
5. temperature	kelvin	K	**Kabs**
temperature difference	kelvin	K	**K**
6. amount of substance	mole	mol	mol
7. luminosity	candela	cd	cd

Note that UCL makes the distinction between *absolute temperature* (Kabs) and *temperature difference* (K). The distinction is necessary to allow conversions between e.g. °C and K; in particular, the relationship: 0 °C = 273.15 Kabs does not hold for temperature differences.

G–2 SI Units

In the table of SI units given below, the column “SI unit” refers to the name of the unit as defined by ISO 1000 and the international system of units (Système International d’Unités, SI) or to unit names composed by SI unit names (e.g. m/s). The column “Sym.” (symbol) shows the recommended string to be used in UCL to denote such a unit, the same is true for the “supported multiples of this unit” in the next columns. The “Definition” column exactly defines the unit.

Quantity	SI unit	Sym.	Supported multiples of this unit						Definition
length	meter	m	km	cm	mm	um	nm	pm	*base unit*
	astronomic unit	AU							1 AU = 149597.870 • 10^6 m
	parsec	pc							1 pc = 206265 AU
area	m^2	m2	km2	dm2	cm2	mm2			1 m^2 = 1 m • m
volume	m^3	m3	dm3	cm3	mm3				1 m^3 = 1 m • m • m
	liter	l, L	hl	cl	ml				1 l = 1 dm^3
	Note that both the upper case letter 'L' and the lower case letter 'l' denote the unit liter.								
mass	kilogram	kg	g	mg	ug				*base unit*
	atomic mass unit	u							1 u = 1.66053 S 10^{-27} kg
	tonne	t							1 t = 10^3 kg
time	second	s	ms	us	ns				*base unit*
	minute	min							1 min = 60 s
	hour	h							1 h = 60 min
	day	d							1 d = 24 h
electric current	ampere	A	kA	mA	uA	nA	pA		*base unit*
temperature	kelvin	Kabs							*base unit*
	degree Celsius	degC							1 °C = 1 K + 273.15
temp. difference	kelvin	K							*base unit*
amount of substance	mole	mol	kmol	mmol	umol				*base unit*
luminosity	candela	cd							*base unit*
plane angle	radian	rad	mrad	urad					*supplementary unit*=m/m
solid angle	steradian	sr							*supplementary unit*=m^2/m^2
frequency	hertz	Hz	THz	GHz	MHz	kHz			1 Hz = 1 s^{-1}
rotational freq.	s^{-1}	1/s	1/min						
force	newton	N	MN	kN	mN	uN			1 N = 1 kg m/s^2
pressure	pascal	Pa	GPa	MPa	kPa	mPa	uPa		1 Pa = 1 N/m^2
	bar	bar	mbar	ubar					1 bar = 10^5 Pa

Quantity	SI unit	Sym.	Supported multiples of this unit						Definition
energy, work, heat	joule	J	TJ	GJ	MJ	kJ	mJ		1 J = 1 N m
	electronvolt	eV	GeV	MeV	keV				1 eV = 1.60219 S 10^{-19} J
torque	Nm	Nm	MNm	kNm	mNm	uNm			1 Nm = 1 N m = 1 J
power	watt	W	GW	MW	kW	mW	uW		1 W = 1 J/s
electric charge	coulomb	C	MC	kC	mC	uC	nC	pC	1 C = 1 A s
	Ah	Ah	mAh	uAh					1 Ah = 3.6 kC
electric potential	volt	V	MV	kV	mV	uV			1 V = 1 J/C
electric capacitance	farad	F	mF	uF	nF	pF			1 F = 1 C/V
electric resistance	ohm Ω	Ohm	GOhm	MOhm	kOhm	mOhm			1 W = 1 V/A
electric conductance	siemens	S	kS	mS	uS				1 S = 1 W^{-1}
magnetic flux	weber	Wb	mWb						1 Wb = 1 V s
magnetic induction	tesla	T	mT	uT	nT				1 T = 1 Wb/m^2
inductance	henry	H	mH	uH	nH	pH			1 H = 1 Wb/A
luminous flux	lumen	lm							1 lm = 1 cd sr
illuminance	lux	lx							1 lx = 1 lm/m^2
activity (of a radionuclide)	becquerel	Bq							1 Bq = 1 s^{-1}
absorbed dose	gray	Gy							1 Gy = 1 J/kg
dose equivalent	sievert	Sv							1 Sv = 1 J/kg
velocity	m/s	m/s							
	km/h	km/h							1 km/h = (1/3.6) m/s
	knot	knot							1 knot = 0.51$\overline{4}$ m/s
angular velocity	rad/s	rad/s							
acceleration	m/s^2	m/s2							
density	kg/m^3	kg/m3		kg/l	g/l				
linear mass density	kg/m	kg/m		mg/m					

Quantity	SI unit	Sym.	Supported multiples of this unit			Definition
momentum	kg m/s	kg m/s				
angular momentum	kg m^2/s	kg m2/s				
moment of inertia	kg m^2	kg m2				
viscosity	Pa s	Pa s	mPa s			
specific acoustic impedance	Pa s/m	Pa s/m				
acoustic impedance	Pa s/m^3	Pa s/m3				
kinematic viscosity	m^2/s	m2/s	mm2/s			
volume flow rate	m^3/s	m3/s	l/s			
surface tension	N/m	N/m	mN/m			
linear expansion coefficient	K^{-1}	1/K				
thermal conductivity	W/(m K)	W/(m K)				
coefficient of heat transfer	W/(m^2K)	W/(m2 K)				
heat capacity	J/K	J/K	kJ/K			
specific heat capacity	J/(kg K)	J/(kg K)				
specific internal energy	J/kg	MJ/kg	kJ/kg			
charge density	C/m^3	C/m3				
surface density of charge	C/m^2	C/m2				
electric field strength	V/m	V/m	MV/m	kV/m	mV/m	uV/m
permittivity	F/m	F/m	uF/m	nF/m	pF/m	
electric polarization	C/m^2	C/m2	kC/m2	mC/m2	uC/m2	
electric dipole moment	C m	Cm				

Quantity	SI unit	Sym.	Supported multiples of this unit	Definition
current density	A/m^2	A/m2	A/mm2	
linear current density	A/m	A/m	A/mm	
magnetic vector potential	Wb/m	Wb/m		
permeability	H/m	H/m	uH/m nH/m	
electromagnetic moment	A m2	A m2		
magnetization	A/m	A/m	A/mm	
magnetic dipole moment	Wb m	Wb m		
resistivity	Ω m	Ohm m	GOhm m MOhm m KOhm m mOhm m	
conductivity	S/m	S/m	MS/m kS/m	
reluctance	H^{-1}	1/H		
radiant intensity	W/sr	W/sr		
radiance	$W/(sr\ m^2)$	W/(sr m2)		
irradiance	W/m^2	W/m2		
quantity of light	lm s	lm s		
luminance	cd/m^2	cd/m2		
luminous exitance	lm/m^2	lm/m2		
light exposure	lx s	lx s		
luminous efficacy	lm/W	lm/W		
mechanical impedance	N s/m	N s/m		
molar mass	kg/mol	kg/mol	g/mol	
molar volume	m^3/mol	m3/mol	l/mol	
molar internal energy	J/mol	KJ/mol		
molar heat capacity	J/(mol K)	J/(mol K)		
concentration of substance B	mol/m^3	mol/m3		
molality of solute substance B	mol/kg	mol/kg		

G–3 Non–SI Units

For convenience, the following non–SI units are also supported.

Quantity	Unit	Sym.	Supported multiples of this unit	Definition
length	inch	in		1 in = 25.4 mm
	An inch is defined to be exactly 25.4 mm. It is used as the “base unit” for imperial lengths.			
	foot	ft		1 ft = 12 in
	yard	yd		1 yd = 3 ft
	mile	mi		1 mi = 1760 yd
mass	pound	lb		1 lb = 453.592 g
	A pound is defined to be exactly 453.592 g. It is used as the “base unit” for imperial masses.			
	ounce	oz		1 oz = (1/16) lb
	grain	gr		1 gr = $(1/437^{1/2})$ oz
	hundredweight	cwt		1 cwt = 112 lb
	ton	ton		1 ton = 20 cwt
temperature	degree Fahrenheit	degF		1 degF = 10/18 degC – 320/18

G–4 Prefix Names and Values

The SI prefixes (also in the ISO 1000 standard) are used with the unit symbols defined above.

E	exa	10^{18}	
P	peta	10^{15}	
T	tera	10^{12}	
G	giga	10^{9}	
M	mega	10^{6}	
k	kilo	10^{3}	
h	hecto	10^{2}	
da	deca	10^{1}	
d	deci	10^{-1}	
c	centi	10^{-2}	
m	milli	10^{-3}	
u	micro	10^{-6}	*should be μ (mu) but this is not in the ASCII character set*
n	nano	10^{-9}	
p	pico	10^{-12}	
f	femto	10^{-15}	
a	atto	10^{-18}	

Appendix H: Implementation Constraints

The following restrictions are not inherent to the language; rather, they are due to the specific implementation of the UCL compiler and its runtime environment. Restrictions on UCL procedures also apply to UCL functions.

- Maximum length of a line of text, and hence of an identifier, is 256 characters.
- Maximum number of local variables/parameters in one procedure is restricted to 65536 (or 2^{16}).
- Maximum number of global variables/parameters in one compilation unit is 65536 (or 2^{16}).
- Maximum number of procedures in one compilation unit is 65536 (or 2^{16}).
- Maximum size of a record variable is 65536 words (or 2^{16}).
- Maximum number of elements in a set is 65536 (or 2^{16}).
- Maximum number of components in an array is 2^{31}–1.
- Maximum number of characters in a string is 2^{31}–1.
- Maximum number of imported user libraries in one AP is 65536 (or 2^{16}).
- Maximum number of imported system libraries in one AP is 65536 (or 2^{16}).
- Maximum size of a boolean expression (containing & or | operators) is 65536 (or 2^{16}) bytes.

The UCL Compiler is part of the Columbus Ground System (CGS).

INDEX

H

I

L

M

N

O

P

Q

R

S

T

U

V

W