BRITISH STANDARD

BS ISO/IEC 9496:2003

CHILL — The ITU-T programming language

ICS 35.060

NO COPYING WITHOUT BSI PERMISSION EXCEPT AS PERMITTED BY COPYRIGHT LAW

BSI Documents
Document Sharing Hub
British Standards

Full document details: BS ISO/IEC 9496:2003

National foreword

This British Standard reproduces verbatim ISO/IEC 9496:2003 and implements it as the UK national standard. It supersedes BS ISO/IEC 9496:1995 which is withdrawn.

The UK participation in its preparation was entrusted to Technical Committee IST/5, Programming languages, their environment and system software interfaces, which has the responsibility to:

— aid enquirers to understand the text;

— present to the responsible international/European committee any enquiries on the interpretation, or proposals for change, and keep the UK interests informed;

— monitor related international and European developments and promulgate them in the UK.

A list of organizations represented on this committee can be obtained on request to its secretary.

Cross-references

The British Standards which implement international publications referred to in this document may be found in the *BSI Catalogue* under the section entitled "International Standards Correspondence Index", or by using the "Search" facility of the *BSI Electronic Catalogue* or of British Standards Online.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

Compliance with a British Standard does not of itself confer immunity from legal obligations.

Summary of pages

This document comprises a front cover, an inside front cover, the ISO/IEC title page, pages ii to vii, a blank page, pages 1 to 221 and a back cover.

The BSI copyright notice displayed in this document indicates when the document was last issued.

Amendments issued since publication

Amd. No.	Date	Comments

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 13 April 2004

© BSI 13 April 2004

ISBN 0 580 43648 9

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

INTERNATIONAL STANDARD

ISO/IEC 9496

Fourth edition
2003-12-15

CHILL — The ITU-T programming language

CHILL — Le langage de programmation de l'UIT-T

Reference number
ISO/IEC 9496:2003(E)

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 9496:2003

CONTENTS

		Page
1	Introduction	1
1.1	General	1
1.2	Language survey	1
1.3	Modes and classes	2
1.4	Locations and their accesses	3
1.5	Values and their operations	3
1.6	Actions	4
1.7	Input and output	4
1.8	Exception handling	4
1.9	Time supervision	5
1.10	Program structure	5
1.11	Concurrent execution	5
1.12	General semantic properties	6
1.13	Implementation options	6
2	Preliminaries	7
2.1	The metalanguage	7
2.2	Vocabulary	8
2.3	The use of spaces	9
2.4	Comments	9
2.5	Format effectors	9
2.6	Compiler directives	10
2.7	Names and their defining occurrences	10
3	Modes and classes	12
3.1	General	12
3.2	Mode definitions	13
3.3	Mode classification	16
3.4	Discrete modes	17
3.5	Real modes	20
3.6	Powerset modes	22
3.7	Reference modes	22
3.8	Procedure modes	23
3.9	Instance modes	24
3.10	Synchronization modes	25
3.11	Input-Output Modes	26
3.12	Timing modes	28
3.13	Composite modes	29
3.14	Dynamic modes	37
3.15	Moreta Modes	38
4	Locations and their accesses	45
4.1	Declarations	45
4.2	Locations	47
5	Values and their operations	54
5.1	Synonym definitions	54
5.2	Primitive value	55
5.3	Values and expressions	70

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

Page

6 Actions ... 79
6.1 General ... 79
6.2 Assignment action ... 79
6.3 If action ... 81
6.4 Case action ... 81
6.5 Do action ... 83
6.6 Exit action ... 86
6.7 Call action ... 87
6.8 Result and return action ... 90
6.9 Goto action ... 90
6.10 Assert action ... 91
6.11 Empty action ... 91
6.12 Cause action ... 91
6.13 Start action ... 91
6.14 Stop action ... 91
6.15 Continue action ... 92
6.16 Delay action ... 92
6.17 Delay case action ... 92
6.18 Send action ... 93
6.19 Receive case action ... 94
6.20 CHILL built-in routine calls ... 97

7 Input and Output ... 102
7.1 I/O reference model ... 102
7.2 Association values ... 104
7.3 Access values ... 104
7.4 Built-in routines for input output ... 105
7.5 Text input output ... 112

8 Exception handling ... 120
8.1 General ... 120
8.2 Handlers ... 121
8.3 Handler identification ... 121

9 Time supervision ... 122
9.1 General ... 122
9.2 Timeoutable processes ... 122
9.3 Timing actions ... 122
9.4 Built-in routines for time ... 124

10 Program Structure ... 125
10.1 General ... 125
10.2 Reaches and nesting ... 127
10.3 Begin-end blocks ... 129
10.4 Procedure specifications and definitions ... 129
10.5 Process specifications and definitions ... 134
10.6 Modules ... 134
10.7 Regions ... 135
10.8 Program ... 135
10.9 Storage allocation and lifetime ... 136
10.10 Constructs for piecewise programming ... 136
10.11 Genericity ... 141

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

Page

11 Concurrent execution 144
11.1 Processes, tasks, threads and their definitions 144
11.2 Mutual exclusion and regions 145
11.3 Delaying of a thread 148
11.4 Re-activation of a thread 148
11.5 Signal definition statements 148
11.6 Completion of Region and Task locations 149
12 General semantic properties 149
12.1 Mode rules 149
12.2 Visibility and name binding 160
12.3 Case selection 167
12.4 Definition and summary of semantic categories 169
13 Implementation options 173
13.1 Implementation defined built-in routines 173
13.2 Implementation defined integer modes 173
13.3 Implementation defined floating point modes 173
13.4 Implementation defined process names 173
13.5 Implementation defined handlers 173
13.6 Implementation defined exception names 173
13.7 Other implementation defined features 173
Appendix I – Character set for CHILL 175
Appendix II – Special symbols 176
Appendix III – Special simple name strings 177
III.1 Reserved simple name strings 177
III.2 Predefined simple name strings 178
III.3 Exception names 178
Appendix IV – Program examples 179
IV.1 Operations on integers 179
IV.2 Same operations on fractions 179
IV.3 Same operations on complex numbers 180
IV.4 General order arithmetic 180
IV.5 Adding bit by bit and checking the result 180
IV.6 Playing with dates 181
IV.7 Roman numerals 182
IV.8 Counting letters in a character string of arbitrary length 183
IV.9 Prime numbers 184
IV.10 Implementing stacks in two different ways, transparent to the user 184
IV.11 Fragment for playing chess 185
IV.12 Building and manipulating a circularly linked list 188
IV.13 A region for managing competing accesses to a resource 189
IV.14 Queuing calls to a switchboard 190
IV.15 Allocating and deallocating a set of resources 190
IV.16 Allocating and deallocating a set of resources using buffers 192
IV.17 String scanner1 194
IV.18 String scanner2 195
IV.19 Removing an item from a double linked list 196
IV.20 Update a record of a file 196
IV.21 Merge two sorted files 197
IV.22 Read a file with variable length records 198
IV.23 The use of spec modules 199
IV.24 Example of a context 199
IV.25 The use of prefixing and remote modules 199

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

Page

IV.26 The use of text i/o 200
IV.27 A generic stack 201
IV.28 An abstract data type 202
IV.29 Example of a spec module 202
IV.30 Object-Orientation: Modes for Simple, Sequential Stacks 202
IV.31 Object-Orientation: Mode Extension: Simple, Sequential Stack with Operation "Top" 204
IV.32 Object-Orientation: Modes for Stacks with Access Synchronization 204

Appendix V – Decommitted features 206
V.1 Free directive 206
V.2 Integer modes syntax 206
V.3 Set modes with holes 206
V.4 Procedure modes syntax 206
V.5 String modes syntax 207
V.6 Array modes syntax 207
V.7 Level structure notation 207
V.8 Map reference names 207
V.9 Based declarations 207
V.10 Character string literals 207
V.11 Receive expressions 207
V.12 Addr notation 207
V.13 Assignment syntax 207
V.14 Case action syntax 207
V.15 Do for action syntax 207
V.16 Explicit loop counters 208
V.17 Call action syntax 208
V.18 RECURSEFAIL exception 208
V.19 Start action syntax 208
V.20 Explicit value receive names 208
V.21 Blocks 208
V.22 Entry statement 208
V.23 Register names 208
V.24 Recursive attribute 208
V.25 Quasi cause statements and quasi handlers 209
V.26 Syntax of quasi statements 209
V.27 Weakly visible names and visibility statements 209
V.28 Weakly visible names and visibility statements 209
V.29 Pervasiveness 209
V.30 Seizing by modulion name 209
V.31 Predefined simple name strings 209

Appendix VI – Index of production rules 210

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the technical committee are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 9496 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*, in collaboration with ITU-T. The identical text is published as ITU-T Rec. Z.200.

This fourth edition cancels and replaces the third edition (ISO/IEC 9496:1998), which has been technically revised.

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

blank

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 9496:2003

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

CHILL – THE ITU-T PROGRAMMING LANGUAGE

This Recommendation | International Standard defines the ITU-T programming language CHILL. When CHILL was first defined in 1980 "CHILL" stood for CCITT High Level Language.

The following subclauses of this clause introduce some of the motivations behind the language design and provide an overview of the language features.

For information concerning the variety of introductory and training material on this subject, the reader is referred to the Manuals, "Introduction to CHILL" and "CHILL user's manual".

An alternative definition of CHILL, in a strict mathematical form (based on the VDM notation), is available in the Manual entitled "Formal definition of CHILL".

1.1 General

CHILL is a strongly typed, block structured language designed primarily for the implementation of large and complex embedded systems.

CHILL was designed to:

- enhance reliability and run time efficiency by means of extensive compile-time checking;
- be sufficiently flexible and powerful to encompass the required range of applications and to exploit a variety of hardware;
- provide facilities that encourage piecewise and modular development of large systems;
- cater for real-time applications by providing built-in concurrency and time supervision primitives;
- permit the generation of highly efficient object code;
- be easy to learn and use.

The expressive power inherent in the language design allows engineers to select the appropriate constructs from a rich set of facilities such that the resulting implementation can match the original specification more precisely.

Because CHILL is careful to distinguish between static and dynamic objects, nearly all the semantic checking can be achieved at compile time. This has obvious run time benefits. Violation of CHILL dynamic rules results in run-time exceptions which can be intercepted by an appropriate exception handler (however, generation of such implicit checks is optional, unless a user defined handler is explicitly specified).

CHILL permits programs to be written in a machine independent manner. The language itself is machine independent; however, particular compilation systems may require the provision of specific implementation defined objects. It should be noted that programs containing such objects will not, in general, be portable.

1.2 Language survey

A CHILL program consists essentially of three parts:

- a description of objects;
- a description of actions which are to be performed upon the objects;
- a description of the program structure.

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

Objects are described by data statements (declaration and definition statements), actions are described by action statements and the program structure is described by program structuring statements.

The manipulatable objects of CHILL are values and locations where values can be stored. The actions define the operations to be performed upon the objects and the order in which values are stored into and retrieved from locations. The program structure determines the lifetime and visibility of objects.

CHILL provides for extensive static checking of the use of objects in a given context.

In the following subclauses, a summary of the various CHILL concepts is given. Each subclause is an introduction to a clause with the same title, describing the concept in detail.

1.3 Modes and classes

A location has a mode attached to it. The mode of a location defines the set of values which may reside in that location and other properties associated with it (note that not all properties of a location are determinable by its mode alone). Properties of locations are: size, internal structure, read-onliness, referability, etc. Properties of values are: internal representation, ordering, applicable operations, etc.

A value has a class attached to it. The class of a value determines the modes of the locations that may contain the value.

CHILL provides the following categories of modes:

- discrete modes: integer, character, boolean, set (enumerations) modes and ranges thereof;
- real modes: floating point modes and ranges thereof;
- powerset modes: sets of elements of some discrete mode;
- reference modes: bound references, free references and rows used as references to locations;
- composite modes: string, array and structure modes;
- procedure modes: procedures considered as manipulatable data objects;
- instance modes: identifications for processes;
- synchronization modes: event and buffer modes for process synchronization and communication;
- input-output modes: association, access and text modes for input-output operations;
- timing modes: duration and absolute time modes for time supervision;
- moreta modes: module, region and task modes for object orientation with single inheritance.

CHILL provides denotations for a set of standard modes. Program defined modes can be introduced by means of mode definitions. Some language constructs have a so-called dynamic mode attached. A dynamic mode is a mode of which some properties can be determined only dynamically. Dynamic modes are always parameterized modes with run-time parameters. A mode that is not dynamic is called a static mode.

With moreta modes CHILL supports object oriented programming in a very versatile manner. There are three kinds of modes for objects:

- module modes: the values of these modes behave very much like modules and resemble therefore mostly the objects in classical object oriented programming (e.g. Smalltalk, C++, Eiffel, Java);
- region modes: the values of these modes behave very much like regions. Such objects are usually not found in classical object oriented programming;
- task modes: the values of these modes have essentially the same structure as regions but have their own thread of control, and communication between them and other objects is done asynchronously.

Classes have no denotation in CHILL. They are introduced in the metalanguage only to describe static and dynamic context conditions.

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 9496:2003

1.4 Locations and their accesses

Locations are places where values can be stored or from which values can be obtained. In order to store or obtain a value, a location has to be accessed.

Declaration statements define names to be used for accessing a location. There are:

1) location declarations;

2) loc-identity declarations.

The first one creates locations and establishes access names to the newly created locations. The latter one establishes new access names for locations created elsewhere.

Apart from location declarations, new locations can be created by means of a *GETSTACK* or *ALLOCATE* built-in routine call yielding reference values (see below) to the newly created location.

A location may be **referable**. This means that a corresponding reference value exists for the location. This reference value is obtained as the result of the referencing operation, applied to the **referable** location. By dereferencing a reference value, the referred location is obtained. CHILL requires certain locations to be **referable** and others to be not **referable**, but for other locations it is left to the implementation to decide whether or not they are **referable**. Referability must be a statically determinable property of locations.

A location may have a **read-only** mode, which means that it can only be accessed to obtain a value and not to store a new value into it (except when initializing).

A location may be composite, which means that it has sub-locations which can be accessed separately. A sub-location is not necessarily **referable**. A location containing at least one **read-only** sub-location is said to have the **read-only property**. The accessing methods delivering sub-locations (or sub-values) are indexing and slicing for strings and for arrays, and selection for structures.

A location has a mode attached. If this mode is dynamic, the location is called a dynamic mode location.

The following properties of a location, although statically determinable, are not part of the mode:

referability: whether or not a reference value exists for the location;

storage class: whether or not it is statically allocated;

regionality: whether or not the location is declared within a region.

1.5 Values and their operations

Values are basic objects on which specific operations are defined. A value is either a (CHILL) defined value or an **undefined** value (in the CHILL sense). The usage of an undefined value in specified contexts results in an undefined situation (in the CHILL sense) and the program is considered to be incorrect.

CHILL allows locations to be used in contexts where values are required. In this case, the location is accessed to obtain the value contained in it.

A value has a class attached. **Strong** values are values that besides their class also have a mode attached. In that case the value is always one of the values defined by the mode. The class is used for compatibility checking and the mode for describing properties of the value. Some contexts require those properties to be known and a **strong** value will then be required.

A value may be **literal**, in which case it denotes an implementation independent discrete value, known at compile time. A value may be **constant**, in which case it always delivers the same value, i.e. it need only be evaluated once. When the context requires a **literal** or **constant** value, the value is assumed to be evaluated before run-time and therefore cannot generate a run-time exception. A value may be **intra-regional**, in which case it can refer somehow to locations declared within a region. A value may be composite, i.e. contain sub-values.

Synonym definition statements establish new names to denote **constant** values.

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

1.6 Actions

Actions constitute the algorithmic part of a CHILL program.

The assignment action stores a (computed) value into one or more locations. The procedure call invokes a procedure, a built-in routine call invokes a built-in routine (a built-in routine is a procedure whose definition need not be written in CHILL and whose parameter and result mechanism may be more general). To return from and/or establish the result of a procedure call, the return and result actions are used.

To control the sequential action flow, CHILL provides the following flow of control actions:

– if action: for a two-way branch;

– case action: for a multiple branch. The selection of the branch may be based upon several values, similarly to a decision table;

– do action: for iteration or bracketing;

– exit action: for leaving a bracketed action or a module in a structured manner;

– cause action: to cause a specific exception;

– goto action: for unconditional transfer to a labelled program point.

Action and data statements can be grouped together to form a module or begin-end block, which form a (compound) action.

To control the concurrent action flow, CHILL provides the start, stop, delay, continue, send, delay case, and receive case actions, and receive and start expressions.

1.7 Input and output

The input and output facilities of CHILL provide the means to communicate with a variety of devices in the outside world.

The input-output reference model knows three states. In the free state there is no interaction with the outside world.

Through an *ASSOCIATE* operation, the file handling state is entered. In the file handling state there are locations of association mode, which denote outside world objects. It is possible via built-in routines to read and modify the language defined attributes of associations, i.e. **existing**, **readable**, **writeable**, **indexable**, **sequencible** and **variable**. File creation and deletion are also done in the file handling state.

Through the *CONNECT* operation, a location of access mode is connected to a location of an association mode, and the data transfer state is entered. The *CONNECT* operation allows positioning of a **base** index in a file. In the data transfer state various attributes of locations of access mode can be inspected and the data transfer operations *READRECORD* and *WRITERECORD* can be applied.

Through the text transfer operations, CHILL values can be represented in a human-readable form which can be transferred to or from a file or a CHILL location.

1.8 Exception handling

The dynamic semantic conditions of CHILL are those (non context-free) conditions that, in general, cannot be statically determined. (It is left to the implementation to decide whether or not to generate code to test the dynamic conditions at run time, unless an appropriate handler is explicitly specified.) The violation of a dynamic semantic rule causes a run-time exception; however, if an implementation can determine statically that a dynamic condition will be violated, it may reject the program.

Exceptions can also be caused by the execution of a cause action or, conditionally, by the execution of an assert action. When, at a given program point, an exception occurs, control is transferred to the associated handler for that exception, if one is specified. Whether or not a handler is specified for an exception at a given point can be statically determined. If no explicit handler is specified, control may be transferred to an implementation defined exception handler.

Exceptions have a name, which is either a CHILL defined exception name, an implementation defined exception name, or a program defined exception name. Note that when a handler is specified for an exception name, the associated dynamic condition must be checked.

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

1.9 Time supervision

Time supervision facilities of CHILL provide the means to react to the elapsing of time in the external world. A process becomes **timeoutable** when it reaches a well-defined point in the execution of certain actions. At this point it may be interrupted. When this happens, control is transferred to an appropriate handler.

Programs may detect the elapsing of a period of time or may synchronize to an absolute point of time or at precise intervals without cumulated drifts. Built-in routines for time are provided to convert absolute time values and duration values into integer values, to suspend a process until a time supervision expires.

1.10 Program structure

The program structuring statements are the begin-end block, module, procedure, process, region and moreta mode. The program structuring statements provide the means of controlling the lifetime of locations and the visibility of names.

The lifetime of a location is the time during which a location exists within the program. Locations can be explicitly declared (in a location declaration) or generated (*GETSTACK* or *ALLOCATE* built-in routine call), or they can be implicitly declared or generated as the result of the use of language constructs.

A name is said to be **visible** at a certain point in the program if it may be used at that point. The scope of a name encompasses all the points where it is **visible**, i.e. where the denoted object is identified by that name.

Begin-end blocks determine both visibility of names and lifetime of locations.

Modules are provided to restrict the visibility of names to protect against unauthorized usage. By means of visibility statements, it is possible to exercise control over the visibility of names in various program parts.

A procedure is a (possibly parameterized) sub-program that may be invoked (called) at different places within a program. It may return a value (value procedure) or a location (location procedure), or deliver no result. In the latter case the procedure can only be called in a procedure call action.

Processes, task locations, regions and region locations provide the means by which a structure of concurrent executions can be achieved.

Generic templates provide the means by which generic modules, regions, procedures, processes and moreta modes can be constructed. These templates can be parameterized by SYN constants, modes and procedures. Generic instantiation statements are used to obtain (nongeneric) modules, regions, procedures, processes and moreta modes which are called generic instances. A generic instance is obtained from a generic template T by replacing in T the formal generic parameters with the corresponding actual generic parameters.

A complete CHILL program is a list of program units that is considered to be surrounded by an (imaginary) process definition. This outermost process is started by the system under whose control the program is executed. A program unit can be a module, a region, a moreta synmode definition statement, a moreta newmode definition statement or a generic template.

Constructs are provided to facilitate various ways of piecewise development of programs. A spec module and spec region are used to define the static properties of a program piece, a context is used to define the static properties of seized names. In addition it is possible to specify that the text of a program piece is to be found somewhere else through the remote facility.

1.11 Concurrent execution

CHILL allows for the concurrent execution of program units. A thread (process or task) is the unit of concurrent execution. The evaluation of a start expression causes the creation of a new process of the indicated process definition. The process is then considered to be executed concurrently with the starting thread. CHILL allows for one or more processes with the same or different definition to be active at one time. The stop action, executed by a process or a task, causes its termination.

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

A thread is always in one of two states; it can be active or delayed. The transition from active to delayed is called the delaying of the thread; the transition from delayed to active is called the re-activation of the thread. The execution of delaying actions on events, or receiving actions on buffers or signals, or sending actions on buffers, or call action to a component procedure of a region location, or call action to a component procedure of a task location in case there is not enough storage to perform can cause the executing thread to become delayed. The execution of a continue action on events, or sending actions on buffers or signals, or receiving actions on buffers, or release of a region location, or at the beginning of the execution of an externally called component procedure of a task location can cause a delayed thread to become active again.

Buffers and events are locations with restricted use. The operations send, receive and receive case are defined on buffers; the operations delay, delay case and continue are defined on events. Buffers are a means of synchronizing and transmitting information between processes. Events are used only for synchronization. Signals are defined in signal definition statements. They denote functions for composing and decomposing lists of values transmitted between processes. Send actions and receive case actions provide for communication of a list of values and for synchronization.

A region or region location is a special kind of module. Its use is to provide for mutually exclusive access to data structures that are shared by several threads.

1.12 General semantic properties

The semantic (non context-free) conditions of CHILL are the mode and class compatibility conditions (mode checking) and the visibility conditions (scope checking). The mode rules determine how names may be used; the scope rules determine where names may be used.

The mode rules are formulated in terms of compatibility requirements between modes, between classes and between modes and classes. The compatibility requirements between modes and classes and between classes themselves are defined in terms of equivalence relations between modes. If dynamic modes are involved, mode checking is partly dynamic.

The scope rules determine the visibility of names through the program structure and explicit visibility statements. The explicit visibility statements influence the scope of the mentioned names. Names introduced in a program have a place where they are defined or declared. This place is called the defining occurrence of the name. The places where the name is used are called applied occurrences of the name. The name binding rules associate a unique defining occurrence with each applied occurrence of the name.

1.13 Implementation options

CHILL allows for implementation defined integer modes, implementation defined built-in routines, implementation defined **process** names, implementation defined exception handlers and implementation defined exception names.

An implementation defined integer mode must be denoted by an implementation defined **mode** name. This name is considered to be defined in a newmode definition statement that is not specified in CHILL. Extending the existing CHILL-defined arithmetic operations to the implementation defined integer modes is allowed within the framework of the CHILL syntactic and semantic rules. Examples of implementation defined integer modes are long integers, and short integers.

A built-in routine is a procedure whose definition need not be written in CHILL and that may have a more general parameter passing and result transmission scheme than CHILL procedures.

A built-in **process** name is a process name whose definition need not be written in CHILL and that may have a more general parameter passing scheme than CHILL processes. A CHILL process may cooperate with built-in processes or start such processes.

An implementation defined exception handler is a handler appended to a process definition. If this handler receives control after the occurrence of an exception, the implementation decides which actions are to be taken. An implementation defined exception is caused if an implementation defined dynamic condition is violated.

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

2 Preliminaries

2.1 The metalanguage

The CHILL description consists of two parts:

- the description of the context-free syntax;
- the description of the semantic conditions.

2.1.1 The context-free syntax description

The context-free syntax is described using an extension of the Backus-Naur Form. Syntactic categories are indicated by one or more English words, written in slanted characters, enclosed between angular brackets (< and >). This indicator is called a non-terminal symbol. For each non-terminal symbol, a production rule is given in an appropriate syntax section. A production rule for a non-terminal symbol consists of the non-terminal symbol at the left-hand side of the symbol *::=*, and one or more constructs, consisting of non-terminal and/or terminal symbols at the right-hand side. These constructs are separated by a vertical bar (|) to denote alternative productions for the non-terminal symbol.

Sometimes the non-terminal symbol includes an underlined part. This underlined part does not form part of the context-free description but defines a semantic category (see 2.1.2).

Syntactic elements may be grouped together by using curly brackets ({ and }). Repetition of curly bracketed groups is indicated by an asterisk (*) or plus ($^+$). An asterisk indicates that the group is optional and can be further repeated any number of times; a plus indicates that the group must be present and can be further repeated any number of times. For example, { *A* }* stands for any sequence of *A*'s, including zero, while { *A* }$^+$ stands for any sequence of at least one *A*. If syntactic elements are grouped using square brackets ([and]), then the group is optional. A curly or square bracketed group may contain one or more vertical bars, indicating alternative syntactic elements.

A distinction is made between strict syntax, for which the semantic conditions are given directly, and derived syntax. The derived syntax is considered to be an extension of the strict syntax and the semantics for the derived syntax is indirectly explained in terms of the associated strict syntax.

It is to be noted that the context-free syntax description is chosen to suit the semantic description in this Recommendation | International Standard and is not made to suit any particular parsing algorithm (e.g. there are some context-free ambiguities introduced in the interest of clarity). The ambiguities are resolved using the semantic category of the syntactic elements.

2.1.2 The semantic description

Each syntactic category (non-terminal symbol) is described in sub-sections **semantics**, **static properties**, **dynamic properties**, **static conditions** and **dynamic conditions**.

The section **semantics** describes the concepts denoted by the syntactic categories (i.e. their meaning and behaviour).

The section **static properties** defines statically determinable semantic properties of the syntactic category. These properties are used in the formulation of static and/or dynamic conditions in the sections where the syntactic category is used.

The section **dynamic properties** defines the properties of the syntactic category, which are known only dynamically.

The section **static conditions** describes the context-dependent, statically checkable conditions which must be fulfilled when the syntactic category is used. Some static conditions are expressed in the syntax by means of an underlined part in the non-terminal symbol (see 2.1.1). This use requires the non-terminal to be of a specific semantic category. For example, *<u>boolean</u> expression* is identical to <*expression*> in the context-free sense, but semantically it requires the *expression* to be of a boolean class.

The section **dynamic conditions** describes the context-dependent conditions that must be fulfilled during execution. In some cases, conditions are static if no dynamic modes are involved. In those cases, the condition is mentioned under **static conditions** and referred to under **dynamic conditions**. In other cases, dynamic conditions can be checked statically; an implementation may treat this as a violation of a static condition.

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

In the semantic description, different fonts are used in the following ways: slanted font (without < and >) is used to indicate syntactic objects; corresponding terms in roman font indicate corresponding semantic objects (e.g. a *location* denotes a location). Bolding is used to name semantic properties; sometimes a property can be expressed syntactically as well as semantically (e.g. the sentence "the *expression* is **constant**" means the same as "the *expression* is a *constant expression*").

Unless otherwise specified, the semantics, properties and conditions described in the sub-section of a syntactic category hold regardless of the context in which in other sections that syntactic category may appear.

The properties of a syntactic category *A* that has a production rule of the form *A ::= B*, where *B* is a syntactic category, are the same as *B* unless otherwise specified.

In this Recommendation | International Standard, virtual names are introduced to describe modes, locations and values which do not occur explicitly in the program text. In such cases the name is preceded by an ampersand (&) symbol. These names are introduced for descriptive purposes only.

2.1.3 The examples

For most syntax sections, there is a section **examples** giving one or more examples of the defined syntactic categories. These examples are extracted from a set of program examples contained in Appendix IV. References indicate via which syntax rule each example is produced and from which example it is taken.

For example, *6.20 (d+5)/5 (1.2)* indicates an example of the terminal string *(d+5)/5*, produced via rule *(1.2)* of the appropriate syntax section, taken from program example no. *6* line *20*.

2.1.4 The binding rules in the metalanguage

Sometimes the semantic description mentions CHILL **special** simple name strings (see Appendix III). These **special** simple name strings are always used with their CHILL meaning and are therefore not influenced by the binding rules of an actual CHILL program.

2.2 Vocabulary

Programs are represented using the CHILL character set (see Appendix I) except for wide character literals, wide character string literals and comments. The representation of a CHILL program is not specified, which means that it is also possible to use a multi-byte character representation. The CHILL alphabet is represented by the syntactic category <*character*>, from which any character that is in the CHILL character set can be derived as a terminal production. The characters of UCS-2 level 1 are represented by the syntactic category <*wide character*>, from which any character that is in the UCS-2 level 1 set can be derived as a terminal production.

The lexical elements of CHILL are:

- special symbols;
- simple name strings;
- literals.

The special symbols are listed in Appendix II. They can be formed by a single character or by character combinations.

Simple name strings are formed according to the following syntax:

syntax:

<*simple name string*> ::= *(1)*
<*letter*> { <*letter*> | <*digit*> | _ }* *(1.1)*

<*letter*> ::= *(2)*
A | B | C | D | E | F | G | H | I | J | K | L | M *(2.1)*
| N | O | P | Q | R | S | T | U | V | W | X | Y | Z *(2.2)*
| a | b | c | d | e | f | g | h | i | j | k | l | m *(2.3)*
| n | o | p | q | r | s | t | u | v | w | x | y | z *(2.4)*

<*digit*> ::= *(3)*
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 *(3.1)*

AVA Documents Document Sharing Hub
Full document details: BS ISO/IEC 9496:2003

semantics: The underline character (_) forms part of the simple name string; e.g. the simple name string *life_time* is different from the simple name string *lifetime*. Lower case and upper case letters are different, e.g. *Status* and *status* are two different simple name strings.

The language has a number of **special** simple name strings with predetermined meanings (see Appendix III). Some of them are **reserved**, i.e. they cannot be used for other purposes.

The **special** simple name strings in a piece must either all be in upper case representation or all be in lower case representation. The **reserved** simple name strings are only reserved in the chosen representation (e.g. if the lower case fashion is chosen, **row** is reserved, **ROW** is not).

static conditions: A *simple name string* may not be one of the **reserved** simple name strings (see Appendix III.1).

2.3 The use of spaces

A sequence of one or more spaces is allowed before and after each lexical element. Such a sequence is called a delimiter. Lexical elements are also terminated by the first character that cannot be part of the lexical element. For instance, *IFBTHEN* will be considered a *simple name string* and not as the beginning of an action **IF** B **THEN**, //* will be considered as the concatenation symbol (//) followed by an asterisk (*) and not as a divide symbol (/) followed by a comment opening bracket (/*).

2.4 Comments

syntax:

<comment> ::= *(1)*
<bracketed comment> *(1.1)*
| *<line-end comment>* *(1.2)*

<bracketed comment> ::= *(2)*
/ <character string> */* *(2.1)*

<line-end comment> ::= *(3)*
– – <character string> <end-of-line> *(3.1)*

<character string> ::= *(4)*
*{ <character> }** *(4.1)*
| *{ <wide character> }** *(4.2)*

NOTE – *End-of-line* denotes the end of the line in which the comment occurs.

semantics: A *comment* conveys information to the reader of a program. It has no influence on the program semantics.

A *comment* may be inserted at all places where spaces are allowed as delimiters.

A *bracketed comment* is terminated by the first occurrence of the special sequence: */. A *line-end comment* is terminated by the first occurrence of the end of the line.

examples:

4.1 */* from collected algorithms from CACM no. 93 */* *(2.1)*

2.5 Format effectors

The format effectors BS (Backspace), CR (Carriage return), FF (Form feed), HT (Horizontal tabulation), LF (Line feed), VT (Vertical tabulation) of the CHILL character set (see Appendix I, positions FE_0 to FE_5) and the *end-of-line* are not mentioned in the CHILL context-free syntax description. When used, they have the same delimiting effect as a space. Spaces and format effectors may not occur within lexical elements (except character string literals).

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

2.6 Compiler directives

syntax:

<directive clause> ::= (1)
<> <directive> { , <directive> } <>* (1.1)

<directive> ::= (2)
<implementation directive> (2.1)

semantics: A directive clause conveys information to the compiler. This information is specified in an implementation defined format.

An implementation directive must not influence the program semantics, i.e. a program with implementation directives is correct, in the CHILL sense, if and only if it is correct without these directives.

A *directive clause* is terminated by the first occurrence of the directive ending symbol (<>). A *directive* may contain any character of the character set (see Appendix I).

static properties: A *directive clause* may be inserted at any place where spaces are allowed as delimiters. It has the same delimiting effect as a space. The names used in a *directive clause* follow an implementation defined name binding scheme which does not influence the CHILL name binding rules (see 12.2).

2.7 Names and their defining occurrences

syntax:

<name> ::= (1)
<name string> (1.1)
| *<qualified name>* (1.2)
| *<moreta component name>* (1.3)

<name string> ::= (2)
<simple name string> (2.1)
| *<prefixed name string>* (2.2)

<prefixed name string> ::= (3)
<prefix> ! <simple name string> (3.1)

<prefix> ::= (4)
*<simple prefix> { ! <simple prefix> }** (4.1)

<simple prefix> ::= (5)
<simple name string> (5.1)

<defining occurrence> ::= (6)
<simple name string> (6.1)

<defining occurrence list> ::= (7)
*<defining occurrence> { , <defining occurrence> }** (7.1)

<set element name> ::= (8)
<simple name string> (8.1)

<set element name defining occurrence> ::= (9)
<simple name string> (9.1)

<field name> ::= (10)
<simple name string> (10.1)

<field name defining occurrence> ::= (11)
<simple name string> (11.1)

<field name defining occurrence list> ::= (12)
*<field name defining occurrence> { , <field name defining occurrence> }** (12.1)

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

<exception name> ::= *(13)*
<simple name string> *(13.1)*
| *<prefixed name string>* *(13.2)*

<text reference name> ::= *(14)*
<simple name string> *(14.1)*
| *<prefixed name string>* *(14.2)*

<component name> ::= *(15)*
<simple name string> *(15.1)*

<component name defining occurrence> ::= *(16)*
<simple name string> *(16.1)*

<qualified name> ::= *(17)*
<simple name string> ! *<component name>* *(17.1)*

<moreta component name> ::= *(18)*
<moreta location> **.** { *<simple name string>* | *<qualified name>* } *(18.1)*

semantics: Names in a program denote objects. Given an occurrence of a *name* (formally: an occurrence of a terminal production of *name*) in a program, the binding rules of 12.2 provide *defining occurrences* (formally: occurrences of terminal productions of *defining occurrence*) to which that (occurrence of) *name* is **bound**. The *name* then denotes the object defined or declared by the *defining occurrences*. (There can be more than one *defining occurrence* for a *name* in the case of *names* with **quasi** *defining occurrences* and in the case of *names* of components of moreta modes.)

Defining occurrences are said to define the *name*. A *name* is said to be an applied occurrence of the name created by the *defining occurrence* to which it is **bound**. The *name* has its rightmost *simple name string* equal to that of the name.

Similarly, field names are **bound** to field name defining occurrences and denote the fields (of a structure mode) defined by those field name defining occurrences. *Moreta component names* are bound to *component defining occurrences* and denote the components (of a moreta mode) defined by those *component name defining occurrences*.

Exception names are used to identify exception handlers according to the rules stated in clause 8.

Text reference names are used to identify descriptions of pieces of source text in an implementation defined way, subject to the rules in 10.10.1.

When a name is **bound** to more than one defining occurrence, each of the defining occurrences to which the name is **bound** defines or declares the same object (see 10.10 and 12.2.2 for precise rules).

Qualified names are used to identify components of *moreta modes.*

definition of notation: Given a *name string* NS, and a string of characters P, which is either a *prefix* or is empty, the result of prefixing NS with P, written P ! NS, is defined as follows:

- if P is empty, then P ! NS is NS;
- otherwise P ! NS is the name string obtained by concatenating all the characters in P, a prefixing operator and all the characters in NS.

For example, if P is "*q ! r*" and NS is "*s ! n*" then P ! NS is "*q ! r ! s ! n*".

static properties: Each *simple name string* has a **canonical** name string attached which is the *simple name string* itself. A *name string* has a **canonical** name string attached which is:

- if the *name string* is a *simple name string*, then the **canonical** name string of that *simple name string*;
- if the *name string* is a *prefixed name string*, then the concatenation in left to right order of all *simple name strings* in the *name string*, separated by prefixing operators, i.e. interspersed spaces, comments and format effectors (if any) are left out.

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

In the rest of this Recommendation | International Standard:

- the name string of a *name*, *exception name* or *text reference name* is used to denote the **canonical** name string of the *name string* in that *name*, *exception name* or *text reference name*, respectively;

- the name string of a *defining occurrence*, *field name, field name defining occurrence*, *moreta component name* or *moreta component defining occurrence* is used to denote the **canonical** name string of the *simple name string* in that *defining occurrence*, *field name, field name defining occurrence*, *moreta component name* or *moreta component defining occurrence*, respectively.

The binding rules are such that:

- names with a simple name string are **bound** to defining occurrences with the same name string;

- names with a prefixed name string are **bound** to defining occurrences with the same name string as the rightmost simple name string in the prefixed name string of the name;

- field names are **bound** to field name defining occurrences with the same name string as the field names.

- *moreta component names* are **bound** to *moreta component name defining occurrences* with the same *name string* as the *moreta component names*.

A *name* inherits all the static properties attached to the name defined by the *defining occurrence* to which it is **bound**. A *field name* inherits all static properties attached to the field name defined by the *field name defining occurrence* to which it is **bound**. A *moreta component name* inherits all static properties attached to the *moreta component name* defined by the *moreta component name defining occurrence* to which it is bound.

static conditions: The *simple name string* denoted in a *qualified name* and followed by **!** must be a *moreta mode name*.

If a qualified name of the form "M ! component name" occurs outside the definition of the moreta mode M, then the component name must be the name of a SYN, a SYNMODE, or a NEWMODE component of M.

3 Modes and classes

3.1 General

A location has a mode attached to it; a value has a class attached to it. The mode attached to a location defines the set of values that may be contained in the location, the access methods of the location and the allowed operations on the values. The class attached to a value is a means of determining the modes of the locations that may contain the value. Some values are **strong**. A **strong** value has a class and a mode attached. **Strong** values are required in those value contexts where mode information is needed.

3.1.1 Modes

CHILL has static modes (i.e. modes for which all properties are statically determinable) and dynamic modes (i.e. modes for which some properties are only known at run time). Dynamic modes are always parameterized modes with run-time parameters.

Static modes are terminal productions of the syntactic category *mode*.

Modes are also parameterized by values not explicitly denoted in the program text.

3.1.2 Classes

Classes have no denotation in CHILL.

The following kinds of classes exist and any value in a CHILL program has a class of one of these kinds:

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 9496:2003

For a mode M there exists the M-value class. All values with such a class and only those values are **strong** and the mode attached to the value is M.

- For a mode M there exists the M-derived class.
- For any mode M there exists the M-reference class.
- The **null** class.
- The **all** class.

The last two classes are constant classes, i.e. they do not depend on a mode M. A class is said to be dynamic if and only if it is an M-value class, an M-derived class, or an M-reference class, where M is a dynamic mode.

3.1.3 Properties of, and relations between, modes and classes

Modes in CHILL have properties. These may be hereditary or non-hereditary properties. A hereditary property is inherited from a defining mode to a **mode** name defined by it. Below a summary is given of the properties that apply to all modes (except for the first, they are all defined in 12.1):

- A mode has a **novelty** (defined in 3.2.2, 3.2.3 and 3.3).
- A mode can have the **read-only property**.
- A mode can be **parameterizable**.
- A mode can have the **referencing property**.
- A mode can have the **tagged parameterized property**.
- A mode can have the **non-value property**.

Classes in CHILL may have the following properties (defined in 12.1):

- A class can have a **root** mode.
- One or more classes may have a **resulting class**.

Operations in CHILL are determined by the modes and classes of locations and values. This is expressed by the mode checking rules which are defined in 12.1 as a number of relations between modes and classes. There exists the following relations:

- Two modes can be **similar**.
- Two modes can be **v-equivalent**.
- Two modes can be **equivalent**.
- Two modes can be **l-equivalent**.
- Two modes can be **alike**.
- Two modes can be **novelty bound**.
- Two modes can be **read-compatible**.
- Two modes can be **dynamic read-compatible**.
- Two modes can be **dynamic equivalent**.
- A mode can be **restrictable** to a mode.
- A mode can be **compatible** with a class.
- A class can be **compatible** with a class.

3.2 Mode definitions

3.2.1 General

syntax:

<mode definition> ::= *(1)*
<defining occurrence list> = <defining mode> *(1.1)*

<defining mode> ::= *(2)*
<mode> *(2.1)*

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

derived syntax: A *mode definition* where the *defining occurrence list* consists of more than one *defining occurrence* is derived from several mode definitions, one for each *defining occurrence*, separated by commas, with the same *defining mode*. For example:

NEWMODE *dollar, pound = INT*;

is derived from:

NEWMODE *dollar = INT, pound = INT*;

semantics: A mode definition defines a name that denotes the specified mode. Mode definitions occur in synmode and newmode definition statements. A synmode is **synonymous** with its defining mode. A newmode is not **synonymous** with its defining mode. The difference is defined in terms of the property **novelty**, that is used in the mode checking (see 12.1).

static properties: A *defining occurrence* in a *mode definition* defines a **mode** name.

Predefined **mode** names, implementation defined integer **mode** names and implementation defined floating point **mode** names (if any, see 3.4.2 and 3.5.1) are also **mode** names.

A **mode** name has a **defining** mode which is the *defining mode* in the *mode definition* which defines it. (For predefined and implementation defined **mode** names this **defining** mode is a virtual mode.) The hereditary properties of a **mode** name are those of its **defining** mode.

A set of recursive definitions is a set of mode definitions or synonym definitions (see 5.1) such that the *defining mode* in each *mode definition* or *constant value* or *mode* in each *synonym definition* is, or directly contains, a **mode** name or a **synonym** name defined by a definition in the set.

A set of recursive mode definitions is a set of recursive definitions having only mode definitions.

Any mode being or containing a **mode** name defined in a set of recursive mode definitions is said to denote a recursive mode. A path in a set of recursive mode definitions is a list of **mode** names, each name indexed with a marker such that:

- all names in the path have a different definition;
- for each name, its successor is or directly occurs in its defining mode (the successor of the last name is the first name);
- the marker indicates uniquely the position of the name in the defining mode of its predecessor (the predecessor of the first name is the last name).

[Example: **NEWMODE** *M* = **STRUCT** *(i M, n* **REF** *M)*; contains two paths: $\{M_i\}$ and $\{M_n\}$.]

A path is **safe** if and only if at least one of its names is contained in a *reference mode*, a *row mode*, or a *procedure mode* at the marked place.

static conditions: For any set of recursive mode definitions, all its paths must be **safe**. (The first path of the example above is not **safe**.)

examples:

1.15 *operand_mode = INT* *(1.1)*

3.3 *complex* = **STRUCT** *(re,im FLOAT)* *(1.1)*

3.2.2 Synmode definitions

syntax:

<synmode definition statement> ::= *(1)*
SYNMODE *<mode definition>* { , *<mode definition>* }* ; *(1.1)*
| *<remote program unit>* *(1.2)*

semantics: A synmode definition statement defines **mode** names which are **synonymous** with their defining mode.

AVA Documents Document Sharing Hub
Full document details: BS ISO/IEC 9496:2003

static properties: A *defining occurrence* in a *mode definition* in a *synmode definition statement* defines a **synmode** name (which is also a **mode** name). A **synmode** name is said to be **synonymous** with a mode M (conversely, M is said to be **synonymous** with the **synmode** name) if and only if:

- either M is the **defining** mode of the **synmode** name;
- or the **defining** mode of the **synmode** name is itself a **synmode** name **synonymous** with M.

Two mode names A and B are **synonymous** if and only if:

- either A and B are the same name;
- or A is the **defining** mode of B and B is a **synmode** name;
- or B is the **defining** mode of A and A is a **synmode** name;
- or the **defining** mode name of A is **synonymous** to B and A is a **synmode** name;
- or the **defining** mode name of B is **synonymous** to A and B is a **synmode** name.

The **novelty** of a **synmode** name is that of its **defining** mode.

If the **defining** mode is a discrete range mode or a floating point range mode, then the **parent** mode of the **synmode** name is that of its **defining** mode. If the **defining** mode is a **varying** string mode, then the **component** mode of the **synmode** name is that of its **defining** mode.

examples:

6.3 **SYNMODE** *month* = **SET** *(jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec);* *(1.1)*

3.2.3 Newmode definitions

syntax:

<newmode definition statement> ::= *(1)*
NEWMODE *<mode definition>* { , *<mode definition>*}*; *(1.1)*
| *<remote program unit>* *(1.2)*

semantics: A newmode definition statement defines **mode** names which are not **synonymous** with their defining mode.

static properties: A *defining occurrence* in a *mode definition* in a *newmode definition statement* defines a **newmode** name (which is also a **mode** name).

The **novelty** of the **newmode** name is the *defining occurrence* which defines it. If the **defining** mode of the **newmode** name is a discrete range mode or a floating point range mode, then the virtual mode &*name* is introduced as the **parent** mode of the **newmode** name. The **defining** mode of &*name* is the **parent** mode of the discrete range mode or the one of the floating point range mode, and the **novelty** of &*name* is that of the **newmode** name.

If the **defining** mode is a **varying** string mode, then the virtual mode &*name* is introduced as the **component** mode of the **newmode** name. The defining mode of &*name* is the **component** mode of the **varying** string mode, and the **novelty** of &*name* is that of the **newmode** name.

If the *defining occurrence* of the mode definition is a **quasi** *defining occurrence*, then the **novelty** is a **quasi novelty**, otherwise it is a **real novelty**.

static conditions: If the **novelty** is a **quasi novelty**, then at most one **real novelty** must be **novelty bound** to it.

examples:

11.6 **NEWMODE** *line* = *INT* (1:8); *(1.1)*

11.12 **NEWMODE** *board* = **ARRAY** *(line)* **ARRAY** *(column) square*; *(1.1)*

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

3.3 Mode classification

syntax:

<mode> ::= *(1)*
[**READ**] *<non-composite mode>* *(1.1)*
| [**READ**] *<composite mode>* *(1.2)*
| *<formal generic mode indication>* *(1.3)*

<non-composite mode> ::= *(2)*
<discrete mode> *(2.1)*
| *<real mode>* *(2.2)*
| *<powerset mode>* *(2.3)*
| *<reference mode>* *(2.4)*
| *<procedure mode>* *(2.5)*
| *<instance mode>* *(2.6)*
| *<synchronization mode>* *(2.7)*
| *<input-output mode>* *(2.8)*
| *<timing mode>* *(2.9)*

semantics: A mode defines a set of values and the operations which are allowed on the values. A mode may be a **read-only** mode, indicating that a location of that mode may not be accessed to store a value. A mode has a **novelty**, indicating whether it was introduced via a newmode definition statement or not.

static properties: A mode has the following hereditary properties:

- It is a **read-only** mode if it is an explicit or an implicit **read-only** mode.
- It is an explicit **read-only** mode if **READ** is specified or it is a **parameterized** array mode, a **parameterized** string mode or a **parameterized** structure mode, where the **origin** array mode name, **origin** string mode name or **origin variant** structure mode name, respectively, in it is a **read-only** mode.
- It is an implicit **read-only** mode if it is not an explicit **read-only** mode and if:
  - it is the **element** mode of a **read-only** string mode or a **read-only** array mode (see 3.13.2 and 3.13.3);
  - it is a **field** mode of a **read-only** structure mode or it is the mode of a **tag** field of a **parameterized** structure mode (see 3.13.4).

A *mode* has the same properties as the *non-composite mode* or *composite mode* in it. In the following sections, the properties are defined for predefined **mode** names and for *modes* that are not *mode names*; the properties of *mode names* are defined in 3.2. **Read-only** modes have the same properties as their corresponding non-**read-only** modes except for the **read-only property** (see 12.1.1.1).

A mode has the following non-hereditary properties:

- A **novelty** that is either **nil** or the *defining occurrence* in a *mode definition* in a *newmode definition statement*. The **novelty** of a mode which is not a *mode name* (nor **READ** *mode name*) is defined as follows:
  - if it is a **parameterized** string mode, a **parameterized** array mode or a **parameterized** structure mode, its **novelty** is that of its **origin** string mode, **origin** array mode or **origin variant** structure mode, respectively;
  - if it is a discrete range mode or a floating point range mode, its **novelty** is that of its **parent** mode;
  - otherwise its **novelty** is **nil**.

  The **novelty** of a mode that is a *mode name* (**READ** *mode name*) is defined in 3.2.2 and 3.2.3.
- A **size** that is the value delivered by *SIZE (&M)*, where *&M* is a virtual **synmode** name **synonymous** with the *mode*.

AVA Documents
Document Sharing Hub
Full document details: BS ISO/IEC 9496:2003

3.4 Discrete modes

3.4.1 General

syntax:

<discrete mode> ::= *(1)*
<integer mode> *(1.1)*
| *<boolean mode>* *(1.2)*
| *<character mode>* *(1.3)*
| *<set mode>* *(1.4)*
| *<discrete range mode>* *(1.5)*

semantics: A discrete mode defines sets and subsets of totally-ordered values.

3.4.2 Integer modes

syntax:

<integer mode> ::= *(1)*
<integer mode name> *(1.1)*

predefined names: The name *INT* is predefined as an **integer mode** name.

semantics: An integer mode defines a set of signed integer values between implementation defined bounds over which the usual ordering and arithmetic operations are defined (see 5.3). An implementation may define other integer modes with different bounds (e.g. *LONG_INT*, *SHORT_INT*, *UNSIGNED_INT*) that may also be used as **parent** modes for ranges (see 13.2). The *&INT* mode is introduced as the virtual mode that contains all the values of all **predefined** integer modes defined by the implementation. The internal representation of an integer value is the integer value itself. Note that *&INT* is not a **predefined** mode (although it may have the same bounds as those of a **predefined** integer mode).

static properties: An integer mode has the following hereditary properties:

- An **upper bound** and a **lower bound** which are the literals denoting respectively the highest and lowest value defined by the integer mode. They are implementation defined.
- A **number of values** which is **upper bound** – **lower bound** + 1.

examples:

1.5 *INT* *(1.1)*

3.4.3 Boolean modes

syntax:

<boolean mode> ::= *(1)*
<boolean mode name> *(1.1)*

predefined names: The name *BOOL* is predefined as a **boolean mode** name.

semantics: A boolean mode defines the logical truth values (*TRUE* and *FALSE*), with the usual boolean operations (see 5.3). The internal representations of *FALSE* and *TRUE* are the integer values 0 and 1, respectively. This representation defines the ordering of the values.

static properties: A boolean mode has the following hereditary properties:

- An **upper bound** which is *TRUE*, and a **lower bound** which is *FALSE*.
- A **number of values** which is 2.

examples:

5.4 *BOOL* *(1.1)*

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

3.4.4 Character modes

syntax:

<character mode> ::= *(1)*
<character mode name> *(1.1)*

predefined names: The names *CHAR* and *WCHAR* are predefined as **character mode** names.

semantics: A character mode defines the character values as described by the CHILL character set (see Appendix I) in case of *CHAR* or by ISO/IEC 10646-1 in case of *WCHAR*. These alphabets define the ordering of the characters and the integer values which are their internal representations.

static properties: A character mode has the following hereditary properties:

- An **upper bound** and a **lower bound** which are the character literals denoting respectively the highest and lowest value defined by *CHAR* or *WCHAR* respectively.
- A **number of values** which is 256 in case of *CHAR*, and which is given in ISO/IEC 10646-1 in case of *WCHAR*.

examples:

8.4 CHAR *(1.1)*

3.4.5 Set modes

syntax:

<set mode> ::= *(1)*
SET (*<set list>*) *(1.1)*
| *<set mode name>* *(1.2)*

<set list> ::= *(2)*
<numbered set list> *(2.1)*
| *<unnumbered set list>* *(2.2)*

<numbered set list> ::= *(3)*
<numbered set element> { , *<numbered set element>*}* *(3.1)*

<numbered set element> ::= *(4)*
<set element name defining occurrence> = *<integer literal expression>* *(4.1)*

<unnumbered set list> ::= *(5)*
<set element> { , *<set element>*}* *(5.1)*

<set element> ::= *(6)*
<set element name defining occurrence> *(6.1)*

semantics: A set mode defines a set of named and unnamed values. The named values are denoted by the names defined by *defining occurrences* in the *set list*; the unnamed values are the other values. The internal representation of the named values is the integer value associated with them. This representation defines the ordering of the values.

The maximum **number of values** of a set mode is implementation defined.

static properties: A *defining occurrence* in a *set list* defines a **set element** name. A **set element** name has a **set** mode attached, which is the set mode.

A set mode has the following hereditary properties:

- A set of **set element** names which is the set of names defined by *defining occurrences* in its *set list*.
- Each **set element** name of a set mode has an internal representation value attached which is, in the case of a *numbered set element*, the value delivered by the *integer literal expression* in it; otherwise one of the values 0, 1, 2, etc., according to its position in the *unnumbered set list*. For example in: **SET** *(a, b)*, *a* has representation value 0, and *b* has representation value 1 attached.
- An **upper bound** and a **lower bound** which are its **set element** names with the highest and lowest representation values, respectively.

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 9496:2003

- A **number of values** which is the highest of the values attached to the **set element** names plus 1.
- It is a **numbered** set mode if the *set list* in it is a *numbered set list*; otherwise it is an **unnumbered** set mode.

static conditions: For each pair of *integer literal expressions* e_1, e_2 in the *set list NUM* (e_1) and *NUM* (e_2) must deliver different non-negative results.

examples:

11.7 **SET** *(occupied, free)* *(1.1)*

6.3 *month* *(1.2)*

3.4.6 Discrete range modes

syntax:

<discrete range mode> ::= *(1)*
<discrete mode name> (<literal range>) *(1.1)*
| **RANGE** (*<literal range>*) *(1.2)*
| **BIN** (*<integer literal expression>*) *(1.3)*
| *<discrete range mode name>* *(1.4)*

<literal range> ::= *(2)*
<lower bound> : <upper bound> *(2.1)*

<lower bound> ::= *(3)*
<discrete literal expression> *(3.1)*

<upper bound> ::= *(4)*
<discrete literal expression> *(4.1)*

derived syntax: The notation **BIN** *(n)* is derived from **RANGE** $(0 : 2^n - 1)$, e.g. **BIN** *(2+1)* stands for **RANGE** *(0 : 7)*.

semantics: A discrete range mode defines the set of values ranging between the bounds specified (bounds included) by the *literal range*. The range is taken from a specific **parent** mode that determines the operations on and ordering of the range values.

static properties: A discrete range mode has the following non-hereditary property: it has a **parent** mode, defined as follows:

- If the discrete range mode is of the form:

  <discrete mode name> (<literal range>)

  then if the *discrete mode name* is not a discrete range mode, the **parent** mode is the *discrete mode name*; otherwise it is the **parent** mode of the *discrete mode name*.

- If the discrete range mode is of the form:

  RANGE (*<literal range>*)

  then the **parent** mode depends on the **resulting class** of the classes of the *upper bound* and *lower bound* in the *literal range*:

  – if it is an M-derived class, where M is an integer mode, then the **parent** mode is a **predefined** integer mode chosen by the implementation such that it contains the range of values delivered by *literal range*;

  – otherwise it is the **root** mode of the **resulting class**.

- If the discrete range mode is a *discrete range mode name* which is a **synmode** name, then its **parent** mode is that of the **defining** mode of the **synmode** name; otherwise it is a **newmode** name and then its **parent** mode is the virtually introduced **parent** mode (see 3.2.3).

A discrete range mode has the following hereditary properties:

- An **upper bound** and a **lower bound** which are the literals denoting the values delivered by *lower bound* and *upper bound*, respectively, in the *literal range*.
- A **number of values** which is the value delivered by *NUM* (U) – *NUM* (L) + 1, where U and L denote respectively the **upper bound** and **lower bound** of the discrete range mode.
- It is a **numbered** range mode if its **parent** mode is a **numbered** set mode.

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

static conditions: The classes of *upper bound* and *lower bound* must be **compatible** and both must be **compatible** with the *discrete mode name*, if specified.

Lower bound must deliver a value that is less than or equal to the value delivered by *upper bound*, and both values must belong to the set of values defined by *discrete mode name*, if specified.

The *integer literal expression* in case of **BIN** must deliver a non-negative value.

If the **parent** mode is an integer mode, there must exist a **predefined** integer mode that contains the set of values included between the **lower bound** and the **upper bound**.

If the discrete range mode is of the form:

RANGE (*<literal range>*) or *<discrete mode name>* (*<literal range>*)

then the evaluation of the 1.*lower bound*, 2.*upper bound*, must not depend directly or indirectly on the value of the 1.**lower bound**, 2.**upper bound** of the discrete range mode. If the discrete range mode is of the form:

BIN (*<integer literal expression>*)

then the evaluation of the *integer literal expression* must not depend directly or indirectly on the value of the **upper bound** of the discrete range mode.

examples:

9.5 INT (2:max) *(1.1)*

11.12 line *(1.4)*

3.5 Real modes

syntax:

<real mode> ::= *(1)*
<floating point mode> *(1.1)*
| *<floating point range mode>* *(1.2)*

semantics: A real mode specifies a set of numerical values which approximate a continuous range of real numbers.

3.5.1 Floating point modes

syntax:

<floating point mode> ::= *(1)*
<floating point mode name> *(1.1)*

predefined names: The name *FLOAT* is predefined as a **floating point mode** name.

semantics: A floating point mode defines a set of numeric approximations to a range of real values, together with their minimum relative accuracy, between implementation defined bounds, over which the usual ordering and arithmetic operations are defined (see 5.3). This set contains only the values which can be represented by the implementation. An implementation may define other floating point modes with different bounds and/or **precision** (e.g. *LONG_FLOAT*, *SHORT_FLOAT*) that may also be used as **parent** modes for ranges (see 13.3). The *&FLOAT* mode is introduced as the virtual mode that contains all the values of all **predefined** floating point modes defined by the implementation. The internal representation of a floating point value is the floating point value itself. Note that *&FLOAT* is not a **predefined** mode (although it may have the same bounds as those of a **predefined** floating point mode).

static properties: A floating point mode has the following hereditary properties:

- An **upper bound** and a **lower bound** which are the literals denoting respectively the highest and lowest value defined by the floating point mode. They are implementation defined.
- A **precision** which is the maximum number of significant decimal digits defined by the mode.

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

- A **positive lower limit** and a **negative upper limit** which are the literals denoting respectively the smallest positive value and the largest negative value exactly representable in the floating point mode, zero excluded.

examples:

FLOAT *(1.1)*

3.5.2 Floating point range modes

syntax:

<floating point range mode> ::= *(1)*
<floating point mode name> (*<float value range>*) *(1.1)*
| **RANGE** (*<float value range>* [, *<significant digits>*]) *(1.2)*
| *<floating point range mode name>* *(1.3)*

<float value range> ::= *(2)*
<lower float bound> : *<upper float bound>* *(2.1)*

<lower float bound> :: = *(3)*
<floating point literal expression> *(3.1)*

<upper float bound> :: = *(4)*
<floating point literal expression> *(4.1)*

<significant digits> ::= *(5)*
<integer literal expression> *(5.1)*

semantics: A floating point range mode defines the set of values ranging between the bounds specified (bounds included) by *float value range* with the number of significant digits specified by *significant digits*. The range is taken from a specific **parent** mode that determines the operations on and ordering of the range values. For example, **RANGE** (–10.0*E*1 : 10.0*E*1, 2) denotes the values: –10.0, –9.9, …, –0.11, –0.1, 0, 0.1, …, 10.0.

static properties: A floating point range mode has the following non-hereditary property: it has a **parent** mode, defined as follows:

- If the floating point range mode is of the form:

  <floating point mode name> (*<float value range>*)

  then if the *floating point mode name* is not a floating point range mode, the **parent** mode is the *floating point mode name*; otherwise it is the **parent** mode of the *floating point mode name*.

- If the floating point range mode is of the form:

  RANGE (*<float value range>* [, *<significant digits>*])

  then the **parent** mode depends on the **resulting class** of the classes of the *upper float bound* and *lower float bound* in the *literal range*:

  – if it is an M-derived class, where M is a floating point mode, then the **parent** mode is a **predefined** floating point mode chosen by the implementation such that it contains the range of values delivered by *float value range*, with the **precision** defined below;

  – otherwise it is the **root** mode of the **resulting class**.

- If the floating point range mode is a *floating point range mode name* which is a **synmode** name, then its **parent** mode is that of the **defining** mode of the **synmode** name; otherwise it is a **newmode** name and then its **parent** mode is the virtually introduced **parent** mode (see 3.2.3).

A floating point range mode has the following hereditary properties:

- An **upper bound** and a **lower bound** which are the literals denoting the values delivered by *lower float bound* and *upper float bound*, respectively, in the *float value range*.

- A **precision** which is, if the floating point range mode is of the form:

  RANGE (*<float value range>* [, *<significant digits>*])

  – the value delivered by *significant digits* if specified;

  – otherwise the greatest **precision** of the **precisions** of *lower float bound* and *upper float bound*.

  Otherwise it is that of the *floating point mode name* or the *floating point range mode name*.

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

static conditions: *Lower float bound* must deliver a value that is less than or equal to the value delivered by *upper float bound*, and both values must belong to the set of values defined by *floating point mode name*, if specified.

There must exist a **predefined** floating point mode that contains both **upper bound** and **lower bound** with the specified **precision**.

The value delivered by *significant digit* must be greater than zero.

The evaluation of the 1.*lower float bound*, 2.*upper float bound*, must not depend directly or indirectly on the value of the 1.**lower bound**, 2.**upper bound** of the floating point range mode.

3.6 Powerset modes

syntax:

<powerset mode> ::= *(1)*
  POWERSET *<member mode>* *(1.1)*
  | *<powerset mode name>* *(1.2)*

<member mode> ::= *(2)*
  <discrete mode> *(2.1)*

semantics: A powerset mode defines values that are sets of values of its member mode. Powerset values range over all subsets of the member mode. The usual set-theoretic operators are defined on powerset values (see 5.3).

The maximum **number of values** of the member mode is implementation defined.

static properties: A powerset mode has the following hereditary property:

- A **member** mode which is the *member mode*.

examples:

8.4 **POWERSET** *CHAR* *(1.1)*

9.5 **POWERSET** *INT (2:max)* *(1.1)*

9.6 *number_list* *(1.2)*

3.7 Reference modes

3.7.1 General

syntax:

<reference mode> ::= *(1)*
  <bound reference mode> *(1.1)*
  | *<free reference mode>* *(1.2)*
  | *<row mode>* *(1.3)*

semantics: A reference mode defines references (addresses or descriptors) to **referable** locations. By definition, bound references refer to locations of a given static mode or a set of related moreta modes; free references may refer to locations of any static mode; rows refer to locations of a dynamic mode.

The dereferencing operation is defined on reference values (see 4.2.3, 4.2.4 and 4.2.5), delivering the location that is referenced.

Two reference values are equal if and only if they both refer to the same location, or both do not refer to a location (i.e. they are the value *NULL*).

3.7.2 Bound reference modes

syntax:

<bound reference mode> ::= *(1)*
  REF *<referenced mode>* *(1.1)*
  | *<bound reference mode name>* *(1.2)*

<referenced mode> ::= *(2)*
  <mode> *(2.1)*

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 9496:2003

semantics: A bound reference mode defines reference values to locations of the specified referenced mode.

If the referenced mode is a non-moreta mode M then the bound reference mode defines reference values to locations of M.

If the referenced mode is a moreta mode MM then the bound reference mode defines reference values to locations of MM or any successor of MM.

static properties: A bound reference mode has the following hereditary property:

- A **referenced** mode which is the *referenced mode*.

examples:

10.42 **REF** *cell* *(1.1)*

3.7.3 Free reference modes

syntax:

<free reference mode> ::= *(1)*
<free reference mode name> *(1.1)*

predefined names: The name *PTR* is predefined as a **free reference mode** name.

semantics: A free reference mode defines reference values to locations of any static mode.

examples:

19.8 *PTR* *(1.1)*

3.7.4 Row modes

syntax:

<row mode> ::= *(1)*
ROW *<string mode>* *(1.1)*
| **ROW** *<array mode>* *(1.2)*
| **ROW** *<variant structure mode>* *(1.3)*
| *<row mode name>* *(1.4)*

semantics: A row mode defines reference values to locations of dynamic mode (which are locations of some parameterized mode with non **constant** parameters).

A row value may refer to:

- string locations with non **constant string length**;
- array locations with non **constant upper bound**;
- parameterized structure locations with non **constant** parameters.

static properties: A row mode has the following hereditary property:

- A **referenced origin** mode which is the *string mode*, the *array mode*, or the *variant structure mode*, respectively.

static condition: The *variant structure mode* must be **parameterizable**.

examples:

8.6 **ROW CHARS** *(max)* *(1.1)*

3.8 Procedure modes

syntax:

<procedure mode> ::= *(1)*
PROC ([*<parameter list>*]) [*<result spec>*]
[**EXCEPTIONS** (*<exception list>*)] *(1.1)*
| *<procedure mode name>* *(1.2)*

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

<parameter list> ::= *(2)*
<parameter spec> { , *<parameter spec>*}* *(2.1)*

<parameter spec> ::= *(3)*
<mode> [*<parameter attribute>*] *(3.1)*

<parameter attribute> ::= *(4)*
IN | **OUT** | **INOUT** | **LOC** [**DYNAMIC**] *(4.1)*

<result spec> ::= *(5)*
RETURNS (*<mode>* [*<result attribute>*]) *(5.1)*

<result attribute>::= *(6)*
[**NONREF**] **LOC** [**DYNAMIC**] *(6.1)*

<exception list> ::= *(7)*
<exception name> { , *<exception name>*}* *(7.1)*

semantics: A procedure mode defines (**general**) procedure values, i.e. the objects denoted by **general procedure** names that are names defined in procedure definition statements. Procedure values indicate pieces of code in a dynamic context. Procedure modes allow for manipulating a procedure dynamically, e.g. passing it as a parameter to other procedures, sending it as message value to a buffer, storing it into a location, etc.

Procedure values can be called (see 6.7).

Two procedure values are equal if and only if they denote the same procedure in the same dynamic context, or if they both denote no procedure (i.e. they are the value *NULL*).

static properties: A procedure mode has the following hereditary properties:

- A list of **parameter specs**, each consisting of a mode and possibly a parameter attribute. The **parameter specs** are defined by the *parameter list*.
- An optional **result spec**, consisting of a mode and an optional result attribute. The **result spec** is defined by the *result spec*.
- A possibly empty list of **exception** names which are those mentioned in the *exception list*.

static conditions: All names mentioned in *exception list* must be different.

If **LOC** is specified in the *parameter spec* or in the *result spec*, the *mode* in it may have the **non-value property**.

If **DYNAMIC** is specified in the *parameter spec* or in the *result spec*, the *mode* in it must be **parameterizable**.

3.9 Instance modes

syntax:

<instance mode> ::= *(1)*
<instance mode name> *(1.1)*

predefined names: The name *INSTANCE* is predefined as an **instance mode** name.

semantics: An instance mode defines values which identify processes. The creation of a new process (see 5.2.15, 6.13 and 11.1) yields a unique instance value as identification for the created process.

Two instance values are equal if and only if they identify the same process, or they both identify no process (i.e. they are the value *NULL*).

examples:

15.39 INSTANCE *(1.1)*

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 9496:2003

3.10 Synchronization modes

3.10.1 General

syntax:

<synchronization mode> ::= *(1)*
<event mode> *(1.1)*
| *<buffer mode>* *(1.2)*

semantics: A synchronization mode provides a means for synchronization and communication between processes (see clause 11). There exists no expression in CHILL denoting a value defined by a synchronization mode. As a consequence, there are no operations defined on the values.

3.10.2 Event modes

syntax:

<event mode> ::= *(1)*
EVENT [(*<event length>*)] *(1.1)*
| *<event mode name>* *(1.2)*

<event length> ::= *(2)*
<integer literal expression> *(2.1)*

semantics: An event mode location provides a means for synchronization between processes. The operations defined on event mode locations are the continue action, the delay action and the delay case action, which are described in 6.15, 6.16 and 6.17, respectively.

The *event length* specifies the maximum number of processes that may become delayed on an event location; that number is unlimited if no *event length* is specified.

An event mode location which contains the **undefined** value is an "empty" event, i.e. no delayed processes are attached to it.

static properties: An event mode has the following hereditary property:

- An optional **event length** which is the value delivered by *event length*.

static conditions: The *event length* must deliver a positive value.

The evaluation of the *event length* must not depend directly or indirectly on the value of the **event length** of the event mode.

examples:

14.10 **EVENT** *(1.1)*

3.10.3 Buffer modes

syntax:

<buffer mode> ::= *(1)*
BUFFER [(*<buffer length>*)] *<buffer element mode>* *(1.1)*
| *<buffer mode name>* *(1.2)*

<buffer length> ::= *(2)*
<integer literal expression> *(2.1)*

<buffer element mode> ::= *(3)*
<mode> *(3.1)*

semantics: A buffer mode location provides a means for synchronization and communication between processes. The operations defined on buffer locations are the send action and the receive case action, described in 6.18 and 6.19, respectively.

The *buffer length* specifies the maximum number of values that can be stored in a buffer location; that number is unlimited if no *buffer length* is specified.

A buffer mode location which contains the **undefined** value is an "empty" buffer, i.e. no delayed processes are attached to it nor are there messages in the buffer.

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

static properties: A buffer mode has the following hereditary properties:

- An optional **buffer length** which is the value delivered by *buffer length*.
- A **buffer element** mode which is the *buffer element mode*.

static conditions: The *buffer length* must deliver a non-negative value.

The *buffer element mode* must not have the **non-value property**.

The evaluation of the *buffer length* must not depend directly or indirectly on the value of the **buffer length** of the buffer mode.

examples:

16.30 **BUFFER** *(1) user_messages* *(1.1)*

16.34 *user_buffers* *(1.2)*

3.11 Input-Output Modes

3.11.1 General

syntax:

<input-output mode> ::= *(1)*
<association mode> *(1.1)*
| *<access mode>* *(1.2)*
| *<text mode>* *(1.3)*

semantics: An input-output mode provides a means for input-output operations as defined in clause 7. There exists no expression in CHILL denoting a value defined by an input-output mode. As a consequence, there are no operations defined on the values.

examples:

20.17 *ASSOCIATION* *(1.1)*

3.11.2 Association modes

syntax:

<association mode> ::= *(1)*
<association mode name> *(1.1)*

predefined names: The name *ASSOCIATION* is predefined as an **association mode** name.

semantics: An association mode location provides a means for representing a relation to an outside world object. Such a relation is called an association in CHILL; associations can be created by the built-in routine *ASSOCIATE* and be ended by *DISSOCIATE*.

An association mode location which contains the **undefined** value is "empty", i.e. it does not contain an association.

3.11.3 Access modes

syntax:

<access mode> ::= *(1)*
ACCESS [(*<index mode>*)] [*<record mode>* [**DYNAMIC**]] *(1.1)*
| *<access mode name>* *(1.2)*

<record mode> ::= *(2)*
<mode> *(2.1)*

<index mode> ::= *(3)*

<discrete mode> *(3.1)*
| *<literal range>* *(3.2)*

derived syntax: The index mode notation *literal range* is derived from the discrete mode **RANGE** (*literal range*).

semantics: An access mode location provides a means for positioning a file and for transferring values from a CHILL program to a file in the outside world, and vice versa.

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 9496:2003

An access mode may define a *record mode*; this record mode defines the **root** mode of the class of the values that can be transferred via a location of that access mode to or from a file. The mode of the transferred value may be dynamic, i.e. the **size** of the record may vary, when the attribute **DYNAMIC** is specified in the access mode denotation or when *record mode* is a **varying** string mode. In the latter case **DYNAMIC** need not be specified.

An access mode may also define an *index mode*; such an index mode defines the size of a "window" to (a part of) the file, from which it is possible to read (or write) records randomly. Such a window can be positioned in an (**indexable**) file by the connect operation. If no *index mode* is specified, then it is possible to transfer records only sequentially.

An access mode location which contains the **undefined** value is "empty", i.e. it is not connected to an association.

static properties: An access mode has the following hereditary properties:

- An optional **record** mode which is the *record mode* if present. It is a **dynamic record** mode if **DYNAMIC** is specified or if *record mode* is a **varying** string mode, otherwise it is a **static record** mode.
- An optional **index** mode which is the *index mode*.
- Optional **upper bound** and **lower bound** which are the **upper bound** and **lower bound** of the *index mode*, if present.

static conditions: The optional *record mode* must not have the **non-value property**.

If **DYNAMIC** is specified, the **record** mode must be **parameterizable** and must not be a **tagless** structure mode.

The *index mode* must neither be a **numbered** set mode nor a **numbered** range mode.

If the *index mode* is a *literal range* of the form:

<lower bound> : *<upper bound>*

then, the evaluation of the 1.*lower bound*, 2.*upper bound*, must not depend directly or indirectly on the value of the 1.**lower bound**, 2.**upper bound** of the access mode.

examples:

20.18 **ACCESS** *(index_set) record_type* *(1.1)*

22.20 **ACCESS** *string* **DYNAMIC** *(1.1)*

20.18 *record_type* *(2.1)*

20.18 *index_set* *(3.1)*

3.11.4 Text modes

syntax:

<text mode> ::= *(1)*
<narrow text mode> *(1.1)*
| *<wide text mode>* *(1.2)*

<narrow text mode> ::= *(2)*
TEXT (*<text length>*) [*<index mode>*] [**DYNAMIC**] *(2.1)*

<wide text mode> ::= *(3)*
WTEXT (*<text length>*) [*<index mode>*] [**DYNAMIC**] *(3.1)*

<text length> ::= *(4)*
<integer literal expression> *(4.1)*

semantics: A text mode location provides a means for transferring values represented in human-readable form from a CHILL program to a file in the outside world, and vice versa. A text mode location has a **text record** sub-location and an **access** sub-location. The **text record** sub-location is initialized with an empty string.

A text mode has a **text length**, which defines the maximum length of the records that can be transferred, and possibly an **index** mode that has the same meaning as for access modes. The **actual length** attribute of a text mode location is the **actual length** of its **text record**.

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

A text mode location which contains the **undefined** value has a **text record** sub-location that contains the empty string and an **access** sub-location that contains the **undefined** value.

static properties: A text mode has the following hereditary properties:

- A **text length** which is the value delivered by *text length*.
- A **text record** mode which is **CHARS** (*<text length>*) **VARYING** in case of **TEXT** and which is **WCHARS** (*<text length>*) **VARYING** in case of **WTEXT**.
- It has an **access** mode which is **ACCESS** [(*<index mode>*)] **CHARS** (*<text length>*) [**DYNAMIC**] in case of **TEXT** and which is **WCHARS** (*<text length>*) [**DYNAMIC**] in case of **WTEXT** (*<index mode>* and **DYNAMIC** are part of the mode only if they are specified).
- Optional **upper bound** and **lower bound** which are the **upper bound** and **lower bound** of the *index mode*, if present.

static conditions: If the *index mode* is a *literal range* of the form:

<lower bound> : *<upper bound>*

then, the evaluation of the 1.*lower bound*, 2.*upper bound*, must not depend directly or indirectly on the value of the 1.**lower bound**, 2.**upper bound** of the text mode.

examples:

26.8 **TEXT** *(80)* **DYNAMIC** *(2.1)*

3.12 Timing modes

3.12.1 General

syntax:

<timing mode> ::= *(1)*
<duration mode> *(1.1)*
| *<absolute time mode>* *(1.2)*

semantics: A timing mode provides a means for time supervision of processes as described in clause 9. Timing values are created by a set of built-in routines. The relational operators are defined on timing values.

3.12.2 Duration modes

syntax:

<duration mode> ::= *(1)*
<duration mode name> *(1.1)*

predefined names: The name *DURATION* is predefined as a **duration mode** name.

semantics: A duration mode defines values which represent periods of time. The set of values defined by the duration mode is implementation defined. An implementation may choose to represent duration values as pairs of precision and value. Duration values are ordered in the intuitive way.

3.12.3 Absolute time modes

syntax:

<absolute time mode> ::= *(1)*
<absolute time mode name> *(1.1)*

predefined names: The name *TIME* is predefined as an **absolute time mode** name.

semantics: An absolute time mode defines values which represent points in time. The set of values defined by the absolute time mode is implementation defined. Absolute time values are ordered in the intuitive way.

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

3.13 Composite modes

3.13.1 General

syntax:

<composite mode> ::= *(1)*
<string mode> *(1.1)*
| *<array mode>* *(1.2)*
| *<structure mode>* *(1.3)*
| *<moreta mode>* *(1.4)*

semantics: A composite mode defines composite values, i.e. values consisting of sub-components which can be accessed or obtained (see 4.2.6-4.2.10 and 5.2.6-5.2.10).

3.13.2 String modes

syntax:

<string mode> ::= *(1)*
<string type> (*<string length>*) [**VARYING**] *(1.1)*
| *<parameterized string mode>* *(1.2)*
| *<string mode name>* *(1.3)*

<parameterized string mode> ::= *(2)*
<origin string mode name> (*<string length>*) *(2.1)*
| *<parameterized string mode name>* *(2.2)*

<origin string mode name> ::= *(3)*
<string mode name> *(3.1)*

<string type> ::= *(4)*
BOOLS *(4.1)*
| **CHARS** *(4.2)*
| **WCHARS** *(4.3)*

<string length> ::= *(5)*
<integer literal expression> *(5.1)*

semantics: A **fixed** string mode defines bit or character string values of a length indicated or implied by the string mode. A **varying** string mode defines bit or character string values whose **actual length** ranges from 0 to the **string length**. The length is known only at runtime from the value of the attribute **actual length**. For a **fixed** string mode the **actual length** is always equal to the **string length**. Character strings are sequences of character values; bit strings are sequences of boolean values.

String values are either empty or have string elements which are numbered from 0 upward.

The string values of a given string mode are totally-ordered in accordance with the ordering of the component values and the following definition.

Two strings s and t are equal if and only if they are empty or have the same length l and $s(i) = t(i)$ for all $0 \le i < l$. A string s precedes t when either:

- there exists an index j such that $s(j) < t(j)$ and $s(0 : j - 1) = t(0 : j - 1)$, or
- *LENGTH* (s) < *LENGTH* (t) and $s = t$(0 **UP** *LENGTH* (s)).

The concatenation operator is defined on string values. The usual logical operators are defined on bit string values and operate between their corresponding elements (see 5.3).

The maximum length of string modes is implementation defined.

static properties: A string mode has the following hereditary properties:

- A **string length** which is the value delivered by *string length*.
- An **upper bound** and a **lower bound** which are the values delivered by **string length** – 1 and 0, respectively.

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

- An **element** mode which is either *M* or **READ** *M*, where *M* is *BOOL*, *CHAR* or *WCHAR* depending on whether *string type* specifies **BOOLS**, **CHARS** or **WCHARS**, or the **element** mode of the *origin string mode name*, respectively. The **element** mode will be **READ** *M* if and only if the *string mode* is a **read-only** mode; in such case it is an implicit **read-only** mode.
- It is a **varying** string mode if **VARYING** is specified or if the *origin string mode name* denotes a **varying** string mode; otherwise it is a **fixed** string mode.

A string mode is **parameterized** if and only if it is a *parameterized string mode*.

A **parameterized** string mode has an **origin** string mode which is the mode denoted by *origin string mode name*.

A **varying** string mode has the following non-hereditary property: it has a **component** mode, defined as follows:

- If the **varying** string mode is of the form:

  <string type> (*<string length>*) **VARYING**

  then it is *<string type>* (*<string length>*).
- If the **varying** string mode is of the form:

  <origin string mode name> (*<string length>*)

  then the **component** mode is *&name* (*string length*), where *&name* is a virtually introduced **synmode** name **synonymous** with the **component** mode of the *origin string mode name*.
- If the **varying** string mode is a *string mode name* which is a **synmode** name, then its **component** mode is that of the **defining** mode of the **synmode** name; otherwise it is a **newmode** name and then its **component** mode is the virtually introduced **component** mode (see 3.2.3).

static conditions: The *string length* must deliver a non-negative value.

The value delivered by the *string length* directly contained in a *parameterized string mode* must be less than or equal to the **string length** of the *origin string mode name*. This condition applies only to the **parameterized** string modes that are not introduced virtually.

The evaluation of the *string length* must not depend directly or indirectly on the value of the **string length** of the string mode.

examples:

7.51 **CHARS** *(20)* *(1.1)*

22.22 **CHARS** *(20)* **VARYING** *(1.1)*

3.13.3 Array modes

syntax:

<array mode> ::= *(1)*
  ARRAY (*<index mode>* { , *<index mode>* }*)
  <element mode> { *<element layout>* }* *(1.1)*
| *<parameterized array mode>* *(1.2)*
| *<array mode name>* *(1.3)*

<parameterized array mode> ::= *(2)*
  <origin array mode name> (*<upper index>*) *(2.1)*
| *<parameterized array mode name>* *(2.2)*

<origin array mode name> ::= *(3)*
  <array mode name> *(3.1)*

<upper index> ::= *(4)*
  <discrete literal expression> *(4.1)*

<element mode> ::= *(5)*
  <mode> *(5.1)*

derived syntax: An *array mode* with more than one index mode (denoting a multi-dimensional array), is derived syntax for an *array mode* with an *element mode* that is an *array mode*. For example:

ARRAY (*1:20,1:10*) *INT*

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 9496:2003

is derived from:

ARRAY (**RANGE** *(1:20)*) **ARRAY** (**RANGE** *(1:10)*) *INT*

Only if this derived syntax is used, is more than one *element layout* occurrence allowed. The number of *element layout* occurrences must be less than or equal to the number of *index mode* occurrences. In that case, the leftmost *element layout* is associated with the innermost *element mode*, etc.

semantics: An array mode defines composite values, which are lists of values defined by its element mode. The physical layout of an array location or value can be controlled by *element layout* specification (see 3.13.5). Two array values are equal if and only if they have the same **number of elements** and the corresponding element values are equal.

The maximum **number of elements** of array modes is implementation defined.

static properties: An array mode has the following hereditary properties:

- An **index** mode which is the *index mode* if it is not a *parameterized array mode*, otherwise the **index** mode is the discrete range mode constructed as:

  &name (*lower bound* : *upper bound*)

  where *&name* is a virtual **synmode** name **synonymous** with the **index** mode of *origin array mode name*, *lower bound* is the lower bound of the **index** mode of the *origin array mode name* and *upper bound* is the *upper index*.

- An **upper bound** and a **lower bound** which are the **upper bound** and the **lower bound** of its **index** mode, respectively.

- An **element** mode which is either *M* or **READ** *M*, where *M* is the *element mode*, or the **element** mode of the *origin array mode name*, respectively. The **element** mode will be **READ** *M* if and only if *M* is not a **read-only** mode and the *array mode* is a **read-only** mode. The **element** mode is an implicit **read-only** mode if it is **READ** *M*.

- An **element layout** which, if it is a *parameterized array mode*, is the **element layout** of its *origin array mode name*; otherwise it is either the specified *element layout*, or the implementation default, which is either **PACK** or **NOPACK**.

- A **number of elements** which is the value delivered by:

  NUM (*upper bound*) – *NUM* (*lower bound*) + 1

  where *upper bound* and *lower bound* are respectively the **upper bound** and the **lower bound** of its **index** mode.

- It is a **mapped** mode if *element layout* is specified and is a *step*.

An array mode is **parameterized** if and only if it is a *parameterized array mode*.

A **parameterized** array mode has an **origin** array mode which is the mode denoted by *origin array mode name*.

static conditions: The class of *upper index* must be **compatible** with the **index** mode of the *origin array mode name* and the value delivered by it must lie in the range defined by that **index** mode.

If the array mode is a *parameterized array mode*, the evaluation of the *upper index* must not depend directly or indirectly on the value of the **upper bound** of the array mode. If the array mode is neither a *parameterized array mode* nor an *array mode name*, and if the *index mode* is a *literal range* of the form:

<lower bound> : *<upper bound>*

then, the evaluation of the 1.*lower bound*, 2.*upper bound*, must not depend directly or indirectly on the value of the 1.**lower bound**, 2.**upper bound** of the array mode.

examples:

5.27 **ARRAY** *(1:16)* **STRUCT** *(c4, c2, c1 BOOL)* *(1.1)*

11.12 **ARRAY** *(line)* **ARRAY** *(column) square* *(1.1)*

11.17 *board* *(1.3)*

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

3.13.4 Structure modes

syntax:

```
<structure mode> ::=                                                    (1)
        STRUCT ( <field> { , <field>}* )                                (1.1)
      | <parameterized structure mode>                                  (1.2)
      | <structure mode name>                                           (1.3)

<field> ::=                                                             (2)
        <fixed field>                                                   (2.1)
      | <alternative field>                                             (2.2)

<fixed field> ::=                                                       (3)
        <field name defining occurrence list> <mode> [ <field layout> ] (3.1)

<alternative field> ::=                                                 (4)
        CASE [ <tag list> ] OF
        <variant alternative> { , <variant alternative> }*
        [ ELSE [ <variant field> { , <variant field> }* ] ] ESAC        (4.1)

<variant alternative> ::=                                               (5)
        [ <case label specification> ] : [ <variant field> { , <variant field> }* ] (5.1)

<tag list> ::=                                                          (6)
        <tag field name> { , <tag field name> }*                        (6.1)

<variant field> ::=                                                     (7)
        <field name defining occurrence list> <mode> [ <field layout> ] (7.1)

<parameterized structure mode> ::=                                      (8)
        <origin variant structure mode name> ( <literal expression list> ) (8.1)
      | <parameterized structure mode name>                             (8.2)

<origin variant structure mode name> ::=                                (9)
        <variant structure mode name>                                   (9.1)

<literal expression list> ::=                                           (10)
        <discrete literal expression> { , <discrete literal expression> }* (10.1)
```

derived syntax: A *fixed field* occurrence or *variant field* occurrence, where *field name defining occurrence list* consists of more than one *field name defining occurrence*, is derived syntax for several *fixed field* occurrences or *variant field* occurrences with one *field name defining occurrence* respectively, each with the specified *mode* and optional *field layout*. In the case of *field layout*, this *field layout* must not be *pos*. For example:

STRUCT (*I,J BOOL* **PACK**)

is derived from:

STRUCT (*I BOOL* **PACK**, *J BOOL* **PACK**)

semantics: Structure modes define composite values consisting of a list of values, selectable by a component name. Each value is defined by a mode that is attached to the component name. Structure values may reside in (composite) structure locations, where the component name serves as an access to the sub-location. The components of a structure value or location are called fields and their names **field** names.

There are **fixed** structures, **variant** structures and **parameterized** structures.

Fixed structures consist only of fixed fields, i.e. fields that are always present and that can be accessed without any dynamic check.

Variant structures have variant fields, i.e. fields that are not always present. For **tagged variant** structures, the presence of these fields is known only at run time from the value(s) of certain associated fixed field(s) called **tag** fields. **Tag-less variant** structures do not have **tag** fields. Because the composition of a **variant** structure may change during run time, the **size** of a variant structure location is based upon the largest choice (worst case) of variant alternatives.

AVA Documents
Document Sharing Hub
Full document details: BS ISO/IEC 9496:2003

In an *alternative field* the *variant alternative* chosen is that for which values give in the case label specification match; if no value match, the *variant alternative* following **ELSE** (which will be present) is chosen.

A **parameterized** structure is determined from a **variant** structure mode for which the choice of variant alternatives is statically specified by means of **literal** expressions. The composition is fixed from the point of the creation of the parameterized structure and may not change during run time. The **tag** fields, if present, are **read-only** and automatically initialized with the specified values. For a parameterized structure location, a precise amount of storage can be allocated at the point of declaration or generation. Note that dynamic **parameterized** structure modes also exist; their semantics are defined in 3.14.4.

The layout of a structure location or value can be controlled by means of a field layout specification (see 3.13.5).

Two structure values are equal if and only if the corresponding component values are equal. However, if the structure values are **tag-less variant** structure values, the result of comparison is implementation defined.

For a mode with the **tagged parameterized property** the **undefined** value denotes a value in which **tag** field sub-values are equal to the corresponding parameter values and all the other ones are equal to the **undefined** value.

static properties:

general: A structure mode has the following hereditary properties:

- It is a **fixed** structure mode if it is a *structure mode* that does not directly contain an *alternative field* occurrence.
- It is a **variant** structure mode if it is a *structure mode* and contains at least one *alternative field* occurrence.
- It is a **parameterized** structure mode if it is a *parameterized structure mode*.
- It has a set of **field** names. This set is defined below for the different cases. A name is said to be a **field** name if and only if it is defined in a *field name defining occurrence list* in *fixed fields* or *variant fields* in a *structure mode*.

  Each *fixed field*, *variant field* and therefore each **field** name of a structure mode has a **field** mode attached that is either *M* or **READ** *M*, where *M* is the *mode* in the *fixed field* or *variant field*. The **field** mode is **READ** *M* if *M* is not a **read-only** mode and either the structure mode is a **read-only** mode, or the field is a **tag** field of a **parameterized** structure mode. The **field** mode is an implicit **read-only** mode if it is **READ** *M*.

  A *fixed field*, *variant field* and therefore a **field** name of a given structure mode has a **field layout** attached to it that is the *field layout* in the *fixed field* or *variant field*, if present; otherwise it is the default field layout, which is either **PACK** or **NOPACK**.
- It is a **mapped** mode if its **field** names have a *field layout* that is *pos*.

fixed structures: A **fixed** structure mode has the following hereditary property:

- A set of **field** names which is the set of names defined by any *field name defining occurrence list* in *fixed fields*. These **field** names are **fixed field** names.

variant structures: A **variant** structure mode has the following hereditary properties:

- A set of **field** names which is the union of the set of names defined by any field name defining occurrence list in fixed fields and the set of names defined by any field name defining occurrence list in alternative fields. **Field** names defined by a field name defining occurrence list in fixed fields are the **fixed field** names of the **variant** structure mode; its other **field** names are the **variant field** names.
- A **field** name of a **variant** structure mode is a **tag field** name if and only if it occurs in any tag list of an alternative field. Alternative fields in which no tag lists are specified are **tag-less** *alternative fields*.
- A **variant** structure mode is a **tag-less variant** structure mode if all its *alternative field* occurrences are **tag-less**. Otherwise it is a **tagged variant** structure mode.
- A **variant** structure mode is a **parameterizable variant** structure mode if it is either a **tagged variant** structure mode or a **tag-less variant** structure mode where for each of the *alternative field* occurrences a *case label specification* is given for all the *variant alternative* occurrences in it.

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

- A **parameterizable variant** structure mode has a list of classes attached, determined as follows:
  - if it is a **tagged variant** structure mode, the list of M_i – value classes, where M_i are the modes of the **tag field** names in the order that they are defined in *fixed fields*;
  - if it is a **tag-less variant** structure mode, the list is built up from the individual **resulting lists of classes** of each *alternative field* by concatenating them in the order as the *alternative fields* occur. The **resulting list of classes** of an *alternative field* occurrence is the **resulting list of classes** of the list of *case label specification* occurrences in it (see 12.3).

parameterized structures: A **parameterized** structure mode has the following hereditary properties:

- An **origin variant** structure mode which is the mode denoted by *origin variant structure mode name*.
- A set of **field** names which is the union of the set of **fixed field** names of its **origin variant** structure mode and the set of those **variant field** names of its **origin variant** structure mode that are defined in *variant alternative* occurrences that are selected by the list of values defined by *literal expression list*.
- The set of **tag field** names of a *parameterized structure mode* is the set of **tag field** names of its **origin variant** structure mode.
- A list of values attached, defined by *literal expression list*.
- It is a **tagged parameterized** structure mode if its **origin variant** structure mode is a **tagged variant** structure mode; otherwise the **parameterized** structure mode is **tag-less**.

For dynamic **parameterized** structure modes see 3.14.4.

static conditions:

general: All **field** names of a structure mode must be different.

If any field has a field layout which is *pos*, all the fields must have a field layout which must be *pos*.

variant structures: A **tag field** name must be a **fixed field** name and must be textually defined before all the *alternative field* occurrences in whose *tag list* it is mentioned. (As a consequence, a **tag** field precedes all the **variant** fields that depend upon it.) The mode of a **tag field** name must be a discrete mode.

The *mode* of *variant field* may have neither the **non-value property** nor the **tagged parameterized property**.

In a **variant** structure mode the *alternative field* occurrences must be either all **tagged** or all **tag-less**. For **tagged** *alternative fields*, *case label specification* must be specified in each *variant alternative*. For **tag-less** *alternative fields*, *case label specification* may be omitted in all *variant alternative* occurrences together, or must be specified for each *variant alternative* occurrence.

If, for a **tag-less variant** structure mode, any of its *alternative fields* has *case label specification* given, all its *alternative fields* must have *case label specification*.

For *alternative fields*, the case selection conditions must be fulfilled (see 12.3), and the same completeness, consistency and compatibility requirements must hold as for the case action (see 6.4). Each of the **tag field** names of *tag list* (if present) serves as a case selector with the M-value class, where M is the mode of the **tag field** name. In the case of **tag-less** alternative fields, the checks involving the case selector are ignored.

For a **parameterizable variant** structure mode none of the classes of its attached list of classes may be the **all** class. (This condition is automatically fulfilled by a **tagged variant** structure mode.)

parameterized structures: The *origin variant structure mode name* must be **parameterizable**.

There must be as many **literal** expressions in the *literal expression list* as there are classes in the list of classes of the *origin variant structure mode name*. The class of each **literal** expression must be **compatible** with the corresponding (by position) class of the list of classes. If the latter class is an M-value class, the value delivered by the **literal** expression must be one of the values defined by M.

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

examples:

3.3 **STRUCT** *(re, im INT)* *(1.1)*

11.7 **STRUCT** *(status* **SET** *(occupied, free),*
CASE *status* **OF**
(occupied): p piece,
(free):

ESAC*)* *(1.1)*

2.6 *fraction* *(1.3)*

11.7 *status* **SET** *(occupied, free)* *(3.1)*

11.8 *status* *(6.1)*

11.9 *p piece* *(7.1)*

3.13.5 Layout description for array modes and structure modes

syntax:

<element layout> ::= *(1)*
PACK | **NOPACK** | *<step>* *(1.1)*

<field layout> ::= *(2)*
PACK | **NOPACK** | *<pos>* *(2.1)*

<step> ::= *(3)*
STEP (*<pos>* [, *<step size>*]) *(3.1)*

<pos> ::= *(4)*
POS (*<word>* , *<start bit>* , *<length>*) *(4.1)*
| **POS** (*<word>* [, *<start bit>* [: *<end bit>*]]) *(4.2)*

<word> ::= *(5)*
<integer literal expression> *(5.1)*

<step size> ::= *(6)*
<integer literal expression> *(6.1)*

<start bit> ::= *(7)*
<integer literal expression> *(7.1)*

<end bit> ::= *(8)*
<integer literal expression> *(8.1)*

<length> ::= *(9)*
<integer literal expression> *(9.1)*

semantics: It is possible to control the layout of an array or a structure by giving packing or mapping information in its mode. Packing information is either **PACK** or **NOPACK**, mapping information is either *step* in the case of array modes, or *pos* in the case of structure modes. The absence of *element layout* or *field layout* in an array or structure mode will always be interpreted as packing information, i.e. either as **PACK** or as **NOPACK**.

If **PACK** is specified for elements of an array or fields of a structure, it means that the use of memory space is optimized for the array elements or structure fields, whereas **NOPACK** implies that the access time for the array elements or the structure fields is optimized. **NOPACK** also implies **referable**.

The **PACK**, **NOPACK** information is applied only for one level, i.e. it is applied to the elements of the array or fields of the structure, not for possible components of the array element or structure field. The layout information is always attached to the nearest mode to which it may apply and which does not already have layout attached. For example, if the default packing is **NOPACK**:

STRUCT (*f* **ARRAY** (*0:1*) *m* **PACK**)

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

is equivalent to:

STRUCT (*f* **ARRAY** (*0:1*) *m* **PACK NOPACK**)

It is also possible to control the precise layout of an array or a structure by specifying positioning information for its components in the mode. This positioning information is given in the following ways:

- For array modes, the positioning information is given for all elements together, in the form of a *step* following the array mode.
- For structure modes, the positioning information is given for each field individually, in the form of a *pos*, following the mode of the field.

Mapping information with *pos* is given in terms of word and bit-offsets. A *pos* of the form:

POS (*<word>* , *<start bit>* , *<length>*)

defines a bit-offset of

NUM (*word*) * WIDTH + *NUM* (*start bit*)

and a length of *NUM* (*length*) bits, where *WIDTH* is the (implementation defined) number of bits in a word, and *word* is an *integer literal expression*.

When *pos* is specified in *field layout* it defines that the corresponding field starts at the given bit-offset from the start of each location of the structure mode, and occupies the given length.

A *step* of the form:

STEP (*<pos>* , *<step size>*)

defines a series of bit-offsets b_i for i taking values 0 to n–1 where n is the **number of elements** in the array, and

$b_i = i$ * *NUM* (*step size*)

The j-th element of the array starts at a bit-offset of $p + b_j$ from the start of each location of the array mode, where p is the bit-offset specified in *pos*. Each element occupies the length given in *pos*.

Defaults

The notation:

POS (*<word>* , *<start bit>* : *<end bit>*)

is semantically equivalent to:

POS (*<word>* , *<start bit>* , *NUM (<end bit>) – NUM (<start bit>*) + 1)

The notation:

POS (*<word>* , *<start bit>*)

is semantically equivalent to:

POS (*<word>* , *<start bit>* , *BSIZE*)

where *BSIZE* is the minimum number of bits which is needed to be occupied by the component for which the *pos* is specified.

The notation:

POS (*<word>*)

is semantically equivalent to:

POS (*<word>* , *0* , *BSIZE*)

The notation:

STEP (*<pos>*)

is semantically equivalent to

STEP (*<pos>* , *SSIZE*)

where *SSIZE* is the *<length>* specified in *pos* or derivable from *pos* by the above rules.

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 9496:2003

static properties: For any location of an array mode the element layout of the mode determines the referability of its sub-locations (including sub-arrays, array slices) as follows:

- either all sub-locations are **referable**, or none of them are;
- if the element layout is **NOPACK** all sub-locations are **referable**.

For any location of a structure mode, the referability of the structure field selected by a **field** name is determined by the field layout of the **field** name as follows:

- the **field** name is **referable** if the field layout is **NOPACK**.

static conditions: If the **element** mode of a given array mode or the **field** mode of a **field** name of a given structure mode, is itself an array or structure mode, then it must be a **mapped** mode if the given array or structure mode is **mapped**.

NUM (*word*), *NUM* (*start bit*), *NUM* (*end bit*), *NUM* (*length*) and *NUM* (*step size*) ≥ 0;

NUM (*start bit*) and *NUM* (*end bit*) ≤ *WIDTH*; *NUM* (*start bit*) ≤ *NUM* (*end bit*).

Each implementation defines for each mode a minimum number of bits its values need to occupy; call this the minimum bit occupancy. For discrete modes it is any number of bits not less than log to the base two of the **number of values** of the mode. For array modes it is the offset of the element of the highest index plus its occupied bits. For structure modes it is the offset of the highest bit occupied.

For each *pos* the *length* specified must not be less than the minimum bit occupancy of the mode of the associated field or array components.

For each **mapped** array mode the *step size* must not be less than the *length* given or implied in the *pos*.

Consistency and feasibility

Consistency: No component of a structure may be specified such that it occupies any bits occupied by another component of the same object except in the case of two **variant field** names defined in the same *alternative field* occurrence; however, in the latter case the **variant field** names may not both be defined in the same *variant alternative* nor both following **ELSE**.

Feasibility: There are no language defined feasibility requirements, except for the one that can be deduced from the rule that the referability of a sub-location of any (**referable** or non-**referable**) location is determined only by the (element or field) layout, which is a property of the mode of the location. This places some restrictions on the mapping of components that themselves have **referable** components.

examples:

17.5 **PACK** *(1.1)*

19.14 **POS** *(1,0:15)* *(4.2)*

3.14 Dynamic modes

3.14.1 General

A dynamic mode is a mode of which some properties are known only at run time. Dynamic modes are always parameterized modes with one or more run-time parameters. For description purposes, virtual denotations are introduced in this Recommendation | International Standard. These virtual denotations are preceded by the ampersand symbol (&) to distinguish them from actual notations which appear in a CHILL program text.

3.14.2 Dynamic string modes

virtual denotation: &*<origin string mode name>* (*<integer expression>*)

semantics: A dynamic string mode is a parameterized string mode with non **constant** length.

static properties: Dynamic string modes have the same properties as string modes, except for the properties described below.

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

dynamic properties:

- A dynamic string mode has a dynamic **string length** which is the value delivered by *integer expression*.
- A dynamic string mode has an **upper bound** and a **lower bound** which are the values delivered by **string length** –1 and 0, respectively.

3.14.3 Dynamic array modes

virtual denotation: *&<origin array mode name>* (*<discrete expression>*)

semantics: A dynamic array mode is a parameterized array mode with non **constant upper bound**.

static properties: Dynamic array modes have the same properties as array modes, except for the properties described below.

dynamic properties:

- A dynamic array mode has a dynamic **upper bound** which is the value delivered by *discrete expression*, and a dynamic **number of elements** which is the value delivered by:

  NUM (*discrete expression*) – *NUM* (*lower bound*) + 1

  where *lower bound* is the **lower bound** of the *origin array mode name*.

3.14.4 Dynamic parameterized structure modes

virtual denotation: *&<origin variant structure mode name>* (*<expression list>*)

semantics: A dynamic **parameterized** structure mode is a **parameterized** structure mode with non **constant** parameters.

static properties: The static properties of a dynamic **parameterized** structure mode are those of a static **parameterized** structure mode except for the following:

- The set of **field** names of a dynamic **parameterized** structure mode is the set of **field** names of its **origin variant** structure mode.

dynamic properties:

- A dynamic **parameterized** structure mode has a list of values attached that is the list of values delivered by the expressions in the *expression list*.

3.15 Moreta Modes

3.15.1 General

syntax:

<moreta mode> ::=		*(1)*
	<module mode>	*(1.1)*
	| *<region mode>*	*(1.2)*
	| *<task mode>*	*(1.3)*
	| *<generic moreta mode instantiation>*	*(1.4)*
	| *<interface mode>*	*(1.5)*
	| *<moreta mode name>* [(*<actual parameter list>*)]	*(1.6)*

semantics:

module mode – A location of *module mode* has the same properties as a *module* without an *action statement list*.

region mode – A location of *region mode* has the same properties as a *region*.

task mode – A location of *task mode* has essentially the same structure as a *module mode location without process definitions*. The direct access to the components of a location, whose mode is a *task mode*, is mutually exclusive. A location, whose mode is a *task mode*, may be executed concurrently with other threads (see 11.1).

generic moreta mode instantiation – A generic moreta mode instantiation is obtained statically by an instantiation of a generic moreta mode template (see 10.11).

interface mode – An *interface mode* consists of specifications and signatures only.

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 9496:2003

static conditions:

Moreta modes are not parameterizable.

Moreta modes and *generic moreta mode templates* cannot be nested.

(1.1) – (1.5) are only allowed in synmode and newmode definitions, i.e. anonymous moreta modes are not allowed.

3.15.2 Module Modes

syntax:

```
<module mode> ::=                                                    (1)
        <module mode specification>                                  (1.1)
      | <module mode body>                                           (1.2)

<module mode specification> ::=                                      (2)
        MODULE SPEC [ [ ASSIGNABLE [ FINAL ] | ABSTRACT ] |
        [ NOT_ASSIGNABLE [ ABSTRACT | FINAL ]]]
        <module inheritance clause>
        {<module specification component>}* [<invariant part>]
        END [<simple name string>]                                   (2.1)

<module mode body> ::=                                               (3)
        MODULE BODY [ [ ASSIGNABLE [ FINAL ] | ABSTRACT ] |
        [ NOT_ASSIGNABLE [ ABSTRACT | FINAL ]]]
        <module inheritance clause>
        {<module body component>}* [<invariant part>]
        END [ <handler> ] [<simple name string>]                     (3.1)

<module inheritance clause> ::=                                      (4)
        [<module inheritance>] [<implementation clause>]             (4.1)

<module inheritance> ::=                                             (5)
        BASED_ON <module mode name>                                  (5.1)

<implementation clause> ::=                                          (6)
        IMPLEMENTS <interface mode name> { , <interface mode name> }*  (6.1)

<module specification component> ::=                                 (7)
        <common module component>                                    (7.1)
      | <declaration statement>                                      (7.2)
      | <simple guarded procedure signature statement>               (7.3)
      | <inline guarded procedure definition statement>              (7.4)
      | <process specification statement>                            (7.5)
      | <signal definition statement>                                (7.6)
      | <grant statement>                                            (7.7)

<module body component> ::=                                          (8)
        <common module component>                                    (8.1)
      | <simple guarded procedure definition statement>              (8.2)
      | <process definition statement>                               (8.3)

<common module component> ::=                                        (9)
        <synonym definition statement>                               (9.1)
      | <synmode definition statement>                               (9.2)
      | <newmode definition statement>                               (9.3)
      | <seize statement>                                            (9.4)

<invariant part> ::=                                                 (10)
        INVARIANT <boolean expression>                               (10.1)
```

semantics: A module mode defines composite values consisting of a list of components selectable by component names.

Module values may reside in (composite) **module** locations.

A *module mode* is defined by giving two separate parts: a *module mode specification* and a *module mode body*.

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003

The **specification** part defines the interface of the values of a *module mode*.

The **body** part defines the behaviour of the values of a *module mode*.

The *boolean expression* of the *invariant part* must be true before and after any call of a **public** component procedure or a **public** component process.

static properties: If the attribute **ASSIGNABLE** is specified, the mode is an **assignable** module mode. An **assignable** module mode can be used in the same way as a mode for which **READ** is not specified (see 3.3).

If the attribute **NOT_ASSIGNABLE** is specified, the mode has the **not_assignable** property, indicating that the location of that mode may not be accessed to store the value and may not be accessed to copy its value.

If neither **ASSIGNABLE** nor **NOT_ASSIGNABLE** is specified the mode is **not_assignable** by default.

If the attribute **ABSTRACT** is specified, the mode is an **abstract** mode.

If a *module inheritance* is given, the mode MD being defined is immediately derived from the mode MB given in the *module inheritance*, and MB is an immediate base mode of MD.

If an *implementation clause* IC is given, the mode MD being defined is immediately derived from the modes given in IC, and these modes are immediate base modes of MD.

The effect of the *module inheritance clause* is that the derived mode behaves as if it contained all components of its immediate base modes except for the constructor and destructor component procedures of these base modes. If any of these base modes is itself a derived mode, this inheritance of components is to be understood in a transitive manner. For visibility see 12.2.

A *module specification component* contained in a *module mode specification* M_S or SEIZEd into M_S, which is granted by M_S, is called a **public** component of the mode of M_S.

A *module specification component* contained in a *module mode specification* M_S or SEIZEd into M_S, which is not granted by M_S, is called an **internal** component of the mode of M_S.

A *module body component* C contained in a *module mode body* M_B or SEIZEd into M_B, is called a **private** component of the mode of M_B if C is neither a **public** nor an **internal** component of the mode of M_B.

An **abstract** module mode has the property **not_assignable**.

static conditions: A module mode cannot be used as the mode in a *synonym definition*.

For each *module mode specification*, there must be one *module mode body* with the same name string in the *defining occurrence*.

If specified, the *simple name string* after **END** must be equal to the name string of the defining occurrence of this mode definition. This holds for *module mode specification* and for *module mode body*.

If one of the attributes **ASSIGNABLE**, **NOT_ASSIGNABLE**, **ABSTRACT** or **FINAL** is specified in a *module mode specification*, it must also be specified in the corresponding *module mode body*.

If a *module mode specification* contains a *module inheritance clause*, the corresponding *module mode body* must contain the same *module inheritance clause*.

If the attribute **INCOMPLETE** (see 10.4) is specified in a *simple guarded procedure signature* then this procedure has the property **incomplete**.

If the attribute **INCOMPLETE** (see 10.4) is specified in a *simple guarded procedure signature statement* this procedure must be **public**.

For each **simple**, **complete** *guarded procedure signature statement* S of a *module mode specification*, the corresponding *module mode body* must contain a corresponding *simple guarded procedure definition statement* D, where the *guarded procedure signature* of S matches the *guarded procedure definition* of D (see 12.1.3).

If P is a **simple, incomplete** *guarded procedure signature* of a *module mode specification*, the corresponding *module mode body* must not contain a *simple guarded procedure definition* matching P.

For each *process specification* of a *module mode specification*, the corresponding *module mode body* must contain a corresponding *process definition* (see 12.1.3).

AVA Documents
Document Sharing Hub

Full document details: BS ISO/IEC 9496:2003