

**EUROTHERM**

---

# **SPECIFICATION OF CDL**

## **Technical Specification**

---

© COPYRIGHT MCMXCII EUROTHERM LIMITED  
All rights strictly reserved. No part of this document may be  
stored in a retrieval system, or transmitted, in any form or by  
any means without prior written permission from Eurotherm  
Ltd

**Draft**

John Juet

## Contents

<b>1</b>	<b>Scope</b>	<b>6</b>
<b>2</b>	<b>Overview of CDL</b>	<b>7</b>
2.1	Execution Model . . . . .	8
2.2	An Example Resource . . . . .	9
<b>3</b>	<b>Lexical conventions</b>	<b>11</b>
3.1	Identifiers . . . . .	11
3.2	Keywords . . . . .	12
3.3	Literals . . . . .	12
3.3.1	Numeric Literals . . . . .	12
3.3.2	Character string literals . . . . .	13
3.3.3	Time literals . . . . .	13
3.4	Comments . . . . .	14
3.5	Other separators and built in operators . . . . .	14
<b>4</b>	<b>Data Types</b>	<b>15</b>
4.1	Elementary data types . . . . .	15
4.2	Variables . . . . .	15
<b>5</b>	<b>Expressions</b>	<b>19</b>
5.1	Operators . . . . .	19
5.1.1	Arithmetical operators . . . . .	20
5.1.2	Logical or bit string operators . . . . .	20
5.1.3	Date and time operators . . . . .	21
5.1.4	Comparison operators . . . . .	21
5.1.5	Indexing . . . . .	21
5.1.6	Precedence . . . . .	22
5.2	Functions . . . . .	23
5.2.1	User defined functions . . . . .	23
5.2.2	Using simple functions . . . . .	24
5.2.3	Functions and arrays . . . . .	25
5.2.4	Standard functions . . . . .	26

---

<b>6</b>	<b>Function Blocks</b>	<b>27</b>
6.1	Default Initial Values . . . . .	29
6.2	Wiring . . . . .	31
6.3	Programs . . . . .	32
<b>7</b>	<b>Statements</b>	<b>33</b>
7.1	Assignment statements . . . . .	33
7.2	Function and function block control statements . . . . .	33
7.3	Selection statements . . . . .	33
7.3.1	If statements . . . . .	35
7.3.2	Case statements . . . . .	36
7.4	Iteration statements . . . . .	37
7.4.1	For statements . . . . .	37
7.4.2	While statements . . . . .	39
7.4.3	Repeat statements . . . . .	39
7.4.4	Exit statement . . . . .	40
<b>8</b>	<b>Sequential Function Charts</b>	<b>41</b>
8.1	Steps . . . . .	41
8.2	Transitions . . . . .	42
8.3	Actions . . . . .	42
8.4	SFC execution . . . . .	46
8.5	SFC actions . . . . .	47

## CONTENTS

---

<b>9</b>	<b>References</b>	<b>48</b>
9.1	Specifying the Remote Data Objects . . . . .	49
9.1.1	Simple types . . . . .	49
9.1.2	Arrays . . . . .	50
9.1.3	Function Blocks . . . . .	51
9.1.4	Services . . . . .	53
9.2	Reading the Remote Information . . . . .	53
9.3	Reading Remote Data . . . . .	54
9.4	Writing Remote Data . . . . .	54
9.5	Reference Time stamp . . . . .	55
9.6	References to References . . . . .	55
9.6.1	Adjustment of scan rates . . . . .	56
9.7	Examining the state of a reference . . . . .	57
9.8	Timeouts and failures . . . . .	57
9.9	Summary of Properties . . . . .	58
9.10	Addressability and Write Protection . . . . .	58
<b>10</b>	<b>Resources</b>	<b>59</b>
10.1	Tasks and task execution model . . . . .	61
10.2	Remote blocks . . . . .	61
<b>11</b>	<b>Services</b>	<b>61</b>
11.1	Service declaration . . . . .	62
11.2	Service invocation . . . . .	63
11.3	Service execution . . . . .	64
11.3.1	Immediate services . . . . .	64
11.3.2	Rendezvous services . . . . .	65
11.4	Service timeouts . . . . .	68
11.5	Services with SFC bodies . . . . .	68
<b>12</b>	<b>Attributes</b>	<b>69</b>
12.1	Attribute definition . . . . .	70
12.2	Attributes at run time . . . . .	71

---

13 Configuration

72

14 Deviations from IEC-1131

72

## VERSION HISTORY

Version	Date	Changes
Draft	July 24, 1992	

# 1 Scope

This document describes Configuration Description Language, CDL, a language for configuring real-time distributed control systems based on the IEC standard for programmable controller, IEC 1131.

CDL is a textual language based on IEC 1131-3 Structured Text with extensions defined by Eurotherm which incorporate previous Eurotherm Group experience of requirements of distributed control systems.

The IEC standard identifies a set of languages which include two graphical languages, Sequential Function Charts and Function Block Diagrams. These map onto and can be mixed with Structured Text. Hence CDL can be used in a mixed-mode graphical and textual programming system, and can be thought of as the canonical form that describes the meaning of the graphical languages.

This document describes CDL, identifies where the language extends the IEC standard and also features of the IEC standard that are currently not supported.

Products that conform to CDL should not introduce any features that are supersets of what is described in this document. Products should provide a document identifying any CDL features not supported, or supported with restrictions.

This document actually reflects the status of the current Eurotherm Control Limited's implementation of CDL. It is known that

- The current implementation in some respects does not conform to the IEC standard (or even to levels of conformance of the IEC standard)
- The current implementation does not fully meet the requirements of other group companies (for example EPA's implementation of SFC).

On the other hand it would be impossible to write a document that did meet all the above requirements since the IEC standard is still changing, and the requirements of the group companies are not yet fully known. So at least this document describes a known implementation.

It is intended that in the fullness of time CDL will be extended so that

- An exact level of conformance to IEC is specified.
- The extensions to IEC are clearly stated, as required by the IEC standard.
- The requirements of other group companies are incorporated.

This document also describes an execution model for CDL. It is not intended that every product that conforms to CDL should conform exactly to the execution model. There should, however, be a clear statement of the mapping between CDL and the product.

It is also anticipated that many products will only support a subset of CDL, for example

- Only a fixed set of blocks may be supported
- Only Resource level configuration may be supported

Section 14 summarises the current differences between CDL and IEC 1131-3.

## 2 Overview of CDL

The top level object in CDL is a *Configuration*. This defines a network topology. It specifies which nodes certain Resources are going to run on.

The next level object is the *Resource*. Typically there is one Resource per node.

Under a Resource are *Resource Level Objects*. Most of these are *Resource Level Blocks* which may be either *Function Blocks*, *Programs*, or (in the future) *Services*. Function Blocks, Programs and Services are together simply called *Blocks* and a Resource Level Block is a Block declared at the top level in the Resource. A Resource may also contain data holders which are CDL variables, such as structures, enumerations, arrays or base types (INT, REAL, etc.). *Note that today the only valid Resource Level Objects are Task, Programs and Function Blocks*

A Block consists of data and an executable algorithm which may represent a continuous control strategy, a sequencing strategy, a display strategy, or the like. Each Block may have input values and output values. A Block can be thought of as a software IC with its input and output values being equivalent of the pins of the IC. Inputs and outputs of Resource Level Blocks may be 'wired' to other Resource Level Blocks in the same way as chips in a printed circuit board are wired together.

The execution of Resource Level Blocks is controlled by special Resource Level Objects called *Tasks*. These are real time tasks which can be executed at a regular interval or on an event. A Task is associated with a set of Resource Level Blocks that it controls. The Task executes each of the Resource Level Blocks that it controls in turn when it is time for the Task to run. It is important to understand that the algorithm in blocks is run each time the controlling Task runs. So any CDL fragment may be executed many many times without any looping constructs being required.

Resource Level Blocks are (like ICs) themselves hierarchic. They can be made up out of a set of smaller Blocks that are also wired together, inside the top level Resource Level Block. Blocks inside a Resource Level Block must be Function Blocks or Services (i.e Programs can only be instantiated at the resource level).

Inter Resource communication is achieved by the *Reference* mechanism. A Reference is a Block that provides a template for mapping onto other objects, either on the same Resource or on another remote Resource. The mapped objects may be any arbitrary collection provided that they all are executed by the same Task. The Reference can be changed at run time (from within CDL) to map onto another set of objects. Type checking for the mapping is performed (once only) at run time. References can be used to set up scanning (i.e wiring) or to single shot read or write data to unwired Block parameters.

A Block type in CDL is an algorithm with some associated data. Some of the data is provided as inputs, some available as outputs, and some is internal to hold the state of the Block. The data represents the instantiation of the Block; a Block type can be used many times in any Resource with different data just as a type of chip can exist many times on a PCB.

CDL contains traditional language constructs such as looping constructs and conditional statements, with Block execution in between. CDL also has a set of built in types (Reals, Integers, Strings, Times and so on) and allows user definable Blocks. CDL is strongly typed; it is illegal to connect or assign conflicting types together. CDL has a set of built in Functions (e.g add, multiply) and allows user definable Functions. (A *Function* is an algorithm with no state giving one output value i.e given the same input values the same output value is always returned.)

CDL also allows specification of Sequential Function Charts which divide Block execution into a series of *Steps* with associated *Actions*. When a Step is active its associated Actions are executed. These Actions are expressed as CDL statements i.e they can be a set of wired Function Blocks that will be executed or another Sequence Flow Chart or some looping and conditional statements. Actions may be executed continuously (i.e all the time the step is active), or only once (when the step becomes active), or various other timed combinations.

Steps have *Transitions* between them, which are Boolean expressions that must evaluate to true in order for the transition to be made. When a transition is made the currently active Step is deactivated and the next

Step is made active. There can be a transition to a set of parallel steps which are all activated together, or there can be a rendezvous between a set of parallel steps into one step.

As previously said, every Block has an execution algorithm associated with it. Blocks can also have other algorithms associated with them. These are known as *Services*. Services are used to provide and receive data from Blocks without explicit wiring, and are useful when there is a need to abstract a set of operations on a Block, and when there is a many to many relationship between Blocks.

Services can be invoked using the Reference mechanism to set up a local image of the remote Service. Service invocations can be accepted either asynchronously to Block execution or at an appropriate user defined moment during Block execution. The former are known as 'method' Services, the latter as 'rendezvous' Services.

Any CDL object has associated with it a set of *Attributes*. Attributes are static characteristics of CDL objects such as engineering units, version numbers and so on.

In summary,

- CDL is a language for distributed applications. The Reference mechanism hides from an application whether the referenced objects are on the local node or on another node; there is no inter-node communication visible at the CDL programming interface, only References to external Blocks (including Services).
- Each Resource contains a set of Blocks which are simply wired together. The Blocks themselves execute under the control of Tasks. The algorithms inside a Block can be expressed in CDL statements which encompass block wiring and sequencing. A Block forms a reusable type; Blocks can contain other Blocks and are therefore hierarchic.
- CDL is based on the IEC 1131-3 Structured Text language. CDL extends IEC 1131-3 which does not have References, Services, or Attributes.

## 2.1 Execution Model

CDL is a deterministic real time language<sup>1</sup>. Tasks are executed either at a regular interval, or on an event. Any Task executed at a regular interval must complete before the next execution is due, otherwise an overrun condition is signalled on a Task Block output variable. The values seen by a data receiving Task in any inter task communication is a *coherent* set from the same cycle of the data providing Task. Whether inter task communication requires inter node communication is invisible to the application. The execution model can be described as follows.

Tasks are pre-emptable real time tasks. Each Task runs according to its associated priority either on a single shot basis when it is triggered, or (the normal case) at a periodic interval. During its execution a Task executes each of its Resource Level Blocks (RLBs) in turn. Input wiring to an RLB is evaluated just before the RLB is executed. In between execution of an RLB it deals with any read, write or Service requests.

---

<sup>1</sup>It is entirely possible for a product to use CDL to describe applications, but to have a different non-IEC execution model.



## 2.2 An Example Resource

Textual (CDL) representation :-

```
(*
 * The Resource Definition
 * =====
 *)

RESOURCE R ON XXXX
  (* declare two tasks and their intervals *)
  TASK T1 ( INTERVAL := t#1s );
  TASK T2 ( INTERVAL := t#3s );
  (* instantiate a program block *)
  PROGRAM Fb1 WITH T1 : FbTypeA (In := Fb2.Out);
  (* instantiate a function block *)
  FUNCTION_BLOCK Fb2 WITH T2 : FbTypeA (In := Fb1.Out, In1 := Fb3.Out);
  (* declare a remote block on another remote resource R1.
    The block's update rate is controlled by task T2. *)
  REFERENCE Fb3 WITH T2: FbTypeB { ref := 'R1:Fb3' };
END_RESOURCE

(*
 * The Function Block TYPE Definitions
 * =====
 *)

FUNCTION_BLOCK FbTypeA

VAR_INPUT
  In, In1 : INT;
END_VAR

VAR_OUTPUT
  Out : INT;
END_VAR

VAR
  B,C : FbTypeB;
END_VAR

(* algorithm of block
  ...
  *)

END_FUNCTION_BLOCK

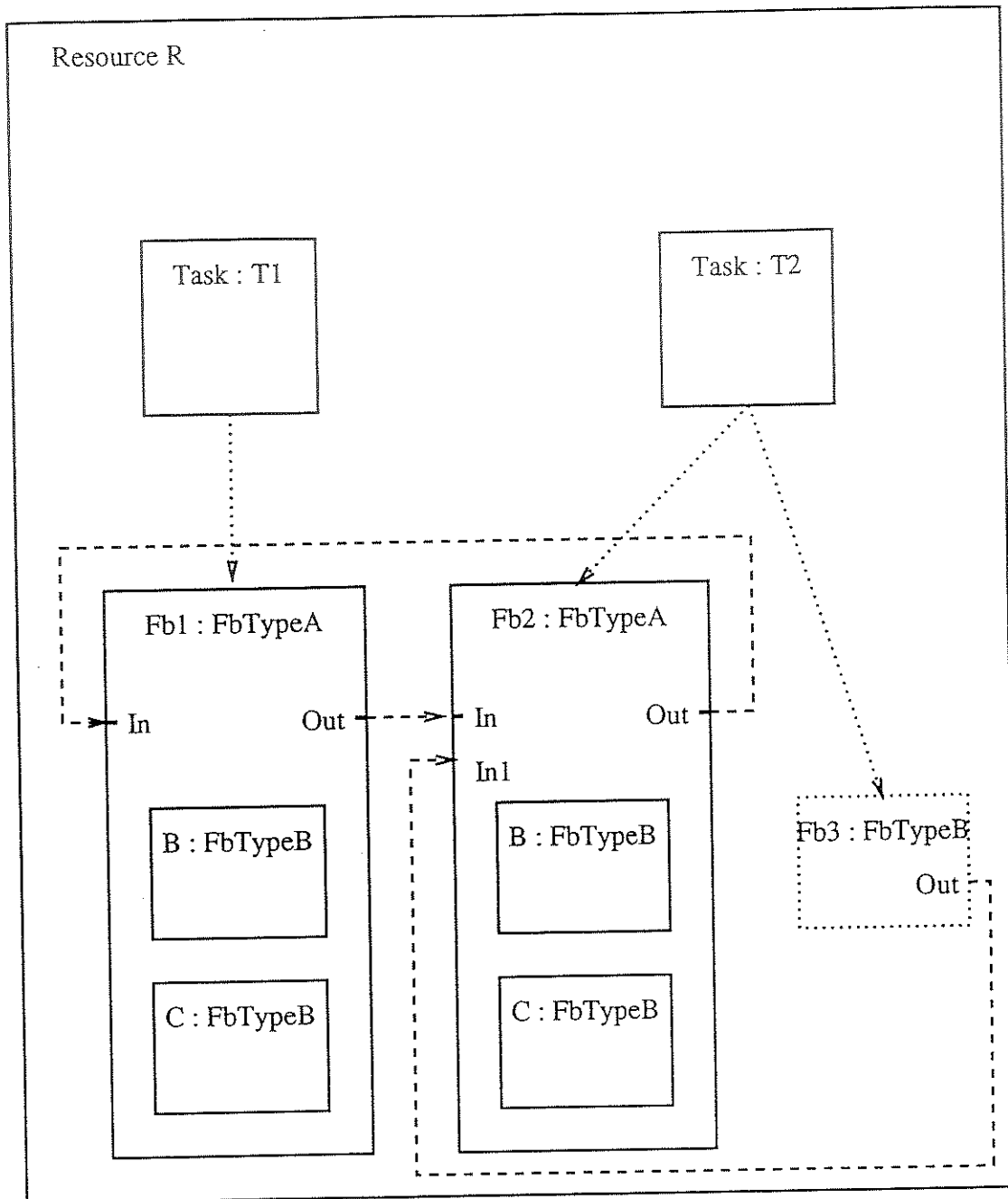
FUNCTION_BLOCK FbTypeB

VAR_OUTPUT
  I : INT;
END_VAR

(* algorithm of block
  ...
  *)
```

END\_FUNCTION\_BLOCK

Graphically this can be represented as in fig 1.



example\_resource, IPH, 28/8/90

Figure 1: Example Resource

### 3 Lexical conventions

CDL description is divided for processing (i.e translation for loading into a run-time control system) into a sequence of tokens. There are five kinds of tokens, identifiers, literals, operators, keywords and other separators.

Blanks or “white space” (tabs, newlines, formfeeds and comments) are ignored except that they serve to separate tokens.

CDL uses the printable ascii character set for its literals, keywords, identifiers, operators and separators.

#### 3.1 Identifiers

An *identifier* is string of letters, digits and underline characters that begins with an underscore or a letter. Underline characters and case are significant, i.e “ABCD” and “AB\_CD” and “abcd” are different identifiers. Multiple leading or embedded underlines are not allowed. Identifiers cannot contain white space.

There are some restrictions on length of identifiers.

Input, in\_out and output VARs of function blocks should be no more than 12 characters long and may not start with underscores. Internal function block parameters (i.e not input, output or in\_out) may be more than 12 characters long, but if so the name is truncated to 12 characters as far as access to the data using the Reference mechanism (section 9) is concerned. Function block type names may be up to 20 characters long. Function block instance names are of course VARs of another block (or program) and so have the same limit as internal VARs.

Table 1: CDL Keywords

ACTION	ACCEPT†	AND	ARRAY
BOOL	BY	BYTE	CASE
DATE	DATE_AND_TIME	DIM†	DINT
DO	DWORD	EDGE††	ELSE
ELSIF	END_ACTION	END_CASE	END_FOR
END_FUNCTION_BLOCK	END_FUNCTION	END_IF	END_PROGRAM
END_REPEAT	END_RESOURCE	END_SERVICE†	END_STEP
END_TRANSITION	END_VAR	END_WHILE	EXIT
FOR	FROM	FUNCTION	FUNCTION_BLOCK
IF	INITIAL_STEP	INT	LINT
LREAL	MOD	MULTIPLY	NON_ST_BODY†
NOT	NO_BODY†	NO_INPUTS †	NO_OUTPUTS†
OF	ON	OR	PROGRAM
QTIME†	REAL	REFERENCE†	REPEAT
RESOURCE	RETAIN	RETURN	ROW†
SCREEN†	SERVICE†	SINT	STEP
STRING	TABLE†	TASK	THEN
TIME	TIME_OF_DAY	TO	TRANSITION
UDINT	UINT	UNTIL	USINT
VAR	VAR_INPUT	VAR_IN_OUT	VAR_OUTPUT
WHEN	WHILE	WITH	WORD
XOR			

†These keywords are not defined in IEC DIS 1131-3

†EDGE has been replaced by rising and falling edge in IEC, it will be replaced in CDL.

## 3.2 Keywords

Keywords are unique combinations of characters of the same form as identifiers. Keywords are all in upper case. Keywords are shown in table 1.

## 3.3 Literals

### 3.3.1 Numeric Literals

There are two classes of numeric literals, integer literals or real literals.

Integer literals may be specified in conventional decimal notation. Integer literals may also be represented in base 2. The base is in decimal notation. The base is followed by a '#' sign and the digits giving the value of the number. The digits must be in groups of four, separated by underscores, e.g 2#0000 2#0010\_0001.

Real literals are distinguished by the presence of a decimal point. An exponent signifies an integer power of ten (specified in decimal notation) by which the preceding number is multiplied to obtain the value represented. An exponent is indicated by the letter E or e followed by a + or – sign and then the integer exponent value.

Integer and real literals and exponents may be preceded by a + or – sign.

Boolean data can be represented by 0 for false, and 1 for true, as well as the pre-declared constant identifiers FALSE (value 0 ) and TRUE (value 1).

Base 2 notation may only be used when assigning or comparing bit string types.

Table 2: Examples of numeric literals

Integer literals	-12 0 1 40967 +3000
Real literals	-12.0 0. 10.1 .11 1.e-12 1.0E+12 .11E+12
Base 2 literals	2#1111.0000 2#0000
Boolean literals	0 1 TRUE FALSE

Table 3: Examples of character string literals

'a string'	A character string of length 8
' '	The empty, zero length, character string
' '	A string containing the space character
'\$ '	A string of length one containing the single quote character
'\$OD\$OA'	A string of length two containing CR and LF
'\$\$1.0'	The string that prints as \$1.0
'\$L' or '\$l'	A string containing one line feed character, LF
'\$P' or '\$p'	A string containing one form feed character
'\$N' or '\$n'	A string containing one new line character
'\$R' or '\$r'	A string containing one carriage return character
'\$T' or '\$t'	A string containing one tab character

### 3.3.2 Character string literals

A character string literal is a sequence of zero or more characters prefixed and terminated by the single quote character '. In character strings three character combinations of \$ followed by two hexadecimal digits (in upper-case) are hexadecimal representations of the eight bit character code. Additionally two character combinations which begin with the dollar sign have a special meaning as shown in table 3.

String literals in CDL have a maximum length of 255.

### 3.3.3 Time literals

There are two sorts of time literals, those that represent an elapsed time or a duration, and those that represent an absolute time, i.e a date and time of day combination.

#### Duration literals

**Millisecond resolution durations** These are delimited on the left by the sequence of characters T# and t#. (TIME# or time# may also be used in IEC 1131-3, CDL currently does not support this).

The duration data is a sequence of numeric literals and duration units which are days (denoted by 'd'), hours (denoted by 'h' or 'H'), minutes (denoted by 'm' or 'M'), seconds (denoted 's' or 'S') and milliseconds (denoted 'ms', or 'MS'), or any combination of these in sequence. (In IEC 1131-3 the least significant time unit may be give in real notation, with no exponent; CDL currently does not support this.) Examples are give in table 4.

Table 4: Time literal formats

t# format duration literals	t#14ms T#14.7ms t#25h10mslms t#5d13h t#25h1m
QT# format duration literals	QT#1slms QT#10s1.2ms QT#100s
Date literals	D#1989-06-25 DATE#2000-10-25
Time of day literals	TIME_OF_DAY#15:36:33 TOD#12:20:11
Date and time literals	DT#2000-10-25-10:11:12 DATE_AND_TIME#1989-06-25-12:20:11

Table 5: Prefixes for time of day and date literals

Date literals	DATE# or D#
Time of day literals	TIME_OF_DAY# or TOD#
Date and time literals	DATE_AND_TIME# or DT#

Overflow in the most significant unit of a duration literal is permitted.

(In IEC 1131-3 single underscores may be used to separate units of duration, CDL currently does not support this.)

In CDL the maximum duration that may be represented is  $2^{32}$  milliseconds, (about 49.7 days).

**Microsecond resolution durations** Certain products require a time accuracy of microseconds. CDL specifies another literal format for these, which allows  $2^{32}$  microseconds to be specified. The format is like duration literals, except that the sequence of time durations starts with "QT#", and is specified in seconds and milli seconds only; real format with no exponent and up to 3 digits after the decimal point may be used for the milli-second part. For example QT#1.3ms is 1300 $\mu$ s, QT#1s2.003ms is 1002003 $\mu$ s, and QT#2.12ms is 2120 $\mu$ s.

**Time of day and date literals** There are three sorts of absolute time literals, "time of day", "date and time" and "date". The prefixes for these are shown in table 5. Some examples are shown in table 4. Date is in the format 'year-month-day', where year is the four digit year, month the two digit month number, and day the two digit day of the month. Time of day is in the format 'hour:minute:second'. Date and time literals are of the format '<date and time literal>-<time of day literal>'.

### 3.4 Comments

Comments are delimited at the beginning by the character sequence (\*, and at the end by \*). Comments are allowed anywhere in CDL except within character string literals defined in section 3.3.2. Comments are treated as white space (see section 3). Comments may not be nested.

### 3.5 Other separators and built in operators

Other separators and the built in operators that make up the lexical structure of CDL are shown in table 6.

Table 6: CDL separators and operators

..	Separate array bounds declarations
.	Hierarchic name separator
;	Statement delimiter
:	Type declarations
:=	Assignment operator
<=	Comparison operator
>=	Comparison operator
<>	Comparison operator
<	Comparison operator
>	Comparison operator
=	Comparison operator
+	Add operator
-	Subtract operator, unary minus
*	Multiply operator
/	Divide operator
**	Power operator
(	Start of parameter lists
)	End of parameter lists
[	Array indexing, bounds specification
]	Array indexing, bounds specification
,	General list delimiter
~	Access properties
::=	wiring operator
{	Delimit attribute assignment
}	Delimit attribute assignment

## 4 Data Types

### 4.1 Elementary data types

Table 7 shows the supported elementary data types.

### 4.2 Variables

A variable is a named place holder for a data type. Different values of the same data type may be assigned to a variable. Identifiers are used for the symbolic representation of variables. Variables are assigned values using assignment statements. Assignments are denoted using a `:=`. The left hand side of the assignment is the variable being assigned to, the right hand side an expression (see section 5), which in the simplest case is a literal value, or another variable. The types of the left hand and right hand side of assignments must match. An example of an assignment of a literal value to a variable called "A" of type DINT is —

```
A := 10;
```

An example of an assignment between two variables, "C" and "B", of the same type is —

```
C := B;
```

Table 7: Elementary data types

BOOL	true or false boolean
SINT	Eight bit signed integer
INT	16 bit signed integer
DINT	32 bit signed integer
USINT	8 bit unsigned integer
UINT	16 bit unsigned integer
UDINT	32 bit signed integer
REAL	single precision (32 bit) floating point
LREAL	double precision (65 bit) floating point
TIME	Duration, 32 bit milliseconds
QTIME	Duration, 32 bit microseconds
TIME_OF_DAY or TOD	Time of day
DATE_AND_TIME or DT	Date and time of day†
DATE	Date‡
STRING	Variable length (255 max) string
BYTE	8 bit bit string
WORD	16 bit bit string
DWORD	32 bit bit string
ROW	‡
TABLE	‡

†The range of this is 00:00 1/1/1970 to  $2^{32}$  seconds later.

‡These data types are used to provide an interface to relational databases. Within the CDL context variables of this type may only be assigned to variables of the same type or manipulated by functions.

A variable may also represent an array of data values, of up to 6 dimensions (5 in the case of STRINGS).

On start up variables may be initialised to a system default value or a user specified initial value.

Variables are declared and used in *Program Organisation Units*, which are functions (5.2), function blocks and programs (6), and services (11).

Variables are declared in the declaration section of Program Organisation Units. Variables have a mode, which defines whether it can be read or written outside and inside the Program Organisation Unit it is declared in. The mode is specified in the declarations section of the Program Organisation Unit and can be —

**VAR\_INPUT** The variable is an input to the Program Organisation Unit and may only be read within it, and only written outside it.

**VAR\_OUTPUT** The variable is an output from the Program Organisation Unit and may be read and written within it but only read outside it.

**VAR\_IN\_OUT** The variable is an input-output to the Program Organisation Unit and may be read and written within it and outside it.

**VAR** The variable is internal to the Program Organisation Unit, and may only be read and written within it.

The scope of the name of a variable is local to the Program Organisation Unit the variable is declared in, except by explicit parameter passing via variables that have been declared as input or outputs or in-outs.

A simple declaration of a DINT variable is

```
VAR
  adint: DINT;
END_VAR
```



A list of names may be used as in

```
VAR
  (* 3 strings that can contain 80 characters *)
  string1, string2, string3: STRING(80);

  (* Default string size of 255 characters *)
  string255: STRING;
  dint1, dint2, dint3 : DINT;
END_VAR
```

This example also shows declaration of an optional string size, in this case 80 bytes, which is the maximum number of characters the string will take. If this is omitted string size has a default of 255 (the maximum allowed in CDL).

Variables may be initialised by an expression, which is known as a cold start value,

```
adint1: DINT := 10; (* custom initialisation *)
adint2, adint3 : DINT := -1; (* adint2 and adint3 both initialised to -1 *)
```

If variables have no explicit cold start value, they are initialised to a default value. The default values of all numeric (integer or floating point types) is zero. For duration the default value is 0 seconds. For time and date and time types the default value is 00:00 a.m 1st January 1970. For all bit string types the default value is 0.

CDL allows expressions as well as constants to be used as cold start values. (This allows cold start values to be passed down through hierarchies of blocks, see 6). Expressions (see section 5) may refer to variables in previously declared declaration sections as in

```
VAR_INPUT
  string1: STRING;
END_VAR
VAR
  string2: STRING := string1;
END_VAR
```

string2 is initialised to the same value as string1. Since string1 can be initialised from outside the block (see 6) a cold start value can effectively be passed to string2 from outside the block.

CDL allows arrays of variable to be declared. For example

```
VAR_INPUT
  array1: ARRAY [ 1 .. 10 ] OF DINT;
END_VAR
```

declares a single dimensional array of 10 DINTs. The left bound of the array must be less than the right bound, and both must be  $\geq 0$ .

```
VAR_INPUT
  array2: ARRAY [ 5 .. 10, 1 .. 20 ] OF DINT;
END_VAR
```

declares a two dimensional array of 100 DINTs. Up to 6 dimensional arrays are supported, except for arrays of strings, for which only 5 dimensions are allowed. Arrays may be initialised using comma separated lists of initialisation values. The rightmost subscript of the array varies most rapidly. For example

```
VAR_INPUT
  array3: ARRAY [ 1 .. 2, 1 .. 3, 1 .. 3 ] OF DINT :=
    1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18;
END_VAR
```

initialises the three dimensional array; because the rightmost subscript of the array varies most rapidly, element [1,1,1] has value 1, [1,1,2] value two, and so on, [2,3,3] has value 18.

Parentheses can be used to repeat values in array initialisation, so for example

```
VAR_INPUT
  array4: ARRAY [ 1 .. 10 ] OF DINT :=
    6(1,2), 3(30), 1;
END_VAR
```

gives elements [1],[3],[5] of array4 the value 1, [2],[4],[6] of array4 the value 2, elements [7],[8],[9] the value 30, and element [10] the value 1.

CDL allows expressions to appear in array initialisation lists, for example

```
VAR_INPUT
  IN : DINT;
END_VAR
VAR
  array5: ARRAY [ 1 .. 4 ] OF DINT :=
    IN, IN + 1, IN + 2, IN + 3;
END_VAR
```

initialises the array to a set of values dependent on the initial value of the input parameter. Since IN can be initialised from outside the block (see 6) a cold start value can effectively be passed to array5 from outside the block.

It is an error to specify too many or too few values to an array initialisation.

String arrays are allowed — for example

```
VAR
  strarr: ARRAY [ 1 .. 3, 1 .. 2 ] OF STRING(20) :=
    3('str1','str2');
```

initialises a two dimensional array of strings each of maximum length 20 so that elements [1,1], [2,1], and [3,1] have value str1 and [1,2], [2,2], and [3,2] have value str2.

CDL allows array to array assignment provided the *overall sizes* of the arrays match. That is if the arrays are mapped into a one-dimensional array, the number of elements must match. For example —

---

```

VAR
  darr1: ARRAY [ 1 .. 3, 1 .. 2 ] OF DWORD;
  darr2: ARRAY [ 1 .. 6 ] OF DWORD;
  darr3: ARRAY [ 1 .. 2, 1 .. 3 ] OF DWORD;
END_VAR

```

```

  darr1 := darr1;
  darr2 := darr1;
  darr3 := darr1;

```

Assignment is done in the same way as in array initialisation; the right hand subscript of arrays is varied the fastest (on both sides of the assignment, and corresponding array elements assigned to each other. So the statement `darr2 := darr1;` is equivalent to

```

darr2[1] := darr1[1,1];
darr2[2] := darr1[1,2];
darr2[3] := darr1[2,1];
darr2[4] := darr1[2,2];
darr2[5] := darr1[3,1];
darr2[6] := darr1[3,2];

```

In string assignment if the destination string is too small the assigned string is truncated to fit. For example

```

VAR
  string: STRING(4);
END_VAR
  string := '1234567';

```

results in string containing 1234.

## 5 Expressions

An expression is a sequence of operators and functions (e.g add, subtract) and operands (either literals or variables) that specifies a computation; an expression always results in a value.

Expressions are strictly type checked. There are strict rules regarding the types and mixing of types allowed as operands; in general the same type must be specified for each operand. Type conversion functions must be used if type mixing is desired.

### 5.1 Operators

Operators are an infix shorthand notation for functions. For example

```
X * 10
```

means

```
MULTIPLY( IN1 := X, IN2 := 10)
```

This section describes operators that are built into CDL.<sup>2</sup>

---

<sup>2</sup>Note that the associated function name is not necessarily built into CDL; it is provided in descriptions below to aid understanding

### 5.1.1 Arithmetic operators

The following arithmetic operators are supported:

Function	Operator	Example	Value
ADD	+	X := 20 + 4	24
SUBTRACT	-	X := X - 10	14
MULTIPLY	*	X := 3 * 5	15
DIV	/	X := X / 2	7
MODULUS	MOD	X := 3 MOD 2	1
EXPT	**	Y := 4.0 ** 3	64.0
MINUS	-	Y := -Y	-64.0

These operators may be used on any of the numeric types (i.e the integer types, or the floating point types).

Add, subtract, multiply and divide must all operate on two arguments of exactly the same CDL type, so it is illegal to add an INT to a SINT say. Integer literals may be used together with any of the integer types, floating point literals may be used with REAL or LREAL types. Integer division truncates its result as shown in the example in the table.

Modulus may only be used with integer types.

Expt (power) must be used with the variable or number having its power taken being a float (REAL or LREAL), and the actual power being any numeric value.

The + operator may also be used to concatenate strings. For example

```
'Hello ' + 'World'
```

is equivalent to

```
'Hello World'
```

### 5.1.2 Logical or bit string operators

The following bit string operators are supported:

Function	Operator	Example	Value
XOR	XOR	X := 2#1010 XOR 2#0010	2#1000
OR	OR	X := 2#0001 OR 2#1000	2#1001
AND	AND	X := 2#0001 AND 2#1100	2#0000
NOT	NOT	X := NOT 2#0101	2#1010

All these operators must have operands of the same type, and the operands must be bit strings types (BYTE, WORD, DWORD) or booleans (BOOL) or bit string literals.

### 5.1.3 Date and time operators

Standard arithmetic operators are available for use with date and time types as shown in the table below.

Operator	Operand type	Operand type	Result type
+	TIME	TIME	TIME
	TIME_OF_DAY	TIME	TIME_OF_DAY
	DATE_AND_TIME	TIME	DATE_AND_TIME
-	TIME	TIME	TIME
	DATE	DATE	TIME
	TIME_OF_DAY	TIME	TIME_OF_DAY
	TIME_OF_DAY	TIME_OF_DAY	TIME
	DATE_AND_TIME	TIME	DATE_AND_TIME
	DATE_AND_TIME	DATE_AND_TIME	TIME
	TIME	Any integer or float	TIME
*	TIME	Any integer or float	TIME

For the  $-$  operator the table above shows the left-hand operand type on the left. It is not possible, for example, to subtract a "time of day" from a "time".

### 5.1.4 Comparison operators

The result type of any comparison is BOOL. Both operands must be of the same type.

The following comparison functions are supported:

Function	Operator	Example	Value
GT	>	1 > 2	0
GE	>=	2 >= 1	1
EQ	=	1 = 1	1
LE	<=	1 <= 3	1
LT	<	1 < 3	1
NE	<>	2 <> 3	1

The operands may be of any type, except for STRING where only = and <> are supported.

### 5.1.5 Indexing

Elements of an array are accessible by using an index expression. Index expressions must be of integer type.

For example, within an expression,

```
a[10]
```

yields the value of element 10 of array a, as does

```
a[5 - 3 + 8]
```

and, if variable z has value 11, so does

```
a[z - 1]
```

The indexed variable

`b[5,10]`

yields the value of element [5,10] of array `b`. If multi-dimensional arrays are considered as being layed out a linear single dimension array then (as in array initialisation, section 4.2), the right most subscript varies most quickly. So if `b` above had been declared as

```
b : ARRAY [1 .. 10, 1 .. 10] OF BYTE
```

element [5,1] would map onto absolute element 40 (where the first element is element 0).

Indexed expressions may either be fully indexed, so that one element is specified, or partially indexed so that a slice is specified. For example given the above declaration of `b`, `b[1,1]` is fully indexed, whereas `b[1]` is a slice of 10 elements. It is an error to specify too many indices. Slices may be used in assignments or with array functions (see page 18).

#### 5.1.6 Precedence

Expressions may be bracketed to determine the order of evaluation. For example

```
(( 5 * 4 ) / 10) + ( 3 - (20/4)) = 0
```

Otherwise order of evaluation is determined by precedence. For example the operator `-` has lower precedence than the operator `*` so `5 * 4 - 4` means `(5 * 4) - 4`.

Operators have the following precedence, (highest to lowest).

- functions and parenthesised expressions
- POWER operator `**`
- unary operator NOT, and unary minus `-`
- DIVIDE `/`, multiply `*`, modulus MOD
- ADD `+` and SUBTRACT `-`
- comparison `<>=<=>=<>`
- AND
- XOR
- OR

Otherwise precedence is by grouping left to right i.e `1 * 4/10` is equivalent to `(1 * 4)/10`.

## 5.2 Functions

A function is a *program organisation unit* which when executed yields exactly one value. A function call may be used in any expression; the type of the function call is the type of the value it returns.

Functions have a set of input parameters, which must be assigned values before the function is invoked to yield its value.

Functions are stateless. The value that a function returns is always a function of its input parameter values, and always the same for the same set of input values.

CDL requires that function parameters are given values by name, not by the position in the text (unlike many computer languages). For example

```
X := ADD( IN1 := 10.1 , IN2 := SUB( IN1 := 5.1, IN2 := X));
```

is (assuming ADD and SUB are defined as one would expect) a long hand way of writing

```
X := 10.1 + ( 5.1 - X);
```

IN1 and IN2 are the input parameters of the functions. In the function call they must be assigned expressions of the same type as they were declared as in the function declaration. In the function call they can appear in any order. Every input parameter must be given a value by an assignment.

Function definitions are split into a function declaration section, where the function's name, its return type, the function's input parameters and any temporary internal variables are declared, and a function body section where the algorithm defining the function is defined.

Functions may not be recursive, that is the body of a function cannot call itself.

### 5.2.1 User defined functions

**Simple functions** The following shows a simple example of a function that just returns the value passed into it.

```
FUNCTION simple : REAL
VAR_INPUT
  IN : REAL;
END_VAR
  simple := IN;
END_FUNCTION
```

The result of the function is given by assigning a value to the function block name — in other words within the body of the function the name of the function can be treated as the single “output” variable.

CDL guarantees that if the function is given no value by an execution of the function body it will have the default initial value of the function type. For example the following function

```
FUNCTION bad: REAL
VAR_INPUT
  IN : REAL;
END_VAR
END_FUNCTION
```

would always result in a value of 0.0.

Functions may have a body that contains any set of CDL statements (section 7). They may also have internal variables. Internal values may be initialised explicitly; they are initialised on every call of the function. For example

```
FUNCTION EX1: LREAL
VAR_INPUT
  IN1, IN2 : LREAL;
END_VAR
VAR
  TEMP: LREAL := 10.01;
  TEMP1 : LREAL;
END_VAR
  TEMP := TEMP * IN2;
  TEMP1 := TEMP * TEMP;
  EX1 := IN1 - TEMP - TEMP1;
END_FUNCTION
```

Here TEMP is initialised to 10.01 on every call of the function, TEMP1 is initialised to the default initial value for LREALs, 0.0, on every call of the function.

Internal variables are hidden from users of functions — only the input parameters and the value of the function are visible.

Note that within the function body the function name acts like an extra internal variable, so it can appear in expressions. For example the statement

```
EX1 := EX1 * 2;
```

could be added to the function EX1 above.

### 5.2.2 Using simple functions

Simple non-array functions may be used in expressions just like single element variables, literals or fully indexed array variables. In the following assignment the above function is used three times,

```
X := EX1( IN1 := 10.0, IN2 := 11.0 ) +
      EX1( IN1 := EX1( IN1 := 100.0 / Y, IN2 := 11.0 + 10 ),
          IN2 := 3.0e-1);
```

Values are assigned to parameters of functions by an assignment statement. Type checking is strict, so that the types of the left and right hand side of assignments, and the types in complex expressions must match.



### 5.2.3 Functions and arrays

Functions may be defined that return arrays and/or have input parameters that are arrays, for example —

```
FUNCTION TRIV : ARRAY [ 1 .. 10 ] OF DINT
VAR_INPUT
  IN : ARRAY [ 1 .. 10 ] OF DINT;
END_VAR
  TRIV := IN;
END_FUNCTION
```

The rules for array to array assignment (see page 18) are followed when using such functions, so the following are all legal because the size of the arrays or array slices being used all match —

```
VAR
  a: ARRAY [ 1 .. 10 ] OF DINT;
  b: ARRAY [ 1 .. 2, 1 .. 5 ] OF DINT;
  c: ARRAY [ 1 .. 4, 1 .. 2, 1 .. 5 ] OF DINT;
END_VAR

  a := TRIV ( IN := b );
  b := TRIV ( IN := a );
  c[1] := TRIV ( IN := b );
  b := TRIV ( IN := c[2] );
  c[2] := TRIV ( IN := c[1] );
  c[2] := TRIV ( IN := TRIV ( IN := b ) );
  c[2] := TRIV ( IN := TRIV ( IN := TRIV ( IN := c[3] ) ) );
```

CDL also supports the concept of functions that can take arguments of arrays of any size. A built in expression DIM is available to query the actual dimensions of a variable. DIM(name,N) returns the actual size of dimension N of variable name.

For example a function to do generic matrix multiplication would be —

```
FUNCTION m_mult : ARRAY [......] OF DINT
VAR_INPUT
  in1, in2 : ARRAY [......] OF DINT;
END_VAR
VAR
  i,j,k : DINT;
END_VAR
  IF DIM(in1,2) <> DIM(in2,1) THEN
    RETURN;
  END_IF;
  FOR i := 1 TO DIM(in1,1) DO
    FOR j := 1 TO DIM(in2,2) DO
      FOR k := 1 TO DIM(in1,2) DO
        m_mult[i,j] := m_mult[i,j] + in1[i,k] * in2[k,j];
      END_FOR;
    END_FOR;
  END_FOR;
END_FUNCTION
```

(FOR statements are described in detail in section 7.4.1.) Here is an example of using the above function —

```
VAR
  m1,m2: ARRAY [1..2,1..2] OF DINT :=
    1,0,0,1;
END_VAR
  m1 := m_mult( in1 := m1, in2 := m2);
```

When using such functions the actual size of the parameters or values being assigned to must be determinable without analysing the algorithm of the function. It is illegal to nest such functions without specifying a size. So the following is an error

```
(* Illegal because the size of the value assigned to the
   first in2 is not known *)
m1 := m_mult( in1 := m1, in2 := m_mult( in1 := m1, in2 := m2));
```

In such circumstances the user can specify an actual size for the parameter when calling the function, for example —

```
(* Actual size of in2 is specified to be 2 by 2 matrix *)
m1 := m_mult( in1 := m1, in2[1..2,1..2] :=
  m_mult( in1 := m1, in2 := m2));
```

#### 5.2.4 Standard functions

CDL has some built in functions. Operators are “built in functions” with a special syntax. In addition type conversion functions are built in.

Products that conform to CDL may have other built in functions.

**Type conversion functions** Type conversion functions are used to convert from one built in data type to another. They may be used to do, for example, mixed integer and floating point arithmetic. Type conversion functions are always of the form

<OLD TYPE>\_TO\_<NEW TYPE>

For example REAL\_TO\_DINT converts a REAL value to a DINT. By convention the input parameter to type conversion functions is always called IN.

The following is an example of use of some type conversion functions —

```
VAR
  a : DINT;
  b : LREAL;
END_VAR
  a := 10 * LREAL_TO_DINT(IN := b);
  b := DINT_TO_LREAL(IN := a * 100 -
    LREAL_TO_DINT(IN := b));
```

## 6 Function Blocks

A function block is a Program Organisation Unit, which, when executed, yields one or more values. Multiple, named, *instances* of a function block can be created. Each has an associated identifier *the instance name* and a data structure that contains its input, output and internal variables. All the values of this data structure persist from one invocation of the function block to the next; therefore invocation of a function block with the same input parameter values does not always yield the same output values.

A function block can be thought of as a software integrated circuit. In the same way as one type of integrated circuit can be used many times in a particular printed circuit board, so a function block type can be instantiated many times in CDL.

Function blocks can also be thought of as a user defined data type. In the same way as a DINT type can be instanced (by declaring a variable of type DINT) so a function block can be instanced by declaring a variable of the function block type.

Function blocks will always run under control of a Task<sup>3</sup>. Therefore any statements in a Function Block are executed every time the task runs, unlike in an ordinary sequential language such as Pascal, where an explicit loop is required to repeat statements.

Only the input and output parameters of a function block are accessible when a block is instanced — the internal variables are hidden.

Function block definitions consist of a function block declaration section, where the function block type name, the variables (input, output and internal) are declared, and a function block body section, where the algorithm defining the function block execution is given. The body consists of either a set of statements (section 7) or a sequential function chart (section 8).

A simple function block declaration is shown below —

```
FUNCTION_BLOCK Run_Av
VAR_INPUT
    Enable : BOOL;
    Input  : REAL;
END_VAR
VAR_OUTPUT
    Output : REAL;
END_VAR
IF Enable THEN
    Output := ( Input + Output ) / 2 ;
ELSE
    Output := Input ;
END_IF ;
END_FUNCTION_BLOCK
```

(IF statements are fully described in section 7.3.1.)

As noted above function blocks can be instanced inside other function blocks. They are declared in the VAR<sup>4</sup> section as if they were types like DINT, LREAL etc. So a block containing two instances of Run\_Av might be —

<sup>3</sup>strictly speaking it is the Resource Level Block that runs under control of the Task and hence its sub-blocks — see section 3.3

<sup>4</sup>CDL at present restricts function block instances to being internal (i.e they may not be input, output or in out)

## FUNCTION BLOCKS

---

```
FUNCTION_BLOCK Use_run_Av
VAR_INPUT
    Enable1, Enable2 : BOOL;
    Input1, Input2   : REAL;
END_VAR
VAR_OUTPUT
    Output1, Output2 : REAL;
END_VAR
VAR
    R1, R2 : Run_Av;
END_VAR
```

Function block instances can be called using function block call statements, which consist of the instance name of the block, followed by named parameter assignments. Continuing the above example, `Use_run_Av` could call its two instances of `Run_Av` as follows —

```
R1( Enable := Enable1, Input := Input1);
R2( Enable := Enable2, Input := Input2);
```

Because input data is preserved between one call and the next of a function block instance, it is not necessary to specify a value for all the input parameters (unlike for functions). The previous value assigned to the input parameter is used. For example, in

```
Input2 := 3.0;
R1( Enable := 1, Input := Input1 + 10.0);
R1( Input := Input2 + 1.1);
Input2 := 10.0;
R1();
```

the second call of `R1` will still have `Enable` set to 1, and the third call will still have `Input` set to its previous value of 4.1.

Inputs to function block instances may only be assigned to when the block is called. They may not be used in expressions.

Values of `VAR_OUTPUT`s of function block instances are always available to be used in expressions. They are accessed using a hierarchic name syntax, where a “.” is used to separate levels in the hierarchy. For function blocks only outputs (and in-outs) are visible so there are only two levels in the hierarchy (except when using Services, see section 11). For example the outputs of the `Use_run_Av` function block above could be set from the values of the outputs of the instances `R1` and `R2` of `Run_Av` above as follows —

```
Output1 := R1.Output;
Output2 := R2.Output + 3.1;
```

Function blocks may also have `VAR_IN_OUT` declarations. Only variables or other function block in-out parameters may be passed to a function block in-out parameter. Furthermore variables and in-out parameters passed to an in-out parameter may be modified from within the function block body. If, for example, a function block was declared as

```
FUNCTION_BLOCK inout
VAR_IN_OUT
    A: DINT;
END_VAR
    A := A + 1;
END_FUNCTION_BLOCK
```

and was instanced and called as follows —

```
VAR
  x : inout;
  Y: DINT;
END_VAR
  Y := 1;
  x( A := Y);
```

then the value of Y after the call to instance x of function block type inout would be 2.

in-out parameters of a function block instances may be used in any expression like output parameters. So the next statement in the above example could be

```
Y := 1 + x.A;
```

and Y will now have the value 3.

CDL allows arrays of function block instances. An array of inout function blocks could be declared as

```
arr : ARRAY [ 1 .. 10 ] OF inout;
```

To access or call elements of the function block instance array indexing must be used —

```
arr[1]( A := Y);
Y := 10 * arr[1].A + 10;
```

## 6.1 Default Initial Values

Variables in function blocks either have a default initial value or a user supplied initial value, as described on page 17. The initial value is referred to as a cold start value. It is however possible to override the initial value of inputs of function blocks when they are instanced.

Suppose a function block type called EX has the declaration in it —

```
VAR_INPUT
  IN : DINT := 100;
  IN2 : DINT := 20;
END_VAR
```

By default this means that the initial value of IN has the value 100 and that IN2 has the value 20.

It is possible to override this when instancing the function block using a parameter list with a set of values passed to cold start each parameter —

```
VAR
  INST: EX ( IN := 10000 );
  INST2: EX ( IN2 := 1, IN := -20 );
END_VAR
```

The values passed in the declaration of the instances INST and INST2 override the initialisation in the function block declaration.

Furthermore CDL allows any expression to be used to cold start parameters. This means that initial values can be passed all the way down the function block hierarchy, by initialising function block instance parameters from function block input parameters. So if the block EX had within it some other function block instances, initial values could be passed through as follows —

```
FUNCTION_BLOCK EX
VAR_INPUT
  IN : DINT := 100;
  IN2 : DINT := 20;
END_VAR
VAR
  AINST: XX ( IN := IN, IN2 := IN2 );
END_VAR
```

By default the parameters AINST.IN and AINST.IN2 will have values 100 and 20 respectively. However if when the block EX is instanced, cold start values for its parameters are specified these will be passed down to AINST.IN and AINST.IN2.

Arrays of function blocks are initialised in the same way as arrays of simple variables, by specifying a comma separated list of initialisers, except that the initialisers are assignments to input parameters of the block, (see page 18). For example —

```
VAR
  AINST: ARRAY [ 1 .. 4 ] OF XX ( IN := 3, IN2 := Y ),
                                     3( IN := 1, IN2 := 3 ) ;
END_VAR
```

initialises as follows —

```
XX[1].IN = 3, XX[1].IN2 = Y,
XX[2].IN = 1, XX[2].IN2 = 3,
XX[3].IN = 1, XX[3].IN2 = 3,
XX[4].IN = 1, XX[4].IN2 = 3
```

Function block inputs and outputs may also be arrays. Within the function block they may be initialised using array initialisation syntax. When instancing the block this initialisation may be overridden using explicit indexing or array to array assignment, or by using array initialisation syntax. The block of type YY shown below

```
FUNCTION_BLOCK YY
VAR_INPUT
  IN1, IN2, IN3: ARRAY [ 1 .. 5 ] OF DINT :=
    1, 2, 3, 4, 5;
END_VAR
```

has user supplied default initialisation of its three input arrays.

When instanced this can be overridden as follows —

```

VAR
  I : ARRAY [ 1 .. 5 ] OF DINT;
  YINST : YY (
    (* Initialisation using array to array assignment *)
    IN1 := I,
    (* Initialisation using indexing *)
    IN2[1] := 1 , IN2[2] := 2, IN2[3] := I[1],
    (* Initialisation using array initialisation syntax *)
    IN3 := { 3 , 4 , -1 , I[2] , 11 }
  );

```

Note that { } has to be used to bracket any array initialisation syntax within function block initialisation.

## 6.2 Wiring

Sometimes it is desirable to always evaluate an expression and give its value to a function block input, or a function block output, on every call of the block.

CDL provides the wiring operator `::=` to allow this.

Suppose, regardless of what path was taken through the body of a function block, one of its outputs always had the value of the sum of two inputs, and an internal variable. One way of implementing this is to assign the expression to the output when the output is declared i.e

```

VAR_OUTPUT
  OUT1 : DINT ::= IN1 + IN2 + INTRNL1;
END_VAR

```

It is then an error to attempt to assign any other value to OUT1 in the block body. The wiring expression, in this case `IN1 + IN2 + INTRNL1`, will be evaluated at the end of the block body execution, and the resulting value assigned to the wired output. Wiring expressions are evaluated in the order they are declared in at the end of block execution.

It is also possible to wire inputs of function block instances. Again it is an error to assign a value to a wired input anywhere else in the block body. The wiring expression is evaluated on every call of the block instance. For example —

```

VAR_OUTPUT
  F : FTYPE ( IN ::= (INO + IN1) / 2);
END_VAR

```

wires the input F.IN to the expression `(INO + IN1) / 2`.

As a final example, the following two blocks are equivalent, though one, using wiring, is shorter than the other —

```

FUNCTION_BLOCK XX
VAR_INPUT
    IN0, IN1 : DINT;
END_VAR
VAR_OUTPUT
    OUT1 : DINT := F.OUT + IN0;
END_VAR
VAR
    F : FTYPE ( IN := 0, IN :=
                (IN0 + IN1) / 2 );
END_VAR
IF IN0 > 1000 THEN
    F ( LARGE := TRUE );
ELSE
    F( LARGE := FALSE );
END_IF;
END_FUNCTION_BLOCK

```

```

FUNCTION_BLOCK XX
VAR_INPUT
    IN0, IN1 : DINT;
END_VAR
VAR_OUTPUT
    OUT1 : DINT;
END_VAR
VAR
    F : FTYPE( IN := 0 );
END_VAR
IF IN0 > 1000 THEN
    F ( LARGE := TRUE , IN := (IN0 + IN1) / 2);
ELSE
    F( LARGE := FALSE , IN := (IN0 + IN1) / 2);
END_IF;
OUT := F.OUT + IN0;
END_FUNCTION_BLOCK

```

The example also shows cold starting the function block input IN as well as wiring it.

It is possible to wire the whole of an array, but NOT single elements of any array. The usual rules for array to array assignment are followed (see page 18).

### 6.3 Programs

Programs are equivalent to function blocks that cannot be instantiated inside other function blocks — therefore they can only appear at the Resource level, see section 10.

Programs are delimited by the PROGRAM END\_PROGRAM key words, e.g —

```

PROGRAM FbTypeA

VAR_INPUT
    In : INT;
END_VAR

VAR_OUTPUT
    Out : INT;
    B : FbTypeB;
END_VAR

VAR
    C : FbTypeB;
END_VAR

(* algorithm of program *)

END_PROGRAM

```



## 7 Statements

Table 8 shows the statements supported by CDL. Statements are always delimited by “;”.

### 7.1 Assignment statements

Assignment statements replace the value of a variable by the result of evaluating an expression. An assignment statement consists of a variable name, on the left hand side, followed by the assignment operator `:=`, followed by the expression to be evaluated. For example the expression

```
A := B;
```

replaces the value of the variable A by the value of the variable B.

See also section 4.2, 5.2.6 and page 18.

### 7.2 Function and function block control statements

Function evaluation is part of expressions (see section 5 and section 5.2).

Function blocks are invoked by a statement consisting of the name of the function block instance followed by a parenthesised list of named input or in-out parameter value assignments, as described in section 6 and shown in table 8.

The `RETURN` statement provides immediate exit from a function or function block. For example —

```
IF IN > 1000 THEN
  (* input overflow *)
  OK := FALSE;
  RETURN;
END_IF;
Y := 10;
```

will return from the function block or function containing the `RETURN` statement when the `IN` variable has a value of more than 1000. In other words the statement `Y := 10` and any after it are not reached once the `RETURN` statement is executed. If the `RETURN` was in a function block body execution resumes at the statement that called the function block. If the `RETURN` was in a function then the function yields whatever value it was given before the return statement was executed (or the default initial value for the function's type if none was given).

### 7.3 Selection statements

Selection statements provide a way of selecting one or more groups of statements for execution, based on a specified condition which has the form of an expression of type `BOOL`. Examples are given in table 8.

Table 8: CDL language statements

Statement type	Example
Assignment (7.1)	A := B; CV := CV + 1; PT := t#300ms
Function block invocation and FB output usage (7.2)	XX( IN := 10, IN1 := 30 - YY / RR); Y := XX.OUT - 33;
RETURN (7.2)	RETURN;
IF (7.3.1)	IF X < 0.0 THEN Y := 10.1; ELSIF Y > 0.0 THEN Z := 11.01; ELSE F := 10.11; END_IF;
CASE (7.3.2)	CASE XX OF 1,5: W := 1; 2: W := 4; 3: W := 100; 4, 6..10: W := 1; ELSE ERROR := 1; END_CASE;
FOR (7.4.1)	FOR i := 1 TO 100 BY 2 DO IF WORDS[i] = 'KEY' THEN EXIT; END_IF; END_FOR FOR j := 10 TO 1 BY -1 DO X := X + 50; END_FOR;
WHILE (7.4.2)	J := 1; WHILE J < 100 AND WORDS[J] <> KEY DO J := J + 2; END_WHILE;
REPEAT (7.4.3)	J := -1; REPEAT J := J + 2; UNTIL J > 100 OR WORDS[j] = 'KEY' END_REPEAT;
EXIT (7.4.4)	EXIT;
Empty statement	;

### 7.3.1 If statements

Simple IF statements have the form

```
IF <boolean expression> THEN
  <group of statements>
END_IF;
```

The <group of statements> are executed if and only if the <boolean expression> evaluates to TRUE.

Alternatives can be supplied using ELSIF, i.e

```
IF <boolean expression> THEN
  <group of statements>
ELSIF <alternative boolean expression> THEN
  <alternative group of statements>
END_IF;
```

First the <boolean expression> is evaluated, if and only if it evaluates to TRUE the first <group of statements> is executed, if it does not evaluate to TRUE then if and only the <alternative boolean expression> evaluates to true the <alternative group of statements> are executed, otherwise execution continues after the IF statement.

Any number of ELSIF alternatives may be specified, for example

```
IF A = 1 THEN
  X := 2; Y := 10;
ELSIF A = 2 THEN
  X := 3; Y := 11;
ELSIF A = 3 THEN
  X := 4; Y := 15;
ELSIF A = 11 THEN
  X := 20; Y := 21;
END_IF;
```

Finally the ELSE clause can be used to specify a group of statements to be executed if conditions for the IF and any associated ELSIFs are false,

```
IF A = 1 THEN
  X := 2; Y := 10;
ELSIF A = 2 THEN
  X := 3; Y := 11;
ELSIF A = 3 THEN
  X := 4; Y := 15;
ELSIF A = 11 THEN
  X := 20; Y := 21;
ELSE
  X := -1; Y := 100;
END_IF;
```

In the above example for any values of A other than 1, 2, 3, and 11, the statements X := -1; Y := 100; are executed.

Note that IF statements can be used to provide conditional calls of function blocks e.g. —

```
IF A = 1 THEN
  FB( SP1 := 2; SP2 := 10);
ELSIF A = 2 THEN
  FB( SP1 := 20, SP2 := 50);
END_IF;
```

### 7.3.2 Case statements

CASE statements consist of a selector expression which is of integer type. The value of the expression is used to branch to a statement group. Each statement group is labelled by one or more integers, and/or a range of integer values. The labels must all be unique, i.e. there must be no overlap in the integer values supported. The statement group branched to is the one labelled by a value which is the same as the selector. If the value of the selector is not in any of the labels then all the statement groups are skipped and execution continues after the case statement, unless an ELSE clause has been used to specify a default group of statements to be executed.

Some case statements and the equivalent IF statement are given in the tables below.

```
CASE A OF
  1: X := 2; Y := 10;
  2: X := 3; Y := 11;
  3 : X := 4; Y := 15;
  11: X := 20; Y:= 21;
END_CASE
```

```
IF A = 1 THEN
  X := 2; Y := 10;
ELSIF A = 2 THEN
  X := 3; Y := 11;
ELSIF A = 3 THEN
  X := 4; Y := 15;
ELSIF A = 11 THEN
  X := 20; Y := 21;
END_IF;
```

Adding ELSE clauses gives

```
CASE A OF
  1: X := 2; Y := 10;
  2: X := 3; Y := 11;
  3 : X := 4; Y := 15;
  11: X := 20; Y:= 21;
ELSE
  X := -1; Y := 100;
END_CASE;
```

```
IF A = 1 THEN
  X := 2; Y := 10;
ELSIF A = 2 THEN
  X := 3; Y := 11;
ELSIF A = 3 THEN
  X := 4; Y := 15;
ELSIF A = 11 THEN
  X := 20; Y := 21;
ELSE
  X := -1; Y := 100;
END_IF;
```

A complicated example showing using labels which are ranges of integer values and lists of values is —

<pre> CASE A OF   20..23:      X := 2; Y := 10;   1,2,51 :    X := 3; Y := 11;   3,100,200..203 : X := 4; Y := 15;   9..11, 300..302,               400, 500: X := 20; Y:= 21; ELSE   X := -1; Y := 100; END_CASE; </pre>	<pre> IF A = 20 OR A = 21   OR A = 22 OR A = 23   THEN     X := 2; Y := 10;   ELSIF A = 2 OR A = 1     OR A = 51 THEN     X := 3; Y := 11;   ELSIF A = 3 OR A = 100     OR A = 200 OR A = 201     OR A = 202     OR A = 203 THEN     X := 4; Y := 15;   ELSIF A = 11 OR A = 10     OR A = 9 OR A = 300     OR A = 301 OR A = 302     OR A = 400 OR A = 500     THEN     X := 20; Y := 21;   ELSE     X := -1; Y := 100;   END_IF; </pre>
---	--

## 7.4 Iteration statements

Iteration statements are used to group a set of statements which are then executed repeatedly within the same block execution cycle.

### 7.4.1 For statements

A simple FOR statement has the form

```

FOR <loop variable> := <initial expression>
  TO <final value expression> DO
  <group of statements>
END_FOR;

```

The <loop variable> must be of integer type, and the <initial expression> and <final value expression> must be expressions of the same integer type. The <loop variable> is initialised to have the value of the <initial expression>. The <group of statements> are executed repeatedly. After each execution the <loop variable> is incremented by 1. Execution of the loop terminates when the <loop variable> reaches the value of the <final value expression>. It is an error to assign a value to the <loop variable> in the <group of statements>.

For example the following loop is a very slow way of adding 10 to variable X —

```

FOR I := 1 TO 10 DO
  X := X + 1;
END_FOR;

```

The <loop variable> can be used within the loop, typically for indexing into arrays. This loop assigns the values 1 to 10 to the first 10 elements of the one dimensional array AR.

```
FOR I := 1 TO 10 DO
  AR[I] := I;
END_FOR;
```

It is also possible to specify an increment expression —

```
FOR <loop variable> := <initial expression>
  TO <final value expression>
  BY <increment expression> DO
  <group of statements>
END_FOR;
```

The <increment expression> must be of the same integer type as the <loop variable>.

On each repetition of the loop the <loop variable> is incremented by the value of the <increment expression>. For example this loop

```
FOR I := 10 TO 1 BY -1 DO
  AR[I] := I;
END_FOR;
```

fills the one dimensional array AR with values 1 to 10, starting with element 10 and working backwards.

Note that the <final value expression> and the <increment expression> are evaluated once and once only before the loop is executed; if they contain variables and those variables are altered in the loop body it makes no difference to the number of times the loop is executed.

For example these two FOR loops are equivalent.

```
J := 20;
K := 2;
FOR I := 1 TO J + 1 BY K DO
  AR[I] := J;
  J := J + 1;
  K := K + J;
END_FOR;
```

```
J := 20;
K := 2;
FOR I := 1 TO 21 BY 2 DO
  AR[I] := J;
  J := J + 1;
  K := K + J;
END_FOR;
```

If the initial value, increment and final value expression are such that the loop variable never reaches the final value exactly, execution will stop on the cycle where the loop variable is greater than final value, if the increment expression is positive, and less than the final value, if the increment expression is negative. So the following loop

```
FOR I := 1 TO 2 BY 2
```

will execute once.

Of course it is possible to have nested loops, since a FOR loop is a statement and the loop body is a set of statements which can themselves be FOR loops. For example this loop fills a two dimensional 3 \* 4 array with the values 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12.

```
FOR I := 1 TO 3 DO
  FOR J := 1 TO 4 DO
    AR2[I,J] := J + 4 * (I - 1);
  END_FOR;
END_FOR;
```

### 7.4.2 While statements

WHILE statements allow a group of statements to be executed while a boolean expression yields the value TRUE —

```
WHILE <boolean-expression> DO
  <group of statements>
END_WHILE;
```

For example the following set of FOR and WHILE loops are equivalent

```
FOR I := 10 TO 1 BY -1 DO
  AR[I] := I;
END_FOR;
```

```
I := 10;
WHILE I > 0 DO
  AR[I] := I;
  I := I - 1;
END_WHILE;
```

```
FOR I := 1 TO 3 DO
  FOR J := 1 TO 4 DO
    AR2[I,J] := J + 4 * (I - 1);
  END_FOR;
END_FOR;
```

```
I := 1;
WHILE I < 4 DO
  J := 1;
  WHILE J < 5 DO
    AR2[I,J] := J + 4 * (I - 1);
    J := J + 1;
  END_WHILE;
  I := I + 1;
END_WHILE;
```

### 7.4.3 Repeat statements

REPEAT statements allow a group of statements to be executed until a boolean expression yields the value TRUE, —

```
REPEAT
  <group of statements>
UNTIL <boolean expression> END_REPEAT;
```

For example the following set of FOR and REPEAT loops are equivalent

```
FOR I := 10 TO 1 BY -1 DO
  AR[I] := I;
END_FOR;
```

```
I := 10;
REPEAT
  AR[I] := I;
  I := I - 1;
UNTIL I < 1 END_REPEAT;
```

```
FOR I := 1 TO 3 DO
  FOR J := 1 TO 4 DO
    AR2[I,J] := J + 4 * (I - 1);
  END_FOR;
END_FOR;
```

```
I := 1;
REPEAT
  J := 1;
  REPEAT
    AR2[I,J] := J + 4 * (I - 1);
    J := J + 1;
  UNTIL J > 4 END_REPEAT;
  I := I + 1;
UNTIL I > 3 END_REPEAT;
```

#### 7.4.4 Exit statement

The EXIT statement allows early termination of the execution of the group of statements in any REPEAT, FOR or WHILE loop. Control passes to the next statement after the first loop terminator. For example these loops are equivalent —

```
FOR I := 1 TO 5 DO
  AR[1] := I;
  IF I = 3 THEN
    EXIT;
  END_IF;
END_FOR;
```

```
FOR I := 1 TO 3 DO
  AR[1] := I;
END_FOR;
```

and in the following example if FLAG is 0 then SUM has the value 15 and if FLAG is 1 then SUM has the value 6 at the end of the loop.

```
SUM := 0;
FOR I := 1 TO 3 DO
  FOR J := 1 TO 2 DO
    IF FLAG THEN EXIT; END_IF;
    SUM := SUM + J;
  END_FOR;
  SUM := SUM + I;
END_FOR;
```



## 8 Sequential Function Charts

Sequential function charts are another way of defining the body of a program (section 6.3) function block (section 6) or service (section 11). A set of internal steps are active at any one time. Transitions can be made on boolean expressions between one or more steps to one or more steps. The boolean expressions may include the time a step has been active. Steps may be associated with an action. Actions are a set of CDL statements that may be another SFC or statements as described in section 7. Actions may be executed continuously while a step is active, or once when the step is active.

Sequential function charts consist of a series of step declarations, transition specifications, and action specifications. They have an equivalent graphical representation defined in IEC-1131-3.

Using SFCs it is possible to loop, to branch, to have parallel threads of execution, and for parallel threads of execution to rendezvous.

The model of SFC execution is that every time a block is executed whose body is a SFC the actions for active steps are executed, and then the transitions out of active steps are tested. If any evaluate to TRUE the next step or set of steps are marked as being active, and the current step or steps are marked as being inactive.

### 8.1 Steps

A step is declared by specifying a step name and an optional action together with a single action qualifier. For example

```
STEP STEP1: FILL(N); END_STEP
```

declares a step called STEP1 with an associated action FILL which has a qualifier N.

Every chart must have an initial step declared using the key word INITIAL\_STEP. For example —

```
INITIAL_STEP START; END_STEP
```

Note that in this example the step has no action, though one could have been specified.

If there is a unique step to which all paths through a SFC lead then this is known as an end step.

Associated with a step are some flags that may be used anywhere in any expression in the function block body as if they were outputs of a function block with the instance name of the step.

Products may supply additional flags for steps.

The flags are

- X The X flag is a BOOL which has value TRUE when the step is active, and FALSE when the step is inactive.
- T The T flag is a TIME which represents the time a step has been active for. It is reset to t#0 each time the step is deactivated<sup>1</sup>.
- QT The QT flag is a QTIME which represents the time a step has been active for. It is reset to qt#0 each time the step is deactivated<sup>1</sup>.
- F If the associated action is itself an SFC that has an end step this flag is TRUE if the action's SFC has reached the end step. Otherwise this flag is FALSE<sup>2</sup>.

Steps may be declared anywhere in the body of a block, service, or program, but they cannot be referenced before they are declared.

<sup>1</sup> Products will either support a QT flag or a T flag.

<sup>2</sup> This flag is only required if actions that have SFC bodies are supported (see section 8.5)

## 8.2 Transitions

Transitions represent the condition whereby one set of steps are inactivated, and another set of steps are activated.

Transitions are declared as follows —

```
TRANSITION <optional name> FROM < from set of steps> TO < to set of steps> :=
    < boolean expression> ; END_TRANSITION
```

The <optional name> may be omitted, if specified it gives a name for the transition. The < from set of steps> and the < to set of steps> are either single step names, or a comma separated list of step names. The < from set of steps> specifies the set of steps that are deactivated when the transition condition is TRUE, and the < to set of steps> the steps that are activated when the transition is true. Note that transitions are only evaluated if all of the < from set of steps> are active.

A simple transition between two steps could be —

```
TRANSITION FROM S1 TO S2 := READY; END_TRANSITION
```

When step S1 is active and the BOOL variable READY is TRUE step S2 will be activated, and S1 deactivated.

If the requirement was to wait in a step for 2 seconds and then move onto another step, the transition could be —

```
TRANSITION FROM S1 TO S2 := S1.T >= T#2s; END_TRANSITION
```

To create a loop where step S1 was active for 2 seconds, and then step S2 became active for 1 second, and then the loop began again, the following transition could be added —

```
TRANSITION FROM S2 TO S1 := S2.T >= T#1s; END_TRANSITION
```

There is nothing to stop a step looping to itself. This is really only useful if the step has an action that is executed once only when the step is activated, (rather than a continuously executed action), and the desire is to trigger the action again on a particular condition —

```
TRANSITION FROM S1 TO S1 := DO_IT_AGAIN ; END_TRANSITION
```

Simultaneous sequences happen when a transition goes to more than one step. For example the following transition branches from step S3 to three other steps when a variable TEMP has reached 100 —

```
TRANSITION FROM S3 TO (S4,S5,S6) := TEMP > 100.0 ; END_TRANSITION
```

In this example steps (S4,S5,S6) all become active when the condition TEMP > 100.0 is TRUE. This sets up three parallel branches of execution.

A rendezvous of parallel braches occurs when a transition occurs from one or more steps to one single step. All the steps “before” the transition (or the “predecessor” steps) must be active before the transition is tested —

```
TRANSITION FROM (S11,S15,S16) TO S7 := QUANTITY > 50 AND OP_IN ;
    END_TRANSITION
```

In this example step S7 becomes active (and steps (S11,S15,S16) are deactivated) if and only if \_\_\_\_\_.

1. Steps (S11,S15,S16) are active (so their .X flag is TRUE)
2. The transition condition QUANTITY > 50 AND OP\_IN is TRUE.

A way to picture SFC execution is to imagine it as a network of steps and transitions with active steps marked with a token. As the SFC evolves the token moves from one active step to another. On reaching a parallel branch the token splits into several tokens, one for each active branch. On the rendezvous of parallel branches the several tokens merge into one.

Once parallelism and branching is introduced there is the danger of having "multiple activations" of paths where a step that is already active is re-activated. For example

```
STEP S3: S3ACT(P) ; END_STEP
```

```
TRANSITION BAD FROM S1 TO (S2,S3) := 1; END_TRANSITION
```

```
TRANSITION FROM S2 TO S1 := 1; END_TRANSITION
```

As written the SFC will continuously loop round steps S1 and S2. However the transition from S1 also activates step S3. Suppose S3 is still active when S1 is active. The action S3ACT will be re-triggered when the transition BAD is evaluated, therefore it may be run more than once in one execution cycle.

If we add some steps to the example above,

```
STEP S3: S3ACT(P) ; END_STEP
```

```
TRANSITION BAD FROM S1 TO (S2,S3) := 1; END_TRANSITION
```

```
TRANSITION FROM S2 TO S1 := 1; END_TRANSITION
```

```
TRANSITION FROM S3 TO S4 := 1; END_TRANSITION
```

```
TRANSITION FROM S4 TO S5 := 1; END_TRANSITION
```

```
TRANSITION FROM S5 TO S6 := 1; END_TRANSITION
```

```
TRANSITION FROM S6 TO S7 := 1; END_TRANSITION
```

The set of active steps at any one time will look like —

```
S1
S2,S3
S1,S4
S2,S3,S5
S1,S4,S6
S2,S3,S5,S7
```

so that the parallel thread of steps S3,S4,S5,S6,S7 has several active steps in it at one. Generally this is an error, because there is an uncontrolled proliferation of tokens active.

Another possible error is to cause a "lock up" whereby an SFC by nature of its topology cannot make any progress. For example in the following SFC

TRANSITION FROM S1 TO S2 := ONE ; END\_STEP

TRANSITION FROM S1 TO S3 := TWO ; END\_STEP

TRANSITION FROM (S2,S3) TO S4 := THREE; END\_STEP

If two transitions out of one step are both TRUE at the same time, the last one textually specified takes precedence. So in this example, because S2 and S3 are mutually exclusive, the transition to S4 can never occur (since any rendezvous only occurs if all of the predecessor steps are active).

To try to reduce the risk of badly behaved SFCs, steps that have a transition to a rendezvous can have only one transition out of them. So the following is illegal —

TRANSITION FROM (S2,S3) TO S4 := 1; END\_TRANSITION

(\* THIS IS ILLEGAL BECAUSE OF ABOVE RENDEZVOUS \*)

TRANSITION FROM S2 TO S5 := 1; END\_TRANSITION

### 8.3 Actions

Actions are CDL statements, including SFCs, that are controlled by step activation and deactivation.

An action has the following format —

```
ACTION <action name>:
<action body, either SFC or ordinary statements>
END_ACTION
```

Actions are associated with a step. When a step is declared an <action name> and qualifier are provided, (see 8.1), for example in —

```
STEP STEP1: TRIGGER(P); END_STEP
```

the TRIGGER action has a P qualifier. Action qualifiers determine how actions are executed. The following qualifiers are supported —

N is the “continuous qualifier”. Actions with this qualifier are executed once on each execution cycle while the associated step is active. When the step is deactivated the action is executed one last time — the code in the action can distinguish this by testing the .X flag of the step.

P is the “one-shot” qualifier. Actions with this qualifier are executed once only each time the step they are associated with changes from inactive to active.

As a simple example consider a program that has to control a chemical process. There are three tanks, each associated controlled with a tank function block. Two liquids are put into the first two tanks and heated until they reach a temperature of 100. The second tank is then mixed for 2 minutes.

The tanks are then drained into the third tank and mixed for 5 minutes. This tank is then drained, and the program goes back to start.

The tanks are represented by function blocks with a REAL temperature output, a mixer on/off input, and two fill and heat BOOL command inputs.

```

PROGRAM mix
VAR
    t1,t2,t3: tank;
END_VAR

INITIAL_STEP mixprog: dotanks_act(N); END_STEP

STEP dotanks: dotanks_act(N); END_STEP

STEP start: init(P); END_STEP

STEP mon1: mon1_act(N); END_STEP

STEP mon2: mon2_act(N); END_STEP

STEP mix2: mix2act(N) ; END_STEP

STEP fill3: fillact(P) ; END_STEP

TRANSITION FROM mixprog TO (start,dotanks) := TRUE; END_TRANSITION

TRANSITION FROM start TO (mon1,mon2) := TRUE; END_TRANSITION

TRANSITION FROM mon2 TO mix2 := t2.temp >= 100.0 ; END_TRANSITION

TRANSITION FROM (mon1,mix2) TO fill3
    := ( t1.temp >= 100.0 ) AND (mix2.T >= t#120s ) ;
END_TRANSITION

TRANSITION FROM fill3 TO start := fill3.T >= t#300s ; END_TRANSITION

ACTION dotanks_act:
(* Execute the tank function blocks continuously *)
    t3();
    t1();
    t2();
END_ACTION

ACTION init:
(* empty tank3, fill tanks 1 and 2, and commence
    heating *)
    t3(fill:= FALSE, mix := FALSE);
    t1(fill := TRUE, heat := TRUE);
    t2(fill := TRUE, heat := TRUE);
END_ACTION

ACTION mix2act:
(* Mix tank2. On exit from the action switch mixing off *)
    IF mix2.X THEN
        t2(mix := TRUE);
    ELSE
        t2( mix := FALSE);
    END_IF;
END_ACTION

ACTION fillact:
(* Fill tank3, empty tanks 1 and 2 and switch off heating.
    Start mixing tank3 *)
    t1( heat := FALSE, fill := FALSE);
    t2( heat := FALSE, fill := FALSE);
    t3(fill := TRUE, mix := TRUE);

```

```

END_ACTION

ACTION mon1_act:
  (* Monitor tank 1 *)
END_ACTION

ACTION mon2_act:
  (* Monitor tank 2 *)
END_ACTION

END_PROGRAM

```

The SFC has two parallel branches. One, the dotanks branch is used to make sure that the tank function blocks are always being executed to do their control.

The other branch actually does the sequencing. It splits into two parallel steps mon1 and mon2 which monitor the tanks until they reach the required temperature. The mon2 goes on to the mix2 step when the second tank is ready for mixing. There is then a rendezvous between mon1 and mix2 when tank1 is at the correct temperature, and tank2 has been mixed, where they go on to step fill3. In this step tank3 is filled, and mixing in tank3 begins. After 5 minutes the program goes back to the start step.

Most of the actions that change the modes of tanks are one-shot P actions. However, a continuous action `mix2act` is used both to switch on mixing of tank2 and to switch it off when the step is deactivated. An alternative would be to have used two one-shot actions, one to switch mixing on, and the next to switch it off, or even more simply to switch mixing off in the `fillact` action.

## 8.4 SFC execution

A SFC is executed on every call of the block it is defined in. SFC execution consists of the following —

1. The actions for any active step are executed according to their qualifiers. So if a step with a one-shot action has just become active on this execution cycle, the action is executed, otherwise it is not. If a step has a continuous action, then the action is executed.
2. The .T or .QT step output flags that say how long a step has been running are updated.
3. The transitions out of any active step are tested, except if a transition is a rendezvous, in which case all predecessor steps must be active for the transition to be tested. If two transitions out of a step are both TRUE then the last one specified takes precedence.
4. Steps that have TRUE transitions out of them are deactivated. Their .X flag is set to false.
5. Any of these steps that have continuous N actions have the actions executed one last time.
6. Steps that have TRUE transitions into them are marked as being active, for the next time round the execution cycle.

This process is repeated every time the block a SFC is defined in is itself invoked.

## 8.5 SFC actions

Action bodies may be either ordinary statements as described in section 7, or they may themselves be another SFC, with its own initial step.

Normally action bodies can be shared by multiple steps (i.e an action can be associated with any number of steps), but if the body is a SFC the action may only be used by one and only one step.

An action that is a SFC can have either a **N** qualifier or another qualifier, the **A** qualifier. These qualifiers affect the way the SFC action is terminated when the associated step is deactivated.

If the action has a **N** qualifier then the transition condition out of the step is not tested until the SFC in the action has reached its end step (a unique step to which all paths through a SFC lead). If there is no end step then the step will never be deactivated.

Consider —

```
STEP usesfc: sfcact(N); END_STEP

TRANSITION never FROM usesfc TO next := TRUE; END_TRANSITION

ACTION sfcact:
  INITIAL_STEP s; END_STEP
  STEP z; END_STEP
  STEP y; END_STEP
  TRANSITION FROM s TO (y,z) := TRUE; END_TRANSITION
END_ACTION
```

`sfcact` has a **N** qualifer and a SFC with no unique end step, so the transition `never` will never occur.

If an end step is reached, then the SFC will halt having executed any action associated with the end step once. (and once only). However next time the action is activated (i.e when the controlling step is re-activated) the SFC will start at its initial step.

The `.F` flag of a step whose action is a SFC is `TRUE` when the SFC reaches its end step.

If a SFC action has a **A** qualifier (known as “abortable”) then any transitions out of the step are tested regardless of the state of the SFC in the action body. If a transition is `TRUE` then

- All active steps in the action body are deactivated and their `.X` flag is set to `FALSE`.
- Any actions associated with a step via a **N** qualifier are executed one extra time on deactivation (in the same way that any **N** action is on deactivation of a step).
- The SFC is re-initialised so that the initial step is the first one executed next time the action is used.

There is an equivalence between SFC actions with **N** and **A** qualifiers. Any SFC action with a **N** qualifier can be written in terms of an **A** qualifier and a test on the `.F` finished flag. These fragments of CDL are equivalent —

STEP s1: sact(N); END_STEP	STEP s1: sact(A); END_STEP
TRANSITION FROM s1 TO s2 := 1; END_TRANSITION	TRANSITION FROM s1 TO s2 := s1.F; END_TRANSITION
ACTION sact:	ACTION sact:
INITIAL_STEP a1; END_STEP	INITIAL_STEP a1; END_STEP
STEP a2; END_STEP	STEP a2; END_STEP
TRANSITION FROM a1 TO a2 := 1; END_TRANSITION	TRANSITION FROM a1 TO a2 := 1; END_TRANSITION
END_ACTION	END_ACTION

By using the different varieties of actions various effects can be achieved —

1. Using the N qualifier and a SFC action ensures that the top-level step remains active until all the steps in the sub SFC are complete.
2. Using the A qualifier allows aborting a sub SFC cleanly when a condition occurs that deactivates the top level step.
3. Calling a function block that has a SFC body from within a continuous action means —
  - When the controlling step is deactivated the sub SFC is no longer executed.
  - When the controlling step is re-activated the sub SFC restarts from where it was stopped.
  - A separate user supplied input to the function block must be provided to explicitly re-initialise the SFC.

## 9 References

Every Function Block or Program can have a reference declaration section. All data specified in this section is remote data, where remote means that it has its defining definition somewhere other than in the current block, though it may well be in the same RESOURCE or indeed TASK. The reference section is denoted by the keyword REFERENCE appearing after the keyword VAR. Note therefore that references are internal to the block they are defined in.

For example

```
PROGRAM ex1
VAR
  writeflag: BOOL;
  id: STRING;
END_VAR
VAR REFERENCE
  remflag: BOOL;
  remid: STRING;
END_VAR
```



has a reference section which contains two remote objects `remflag` and `remid`.

Any object may be declared to be a reference, from simple variables of any type, to arrays of any type ( except arrays of function blocks ), to function blocks.

References have a set of “properties” which are predefined built in variables. Properties may be assigned or read or both. Properties cannot be wired to but they can have initial values. Properties are used to control and monitor the reading and writing of data via the reference.

The first stage in accessing remote data is specifying where it is and then matching it to the local data. If this operation is successful then the remote data may be read and/or written.

## 9.1 Specifying the Remote Data Objects

A VAR REFERENCE has an associated string, the “ref” string. This is set by assigning it from within the ST PROGRAM, for example

```
remflag~ref := 'Res1:pid1.in';
```

or at cold start, for example

```
VAR REFERENCE
  remflag: BOOL { ref := 'Res1:pid1.in' };
END_VAR
```

The `~` indicates that the next name is a “property” of a VAR REFERENCE object. Property names are predefined, and the `ref` property is the reference string. Any ST string or ST string expression may be assigned to it.

The `currref` property when read returns the last set reference string.

### 9.1.1 Simple types

For a simple VAR REFERENCE, that is one which is a simple built in ST type (e.g DINT, BOOL, LREAL) the reference string must specify the full hierarchic path to the object, prefixed by an optional RESOURCE name and “.”. The syntax is

```
simple_ref_string ::= [ resource_name ] ':' name { '.' name }
```

(using the usual BNF notation where `[]` means an option, and `{ }` means zero or more of the enclosed). `name` is any valid ST name.

The `resource_name` is the name of a remote RESOURCE. If omitted the reference is to something in the local RESOURCE. The list of names separated by `.` is the full path to the remote object. So in the above example `Res1` is the name of the remote RESOURCE, `pid1` is a block in the remote RESOURCE which contains a variable `in`.

When a reference string is assigned the RESOURCE will query the specified remote RESOURCE for information about the specified object. The information returned is

- The addressability. Is it possible to address this object?

- The type of the remote object (DINT, LREAL etc.)
- The mode of the remote object (input, output etc.)
- The size of the remote object, which will be 1 for simple types and the total number of elements for an array type.
- A fast address for the remote object.
- The TASK that owns the remote object.
- Any write protection.

In order for reads and writes to be performed, the remote data must be addressable. In addition the type and size must match the type and size of the local VAR REFERENCE.

For simple types (i.e VAR REFERENCES that are not blocks) the mode of the remote object may be anything (i.e local internals match remote internals, in-outs, inputs or outputs).

For complex types (i.e VAR REFERENCES that are blocks) the mode of the remote object must match exactly, except that remote internals may match local inputs, in-outs or outputs.

### 9.1.2 Arrays

For arrays of simple variables, as well as the type and mode matching, the total number of elements in the remote object must match the number in the local object. So, for example a remote 2 by 10 array would match a local 10 by 2 array. In general a remote array with 6 dimensions  $i_1, i_2, i_3, i_4, i_5, i_6$  matches a local one with dimensions  $j_1, j_2, j_3, j_4, j_5, j_6$  provided  $i_1 * i_2 * i_3 * i_4 * i_5 * i_6 = j_1 * j_2 * j_3 * j_4 * j_5 * j_6$ . Local data at position  $x_1, x_2, x_3, x_4, x_5, x_6$  would be the remote data at position  $y_1, y_2, y_3, y_4, y_5, y_6$  if the same position has been specified when the array is "flattened" into a one-dimensional array.

It is also possible to match to single elements of an array, or to the whole of a sub-array, in exactly the same way as for ordinary array assignment.

For example given a remote array declared as

```
array: ARRAY[1..10,1..10] OF DINT;
```

and the local declaration

VAR REFERENCE

```
matchall: ARRAY[1..10,1..10] OF DINT;  
matchpart: ARRAY [1..10] OF DINT;  
matchele: DINT;
```

then the following would be valid.

```
matchall~ref := 'Remote:prog.array'  
matchpart~ref := 'Remote:prog.array[2]'  
matchele~ref := 'Remote:prog.array[1,2]'
```

So it is possible in the reference string to index into remote arrays, and have a successful match provided the size of the local object matches.

Note that a local array object can only have a single reference string (not an array of them).

### 9.1.3 Function Blocks

A VAR REFERENCE can also be a function block. The reference string then specifies one or more remote data objects that are matched to the inputs, outputs and in-outs of the local object. In the simplest case where only one remote object is specified, the remote object must be a block with parameters which match the local object's parameters in name, mode, type and size except that a remote internal may match a local input, output, in-out or internal. In other words each parameter of the remote object is individually matched to the local object's parameters by name as if it were a simple type. (The remote block may have extra parameters that are not matched). The local block is then an image of (possibly part of) the remote block's data.

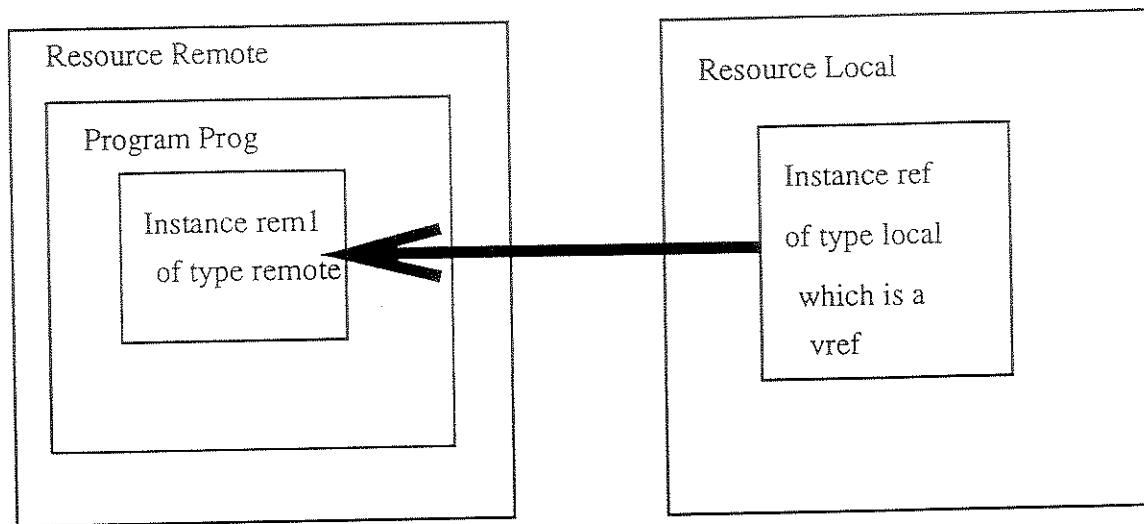


Figure 2: A reference to a remote block

The diagram in figure 2 should help understanding of the example shown below. Given a remote block such as

```
FUNCTION_BLOCK remote
VAR_INPUT
    in1:LREAL;
    in2: ARRAY[1..10] OF DINT;
END_VAR
VAR_OUTPUT
    out1: BOOL;
    ignored: BOOL;
END_VAR
```

which was instantiated in a RESOURCE called Remote in a PROGRAM called prog as block instance rem1. and a local block definition of the form

```
FUNCTION_BLOCK local
VAR_INPUT
    in1:LREAL;
    in2: ARRAY[1..10] OF DINT;
END_VAR
VAR_OUTPUT
    out1: BOOL;
END_VAR
```

then the following VAR REFERENCE

## REFERENCES

---

### VAR REFERENCE

```
ref:local;
```

### END\_VAR

```
ref`ref := 'Remote:prog.rem1';
```

would match the in1, in2 and out1 parameters, and would enable data to be exchanged via those parameters.

Of course by instantiating a local instance of remote all of its visible parameters may be matched.

It is also possible to specify a list of remote objects that are to be matched to a local block, by using a special syntax in the reference string. Fully hierarchic names can be put in a comma separated lists, or as a shorthand '{' and '}' are used to bracket comma separated lists of names which are then all taken to be relative to the previous hierarchic name. For example the string

```
a{b,x.e,f{j,k,l{m,n}}}
```

expands to the names

```
a.b, a.x.e, a.f.j, a.f.k, a.f.l.m, a.f.l.n
```

A local parameter must be assigned to each name in the resulting expanded list, for example

```
a{ loc1 := b, c { loc2 := d, loc3 := e}}
```

means that the local parameter loc1 is matched to a.b, loc2 to a.c.d and loc3 to a.c.e.

The full syntax of the reference string is

```
ref_string ::= simple_ref_string | complex_ref_string
```

```
simple_ref_string ::= [ resource_name ] ':' name { '.' name }
```

```
complex_ref_string ::= [ resource_name ] ':' [ primary_name ]  
                      '{' alternate_names_list '}'
```

```
alternate_names_list ::= alternate_names { , alternate_names_list }
```

```
alternate_names ::= [ hierarchic_name ] '{' alternate_names_list '}' |  
                   final_name
```

```
final_name ::= local_parameter_name ':' hierarchic_name
```

```
primary_name ::= name { '.' name }
```

```
hierarchic_name ::= name { '.' name }
```

```
local_parameter_name ::= name
```

```
resource_name ::= name
```

The `primary_name` is the name relative to which all the following list of names is specified. If absent the following list of names is taken relative to the remote RESOURCE as a whole, (i.e the list of names must contain the full path to the object). The { and } notation brackets a comma separated list of hierarchic names. Any name may itself contain a list of sub-objects using the { and } notation. At the bottom level the `local_parameter_name` specifies the parameter of the local VAR REFERENCE that is to be matched to the remote name. Any parameters of the local VAR REFERENCE that are not explicitly assigned remote objects are matched to parameters of the same name in the `primary_name`; if the latter is absent this is an error.

Duplicate `local_parameter_names` are not allowed, but duplicate remote names are, so it is possible to match one remote variable to two or more local variables.

For example

```
VAR REFERENCE
  RP : RemPID;
END_VAR
  RemPID^ref := 'R1:a{ Sp := b,c{ Pv := d, Op := e}}'
```

means that `R1:a.b` must match `RP.Sp`, `R1:a.c.d` must match `RP.Pv` and `R1:a.c.e` must match `RP.Op`. If `RP` has an extra parameter `X` then it is matched to `R1.a.X`.

#### 9.1.4 Services

A VAR REFERENCE may also contain services (see section 11)

## 9.2 Reading the Remote Information

When a reference string is assigned to a VAR REFERENCE, the RESOURCE will work out the names of the remote objects it needs to match to the local, and send (in one message) these to the remote RESOURCE. Various errors may then occur. These may be found by examining the VAR REFERENCE's status property. for example

```
IF ref^status > 1 THEN
  (* an error of some sort *)
END_IF
```

The errors associated with matching are

- A syntax or other error in the reference string
- The remote RESOURCE is not reachable.
- One or more of the remote objects do not exist.
- The remote objects are owned by more than one TASK, and so cannot be made into one reference.
- The remote objects do not match the local ones according to the rules specified above.

Once a match has been completed successfully a single read is issued to ensure that the local copy of data contains something intelligent.

This operation is known as "reading a template".

### 9.3 Reading Remote Data

Once a reference has been matched to the remote objects, data may be read. The scan property sets the scan rate in milliseconds for the remote data. For example

```
ref^scan := t#10s;
```

or else at cold start

```
VAR REFERENCE
  ref : local { scan := t#10s }
END_VAR
```

sets the scan rate for the reference `ref` to be once every 10 milliseconds. This means that every 10 milliseconds the RESOURCE will send a read message to the remote RESOURCE specified in the reference string to read all of the remote data. When the reply comes back the local image of the remote data is updated. A scan rate of zero means no reads are performed.

If, however, no reply is received by the time the next scan is due, no timeout occurs, and no message is resent. (Timeouts are handled separately with a separate global value). Thus specifying a very fast scan time means that the data will be read as fast as possible, being limited by the rate at which the remote TASK responds and the local TASK sends messages.

A scan is performed on each TASK cycle where "Time now  $\geq$  Time of last scan + Scan time" (a scan time of  $0 = \infty$ ). Therefore to perform a single-shot read, the scan time should be changed from 0 to a value  $\leq$  the TASK cycle time and then reset to 0 on the next TASK cycle.

A property called `currscan` can be used to read the last set scan rate.

For blocks all data is read (i.e. all matched outputs, inputs and in-outs) and placed in the local VAR REFERENCE.

Accessing the data of the VAR REFERENCE from CDL always returns the last data read.

The property `newData` may be read to determine if any new data has been read since the last time this property was read. The reading of the property is a destructive operation.

### 9.4 Writing Remote Data

A write to the remote object is triggered by either assigning to it (if it is a simple variable) or calling it (if it is a block) passing it input parameters as usual. A write message is generated, unless a write message is already outstanding. In this case a flag is set to indicate that another write is required when the previous write is acknowledged. In this way the latest local value is always written to the remote object. Note that writes can not occur faster than the rate at which the remote TASK acknowledges them, i.e. every assignment does not guarantee a write.

If the VAR REFERENCE is a simple variable or array all the local data is sent (even if only part of an array has been written).

If the VAR REFERENCE is a function block the inputs and in-outs of the block that were assigned a value in the call of the block are sent (the same rules are followed for arrays as in the previous paragraph). After the write (in fact with the response to the write) the values of the remote object are read, so that the local copy is automatically updated with the latest values of the remote data.<sup>5</sup> This means that to perform control

<sup>5</sup>Note that this does not mean response to the write waits for the remote block to execute, merely that a read is done *immediately* after the write and the data sent back as part of the response to the write.

across the network at a particular rate a write should be issued at that rate, and a read of the data not being written will automatically occur on any write, hence at the same rate. For example, suppose a block that was being executed every second had the following declarations and wiring in it —

```
VAR REFERENCE
  RemVal: Analogue( Op := locpid.Op );

END_VAR

VAR
  LocPid: PID ( Pv := RemVal.Process_Val );
END_VAR

  LocPid();
  RemVal();
```

This would once a second execute the local PID, send the computed output value to the remote Analogue block, and then receive a new value of of the .Process\_Val output.

In addition to the above constraints no data that is reported as write protected at the remote RESOURCE is sent in a write message.

It is possible to mix writing with scanning on a single VAR REFERENCE however it should be noted that if a scan is currently in progress then the write will not be sent until the read has completed.<sup>6</sup>

It is possible to prevent the writing of data by setting the dontWrite property. Once set no data will be written to the remote objects until the property is cleared. Once cleared the write will occur at the end of the next task cycle, this is unlike a normal write where the write is issued at the time of the assignment. While the dontWrite property is set any incoming reads (from setting non-zero scan rate) will be discarded as they would overwrite the local data. This is particularly useful when writing to elements of arrays, to prevent the whole array being sent across the network for each write of an element.

## 9.5 Reference Time stamp

A var reference has a time stamp giving the time at which data was last refreshed. The time is available via two properties, the first TimeStamp gives the time of last update in DATE\_AND\_TIME to second accuracy. Another QTIME property QTimeStamp gives the additional number of micro seconds to be added to TimeStamp.

## 9.6 References to References

It is possible for a Reference to be a Reference to a remote object which may itself be a Reference to some other object. Of course it is then possible for there to be a cycle in a chain of References, so that following the chain leads back at some point to an object already in the chain.

The guiding principles are that

- At the completion of a read/write any data in a var reference is always a coherent set of data from some time in the history of the ultimate destination.
- Writes and service calls always propagate through to the ultimate destination, if possible.
- Any cycles will be detected and an error reported.

<sup>6</sup>There is actually no such things as a read or write message — there is only a read-write message that specifies some data to write and some to read. Read messages are read-write messages with an empty set of data to write.

Various possibilities exist —

1. The remote object is not a var reference
2. The remote object is a single var reference
3. The remote object is a combination of var references
4. The remote object is a mixture of a var reference and some other objects

This is known as the “resolution” of a var reference. The actual resolution via a property of var references, resolution. The values are as in the above list. Value 0 means that the template has not yet been read, or there has been a mismatch.

Note that since reference strings may change dynamically these states may also change at run time.

In case 2 the system works transparently through the remote object. For a read if the remote data is coherent and more recent than the last data obtained then it is read directly, otherwise a read through occurs, (i.e the read is forwarded on). This process occurs at every intermediate var reference until the ultimate destination is reached.

One exception to this is that on the first read of data, or any change of of the scan rate, there will always be a read through to the ultimate destination.

If the remote data is not up to date, a read through occurs.

For writes and service requests (section 11), the request is always forwarded on to the ultimate destination.

Case 1 above has already been explained.

Cases 3 and 4 are regarded as errors. These errors are reported in the status, readStatus, and writeStatus properties ( see section 9.7). Note because the reference strings of the remote objects can change at run time the error is “unsuccessful operation”, not “failed”.

### 9.6.1 Adjustment of scan rates

If in case 2 a reference does a read request and an intermediate reference detects that its local data is not recent enough, as noted, a “read through” occurs. The intermediate reference will also adjust its local scan rate to try to cache the data in anticipation of the next read request, so that a read through is avoided. The scan rate will be decayed over time, though it will never fall below that set by the local CDL.

The intention is to provide automatic caching of remote data in intermediate nodes, that adjusts itself according to the data request rate.

There is a BOOL property called `~propertyProtect` that if set prevents this adjustment occurring to a reference. This defaults to TRUE.



## 9.7 Examining the state of a reference

The `status` property of type `DINT` is available to determine the current state of a reference. The `status` property has the values and meaning shown in the following table —

State	Value	Meaning
OK	0	Last operation succeeded
InProgress	1	Read, write or read template in progress
ParseFail	2	A reference string had the wrong syntax
ResolveFail	3	Local names in the reference string did not match to the local object, or were duplicated
NoResources	4	The task has no buffers left to send messages with, or the expanded reference string is too long
TemplateMismatch	5	The read template did not match the local one
Unreachable	6	The router was not reachable
BadStatus	7	Either a read template specified non-existent objects, or read failed to read the data at the remote end (though the message arrived), or a write failed to write (for example if some block outputs were being written)
NonUniqueOwner	8	In a read template the remote objects belong to more than one remote task
SystemError	9	This should not be seen, if seen there is an internal error in the Resource
Cyclic	10	The reference results in a cycle of references that point back to this one
NotCoherent	11	The reference points to a set of other references and objects, possibly owned by more than one task, or in error

## 9.8 Timeouts and failures

All read, write and read-template operations have built in timeouts.

On a timeout the Resource will automatically try to re-read the template of the remote objects again. This is to ensure consistency if the timeout was because a remote Resource was reloaded.

In addition each message contains a checksum for the remote Resource. This is stored in the local reference, and if a read or write returns with a different checksum to the local one then the remote Resource has been reloaded between the read or write. Again the remote template will be re-read.

The automatic re-read of templates in these circumstances means that a timeout is not visible to the user via the `status` property, so two other properties are available to examine the success of read or write messages. These are `readstatus` and `writestatus` `DINT`s. These both have the following states

State	Value	Meaning
OK	0	Last operation succeeded
InProgress	1	Read, write in progress
Failed	2	The last read or write failed
Undefined	3	No read or write issued with this ref string
Unsuccessful	4	There is a problem with references to references, status will be 10 or 11

Reads and writes are essentially asynchronous. To simulate synchronous operations a Sequential Function Chart may be used, which tests the read/write status in a transition to determine when an operation completed.

A synchronous write may be performed by writing the remote data in a step and transitioning out of the step when the reference has OK writestatus.

A synchronous read may be performed by setting the scan property from 0 to a positive large value in a step (so that only one read will be done), transitioning out of the step when readstatus is OK, and setting the scan property to zero.

## 9.9 Summary of Properties

The following table summarises the properties that a reference has, (see also section 11, page 68).

Property	ST Type	Mode	Meaning
ref	STRING	Input	Specifies the object(s) referred to (section 9.1)
status	DINT	Output	Monitor any errors using a reference (section 9.7)
writestatus	DINT	Output	Monitor success or failure of the last write operation (section 9.7)
readstatus	DINT	Output	Monitor success or failure of the last read operation (section 9.7)
curref	STRING	Output	The current reference string (section 9.1)
scan	TIME	Input	Used to set the scan rate (section 9.3)
propertyProtect	BOOL	Input	Prevent automatic scan rate adjustment (section 9.6.1)
TimeStamp	TIME	Output	Last time of data update (section 9.5)
QTimeStamp	QTIME	Output	Last time of data update (section 9.5)
resolution	Usint	Output	References to references (section 9.6)
currscan	TIME	Output	The current scan rate (section 9.3)

## 9.10 Addressability and Write Protection

It is possible to control whether any instanced variable or function block is writable or addressable via a var reference.

**Write Protection** Using the same style of syntax as for properties, it is possible to write protect a variable — for example

```
VAR
  x : DINT { writeProtect := 1 };
END_VAR
```

This means that x cannot be written to via a var reference.

writeProtect is a BOOL attribute (see section 12), and may be initialised with any constant boolean expression.

It is not possible to write protect a block instance, only simple variables may be protected (though of course they may be inputs, outputs or in-outs of blocks).

It is an error to write protect a var reference.

**Addressability** Similarly it is possible to make any object invisible using a BOOL `addressable` attribute —

```
VAR
  x : DINT { addressable := 0 };
  Y : FBTYPE { addressable := 0 };
END_VAR
```

In the case of simple variables setting addressability to FALSE means that the variable cannot be addressed via a var reference.

In the case of a block setting addressability to FALSE means that the block and none of the blocks it instances (to any level below it) can be addressed via a var reference. It is an error to make a block that contains var references unaddressable.

By default internal variables are not addressable, and input, output and in-out variables of a block are addressable. `addressable` may be used to override the defaults.

## 10 Resources

A Resource declaration usually represents a node on a network. Some nodes may be able to run more than one Resource at a time. Some nodes may provide a Resource interface to some other lower level network. A Resource instances a set of blocks (Programs or Function Blocks) and a set of Tasks that execute a block.

A simple example of a Resource declaration is —

```
RESOURCE tst1 ON controller1

  TASK fast ( INTERVAL := t#3ms );
  TASK slow ( INTERVAL := t#100ms );

  PROGRAM digitals WITH fast : dig;

  PROGRAM analogs WITH slow: control;

END_RESOURCE
```

The RESOURCE <resource name> ON <controller type> specifies a resource name and a target machine type.

The TASK <task name> ( <task parameter assignments> ) declares a task and initialises the task parameters. Tasks always have an INTERVAL parameter. In the example there are two tasks specified, `fast` and `slow`. These are executed every 3 and 100 milliseconds respectively. More precisely these will execute the blocks that they control, once and once only in every 3 and 100 millisecond interval. (Any failure to do this is known as an overrun and is an error.)

It is possible to assign a task to a specific processor in a multi-processor system by using the syntax

```
TASK <task name> ( <task parameter assignments> ) ON <processor name>
```

<processor name> is a pre-declared identifier for the processor. The set of supported processor names, if any, is a product specific issue. For example

## RESOURCES

---

```
TASK fast ( INTERVAL := t#1ms ) ON DMC_1;  
TASK slow ( INTERVAL := t#1s ) ON C68000;
```

A set of function block or program instances may then be declared, with the general syntax

```
<block class> <block instance name> WITH <task name> :  
    <block type> <optional wiring and initialisation>
```

<block class> is either PROGRAM or FUNCTION\_BLOCK.

<task name> is the name of a previously declared task that controls execution of the block with instance name <block instance name>.

<block type> is the type of the block (e.g PID).

<optional wiring and initialisation> may be omitted — if given it assigns values to the inputs of the block using the usual parenthesised list of parameters syntax as in a function block call.

In the example above two programs are declared, digitals of type dig and analogs of type control. These are run under control of fast and slow respectively.

In the above example there is no communication between the two programs, and there is no initialisation of any data in the two programs. This can be achieved by using wiring (section 6.2) and cold start syntax (section 6.1). There are two restrictions, however —

1. Whereas general cold start values support arbitrary expressions containing variables, cold start values at the Resource level must be constant expressions.
2. Whereas general purpose wiring supports expressions, at the Resource level only simple assignments are allowed.

An example of a Resource with inter-block wiring and cold start initialisation is —

```
RESOURCE tst1 ON controller1  
  
    TASK fast ( INTERVAL := t#3ms );  
    TASK slow ( INTERVAL := t#100ms );  
  
    PROGRAM digitals WITH fast : dig ( go := analog1.go );  
  
    FUNCTION_BLOCK analog1 WITH slow:  
        control1 ( in := analog1.out, in := 2.0, en := 1 );  
  
    FUNCTION_BLOCK analog2 WITH slow:  
        control1 ( in := analog2.out, in := 1.0, en := 0 );  
  
END_RESOURCE
```

Array inputs to resource level blocks can be initialised using either indexing or array initialisation syntax —

```
FUNCTION_BLOCK arrs WITH slow: floats  
    ( in1[1] := 1.0, in1[2] := 2.0, in1[3] := 1.1,  
      in2 := { 1.1, 2.1, 3.1 } );
```

Array inputs or outputs can be wired, with the usual restrictions on array to array assignment (see page 18).

## 10.1 Tasks and task execution model

The full set of task parameters supported by a CDL node is a product specific issue. Task parameters may be wired to block parameters in the same way as any block parameter.

The INTERVAL parameter should always be supported. It means that the task will be run once during every INTERVAL seconds. INTERVAL may be of type TIME or QTIME, again this is a product specific issue.

Other parameters, such as PRIORITY which assigns a UDINT priority to a task in a pre-emptive task execution system may be supported.

The edge triggered BOOL parameter SINGLE may be supported, if so it denotes that the task is event driven and will run when SINGLE goes from FALSE to TRUE. INTERVAL and SINGLE are mutually exclusive (i.e if SINGLE is used INTERVAL is zero) and should not be used together with the same task.

## 10.2 Remote blocks

It is possible to specify that blocks at the Resource level are VAR REFERENCES to blocks in another resource. (see section 9). The keyword REFERENCE replaces the PROGRAM or FUNCTION\_BLOCK keywords. The properties of the reference are specified in the usual place, after the type of the block. The general syntax is —

```
REFERENCE <block instance name> WITH
  <task name> : <block type> <optional wiring>
  { <property assignments> }
```

A real example is

```
RESOURCE tst1 ON controller1

  TASK slow ( INTERVAL := t#10s );

  FUNCTION_BLOCK analog1 WITH slow:
    control1 ( in ::= remote1.Op, in := 2.0);

  REFERENCE remote1 WITH slow : PID ( Pv ::= control1.out )
  { ref := 'R2:A.PID' }

END_RESOURCE
```

Here every ten seconds (the interval of task slow), a new process value is sent to the remote PID, and a new value for the control1 input is received from the remote PID.

## 11 Services

Services are an extension to the IEC1131-3 standard designed to provide remote procedure call and synchronisation of tasks across the network, and extra methods for function blocks resulting in easier to understand and more modular blocks.

Services make it significantly easier to write distributed ST applications.

## 11.1 Service declaration

Services are extra “methods” on function blocks. They are declared at the end of the normal function block variable section. Services have input declarations, output declarations and internal declarations, as well as a service body. Formally the syntax for a function block declaration is

```
<function block> ::= <function block variables>
                    [ < service declarations>]
                    <function block body>

<service declarations> ::= { SERVICE <service name>
                             <service inputs>
                             <service outputs>
                             [<service internals>]
                             END_SERVICE }

<service inputs> ::= <function block input declarations>
<service outputs> ::= <function block output declarations>
<service body> ::= <function block body>
```

Note that services do not have in-out variables.

A <service body> can access any variable or call any blocks declared in the function block (but not in another service) in addition to its own inputs and outputs. The rules for access are the same as for a function block body, so any inputs are read only, outputs and internals are read-write. The names declared in <service inputs> and <service outputs> must not conflict with names declared in <function block variables>, i.e the service names are in the same scope as the function block variable names. However once the service body has been compiled the service internal variable names are no longer accessible (they are no longer in scope), though a service can call other services declared in the block. To avoid the possibility of recursion no forward calling of other services is allowed.

Services and their inputs and outputs are available to the main function block body. There are three classes of users for services. Firstly the service owner, the block the service is declared in. Secondly the service instantiator, the block that instantiates the service owner. Finally there are any var references to an instance of the service owner. Every user of a service has their own copy of the input and outputs which contain the result of the last invocation they made of the service. Service internals are however shared between calls of a service.

Service internal variables can be cold started just like any other parameter. Services are cold started after function block cold start. Service input and outputs cannot be cold started. They have the default values 0 for numeric types and the empty string for strings.

Service parameters can have no attributes set. A service can, however, be made unaddressable if desired (it cannot be write protected).

An example service declaration is shown below —

```
FUNCTION_BLOCK ex
VAR_INPUT
    in1: DINT;
END_VAR
VAR_OUTPUT
    out1: DINT;
END_VAR;
VAR
    loc1 : DINT;
END_VAR
```

```

(* set loc1 to service input, if it is
   between 0 and 10, else signal an error *)
SERVICE service1
VAR_INPUT
  myin: DINT;
END_VAR
VAR_OUTPUT
  ok: BOOL;
END_VAR
IF myin < 10 AND myin > 0 THEN
  loc1 := myin;
  ok := TRUE;
ELSE
  ok := FALSE;
END_IF;
END_SERVICE

out1 := in1 + out1 / loc1 ;

END_FUNCTION_BLOCK

```

## 11.2 Service invocation

There are three classes of users for services. Firstly the service owner, the block the service is declared in. Secondly the service instantiator, the block that instantiates the service owner. Finally there are any var references to an instance of the service owner.

Services are invoked in the same way as function blocks; i.e if an input value is not specified it defaults to the last value set by the user.

Each user has a private copy of the service outputs and inputs reflecting the last invocation of the service that they made. It follows that if no invocation has ever been made by that user the values of the service outputs and inputs are zero for numeric values and the empty string for strings.

A service may be invoked either by a block which has a local instance of a block containing services (the service instantiator), or by a block that has a var reference to a block containing services, or within the block (the service owner) itself. The syntax for invocation for a service is like the syntax of a function block call, except that the service name follows the function block name and a dot if the service is in a var reference or locally declared block —

```

<service invocation> ::= [ < function block name> ] . < service name>
                        ( < service input assignments> ) ;

< service input assignments > ::= < function block input assignments>

```

So continuing the above example, an instance of block “ex” called “exinst” would have the service “service1” invoked as

```
exinst.service1( myin := 4 );
```

Outputs from services are read using the usual hierarchic name syntax, e.g.

```
IF exinst.service1.ok THEN
  message( IN := 'set value ok$N');
ELSE
  message( IN := 'failed to set value$N');
END_IF;
```

### 11.3 Service execution

There are two sorts of services. The first is an "immediate service", where the service is executed as soon as possible after it is invoked. The second is a "rendezvous" service where the service request is not executed until the block containing the service has accepted it.

Services have one extra output variable built into them.

This of type USINT and is called "waiting". This will have different values for each user of the service i.e for every remote user using the service via a var reference and for the service owner and instantiator. The values of "waiting" are

0	No service request made or the last request has completed.
1	Request is awaiting execution
2	Request was rejected because another user is waiting for the service to complete (usually only applies for rendezvous services or services with SFC bodies)
3	Waiting for the network

**Services and Var References** A var reference can be made to a block that contains many services; because only one outstanding operation is allowed on var references, only one service request can be outstanding at any one time for each var reference containing services. Trying to call another service when another operation is outstanding will set the local service inputs and then generate an error on the var reference status output, either "bad state" if there is a read or write in progress or "operation in progress" if a service request is in progress (see section 9.7).

If a read or write is done with the var reference none of the service parameters are read or written. This is because they are effectively private to each user of the service. This also means that the service outputs have no meaning until the first invocation of a service.

#### 11.3.1 Immediate services

**Local execution** If the service is invoked in a block instantiated locally or it is a service of the current block, then the service body is executed immediately the service inputs are assigned, just as if the service was a local function block. The "waiting" output should always be 0.



**Remote execution** If the service is invoked in a block that is a reference to a remote block, then execution proceeds as follows -

1. The service inputs are written to the local reference, and a service execute request is sent to the task owning the block with the service. The request contains the local input values of the service. The "waiting" output is set to 3.
2. At the beginning of the remote task cycle the service message is received, and the service should be executed. The output values are sent back to the requesting task in a service response message.
3. When the requesting task receives the response the local output variables are updated and the "waiting" output is set back to 0.
4. The only reason the service might not be executed is if a debug break point had been set in it before the service request arrived, and a previous request had broken at that wait point. In this case the service request will be rejected and the "waiting" output will be set to 2.

The CDL code for executing the service might look like

```
<block name>.<service name>( <input values> )
IF <block name>.<service name>.waiting = 0 THEN
  process service outputs
END_IF;
```

### 11.3.2 Rendezvous services

A rendezvous service is not executed until the function block containing the service accepts it. This is done using a new CDL accept statement —

```
<accept statement> ::= WHEN <boolean condition> ACCEPT <service name>
```

For example

```
FUNCTION_BLOCK ex2
VAR_INPUT
  in1: DINT;
  change: BOOL;
END_VAR
VAR_OUTPUT
  out1: DINT;
END_VAR;
VAR
  loc1 : DINT;
END_VAR

(* set loc1 to service input, if it is
   between 0 and 10, else signal an error *)
SERVICE service1
VAR_INPUT
  myin: DINT;
END_VAR
VAR_OUTPUT
  ok: BOOL;
END_VAR
```

## SERVICES

---

```
IF myin < 10 AND myin > 0 THEN
    loc1 := myin;
    ok := TRUE;
ELSE
    ok := FALSE;
END_SERVICE

out1 := in1 + out1 / loc1 ;

(* only accept changes to loc1 if the input
   change is true *)
WHEN change ACCEPT service1;

END_FUNCTION_BLOCK
```

Only one rendezvous service request to a particular service for a block can be made at a time; any others when one is outstanding are rejected.

The WHEN statement is effectively implemented as

```
IF service has been requested AND <boolean condition> THEN
    execute service body
END_IF
```

**Local execution** If the service is invoked in a block instantiated locally or is to a local declared service, then if no other request is outstanding the input values are assigned, and "waiting" is set to 1. Once the service has been executed then "waiting" is set back to zero. So the following CDL sequence could be used to invoke a rendezvous service

```
IF want to execute a service THEN
    <block name>.<service name>(<input values> )
END_IF;
IF <block name>.<service name>.waiting = 2 THEN
    failed to get service request
ELSIF <block name>.<service name>.waiting = 1 THEN
    set flag to indicate waiting for service result
END_IF;
IF <block name>.<service name>.waiting = 0 AND waiting for service THEN
    process the service outputs
    clear waiting for service flag
END_IF;
<block name>(); (* execute the service owner *)
```

Note that not testing the "waiting" output after asking for a service might be a coding error because the service could complete immediately the service owner block is executed; the service caller would never know that the original request had been accepted.

**Remote execution** If the service is invoked in a block declared as a var reference, then execution proceeds as follows

1. The service inputs are written to the local reference, and a service execute request is sent to the task owning the block with the service. The request contains the local input values of the service. The local "waiting" variable is set to 3.
2. At the beginning of the remote task cycle the service message is received. If no other request is outstanding then the service inputs are written and the service request is noted. A message is sent back to the requestor to set "waiting" to 1. However if there already is a service request outstanding the service inputs are not written and a message is sent back setting "waiting" to 2.
3. When a service is accepted, a message is sent back to the requestor containing the service output values and setting the "waiting" output back to 0.

The CDL code for executing a remote rendezvous service is similar to that for executing a local one.

(See also 11.4).

**Template matching with services** References to blocks containing services go through two phases of template matching. First the input, outputs and any services are matched, and then each service has its parameters separately matched. There is no syntax for overriding the names of the remote parameters, so the matching of service parameters is done purely by name. The actual service name may be overridden like any other function block parameter. For example the following local template

```
FUNCTION_BLOCK exref
SERVICE rem
VAR_INPUT
    myin: DINT;
END_VAR
VAR_OUTPUT
    ok: BOOL;
END_VAR
NO_BODY
END_SERVICE
NO_BODY
END_FUNCTION_BLOCK
```

could match an instance in resource "Res" called "path.exinst" of function block "ex" above using the declaration

```
VAR REFERENCE
    remote: exref { ref := 'Res:path.exinst{ rem := service1}'}
END_VAR
```

No distinction is made between immediate and rendezvous services for the purpose of template matching.

There is no requirement that all the inputs and outputs of the remote service have to be matched, a subset may be used.

**Values of servStatus property** Var references containing services have a “servStatus” property that can be used in the same way as “readStatus” and “writeStatus” (see section 9.9).

0	OK last operation succeeded
1	In progress, i.e waiting for the network
2	Last service request failed because of a network related error or timeout
3	Undefined — no request ever made

Note that value 2 does not mean that a service request was rejected. It only denotes any problems reaching the service provider (i.e if “waiting” is set to 2 “servStatus” will have value 0.)

## 11.4 Service timeouts

For immediate services the usual var reference timeout is used, (section 9.8). If a reply is not received after a timeout, then the servStatus and Status properties of the reference are set to denote the error.

Other errors such as reloading of resources etc. are handled just as for normal references.

For services the timeouts apply to the acknowledgement of the service request. There is no timeout on the actual execution of the service. However a periodic message is sent to the remote task whenever a service is awaiting completion. If no response to this is received then the servStatus property indicates a network error, “waiting” is set to 2 and the Status property indicates a network error.

A user can implement a specific timeout by resetting the reference string after a given amount of time. This will cause a re-read of the template, and discarding of any results of the service if they are received subsequently.

## 11.5 Services with SFC bodies

It is possible for Services to have a sequential function chart as a body. The SFC must have a unique end step, that is one and only one step to which all paths through the chart lead, and from which there are no transitions. Once a Service with an SFC has started execution each time the block the service is declared in is invoked, the Service will execute its SFC once, i.e the actions for the active steps are called, and the transition conditions examined to see which steps will become active and which inactive on the next cycle.

The Service is considered to have completed execution when the end step is reached, and at this point the result are sent back to the invoker of the service. A simple example of an SFC in a Service calling a communications block is shown below —

```
FUNCTION_BLOCK servex
.....
VAR
  (* The 'state' IN_OUT is written 2 for a write, or 3 for a read
  (* request, the comms block sets it to 1 while the request is
  (* pending, and to 0 when it is done *)
  docomms: comms;
END_VAR

SERVICE invcomms
VAR_INPUT
  addr: STRING;
  val: LREAL;
  req: BOOL; (* 0 - read, 1 - write *)
END_VAR
```

```

VAR_OUTPUT
  newval: LREAL;
  ok: BOOL;
END_VAR

INITIAL_STEP start: start_act(N); END_STEP

TRANSITION FROM start TO go := docomms.state = 1; END_TRANSITION

STEP go: go_act(N); END_STEP

TRANSITION FROM go TO end := docomms.state = 0; END_TRANSITION

STEP end : end_act(P); END_STEP

ACTION start_act:
  (* COMMS REQUEST STARTED *)
  IF req THEN
    docomms( state := 2, addr := addr, val := val);
  ELSE
    docomms( state := 3, addr := addr);
  END_IF;
END_ACTION

ACTION go_act:
  (* keep polling the comms block, until it has finished *)
  docomms( );
END_ACTION

ACTION end_act:
  (* COMMS REQUEST DONE *)
  newval := docomms.newval;
  ok := newcomms.ok;
END_ACTION

END_SERVICE
.....
END_FUNCTION_BLOCK

```

## 12 Attributes

*This section is provisional*

The IEC1131-3 Structured Text (ST) language only provides sufficient attributes for objects such as Function Blocks, Parameters, Programs to support the definition of "run-time" algorithms and the "run-time" execution model.

Extra object attributes are required to add information particularly for supporting programming and diagnostic tools and operator stations, for example, a Function Block needs a version number for software maintenance purposes, Function Block parameters need units such as V, KVA, kg/s etc.

In the following example, the extra attributes as defined by CDL are shown in braces <sup>7</sup>.

<sup>7</sup> Attribute names may not be the same as property names for VAR REFERENCES if the attributes are being given to references.

```
FUNCTION_BLOCK example { short_desc := 'a block to demonstrate attributes',
                        class := 'EXAMPLES',
                        view_version := '1.0',
                        exec_version := '1.1'
                      }

VAR_INPUT
    Process_Value : REAL { units := 'kg/m',
                          format := '#f7.2' };
    Direction : USINT { enum_strings := 'UP:DOWN:LEFT:RIGHT'};
END_VAR
```

The CDL extensions provide a generalized method of adding attributes to IEC1131-3 objects. There is no preconceived use of attributes; they simply add information that is absent from the ST language and can be used for such things as : descriptions for assisting with user documentation and on-line help, software version management, run-time parameters eg default execution rates, access control, display information etc.

## 12.1 Attribute definition

All attributes are defined in after the declaration section of an object. Attributes may be added to

- All VAR declarations
- After a Function Block or Program type name declaration
- After a Service name declaration
- After a Resource name
- After the declaration of tasks or blocks in a Resource
- To a step declaration<sup>1</sup>
- To an action declaration<sup>1</sup>
- To a transition declaration<sup>1</sup>

Attribute definition is a name and string value pair specified by

<attribute name> := <attribute string value>

An attribute value can refer to the attribute of another object within the same scope, (except if it is a runtime attribute and the object is a reference). The same rule for scoping are used as for ordinary assignment. The “@” character is the “attribute access” character (whereas the “.” character is used for hierarchich name access). For example

```
VAR
    (* declare attribute attrib and assign value 'val1' *)
    A : DINT { attrib := 'val1' };
    (* declare attribute new and make its value the same as that
       of the attribute 'attrib' of variable A *)
    B : DINT { new := A@attrib }
```

---

<sup>1</sup>Run time attributes (see section 12.2) cannot be added to these

When instantiating a block the attributes of its input, output or in-out parameters defined in the block type declaration may be overridden.

VAR

```
(* Declare a Why attribute for the instance FBINST.
   Override the Units attribute of the PV parameter of FBTYPE.
   Override the Units attribute of the SP parameter of FBTYPE,
   make it the same as the PV attribute of this instance of FBTYPE *)
FBINST: FBTYPE ( PV := 10.0 , SP := 100 )
    { Why := 'TempControl', PV@Units := 'Fahr',
      SP@Units := FBINST.PV@Units }
```

END\_VAR

## 12.2 Attributes at run time

By default attributes are not stored in run time systems. However some systems may wish to store certain attributes at run time. They are then made available to CDL at run time using the reference mechanism. Attributes are read-only at run time.

If an attribute declaration uses the wiring operator `::=` then this means that the attribute should be made available at run time e.g.

VAR

```
A : DINT { attrib ::= 'val1' };
B : DINT { new ::= A@attrib }
```

Run time attributes and non run time attributes are a distinct set. The left hand and right hand side of attribute assignment must both be run time or non-run time, and the operator used for assignment must match; it is an error to mix them using the attribute access `@` character, or by overriding attributes. For example the following is an error —

```
A : DINT { attrib ::= 'val1' };
B : DINT { new := A@attrib };
```

because “new” is not a run time attribute. Similarly if “Units” were a run time attribute, the following would be an error —

```
B1: PID { PV@Units := 'Centigrade' }
```

An attribute can be matched to a CDL string in a VAR REFERENCE. The reference string can contain the `@` character to allow attribute access. For example

VAR REFERENCE

```
X : STRING { ref := 'R1:PID.PV@Units' }
```

END\_VAR

would access the attribute “Units” of the PV parameter of the PID resource level block in resource “R1”.

## 13 Configuration

*This section is not complete*

## 14 Deviations from IEC-1131

*This section is not complete. The deviations from IEC are under review and will be corrected if possible.*

The following lists where IEC-1131 supports more than CDL.

1. There is no support for RETAIN data
2. The set of built in functions to be supported as described by IEC is not yet incorporated into CDL
3. Time literal format does not support floating point time format, or underscores
4. There is no support for hex and octal literals, (This will be provided soon!)
5. Binary literals can only be assigned to string types
6. Integer types can only be assigned decimal literals
7. Keywords for time types (TOD, DT) are not supported
8. Structures and derived data types (other than function blocks) are not supported
9. The default start date for time types is 1/1/1970
10. There is no directly represented io
11. ENO and OK outputs from functions are not supported
12. Globals and var externals are not supported
13. Constants are not supported
14. Function blocks as input, in-outs or outputs are not supported
15. Only a subset of action qualifiers is supported
16. The Configuration construct is not supported
17. At present only one ACTION is supported per step
18. There is no default for an ACTION qualifier
19. There is no ACTION output parameter
20. EDGE is not properly supported
21. No default values for function parameters are supported
22. Function parameters have to be given values

The following lists where CDL extends IEC-1131.

1. Cold start values that are expressions
2. Initialisation of function block instances
3. Var References



4. Services
5. Attributes
6. Inter-block wiring in Resources
7. Wiring statements in function block instance declarations
8. Wiring of function block outputs
9. Specification of target for a task in a Resource (TASK X ON target)
10. Actions that are SFCs
11. Extra data types such as QTIME

