X/Open Preliminary Specification

Architecture Neutral Distribution Format (XANDF)

X/Open Company Ltd.

X/Open Preliminary Specification

Architecture Neutral Distribution Format (XANDF)

ISBN: 1-85912-146-2

X/Open Document Number: P527

Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.org

Contents

Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

X/Open defines this CAE in a set of specifications which include an evolving portfolio of application programming interfaces (APIs) which significantly enhance portability of application programs at the source code level, along with definitions of and references to protocols and protocol profiles which significantly enhance the interoperability of applications and systems.

The X/Open CAE is implemented in real products and recognised by a distinctive trade mark — the X/Open brand — that is licensed by X/Open and may be used on products which have demonstrated their conformance.

X/Open Technical Publications

X/Open publishes a wide range of technical literature, the main part of which is focussed on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

There are two types of X/Open specification:

- *CAE Specifications*

  CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement an X/Open CAE specification can enjoy the benefits of a single, widely supported standard. In addition, they can demonstrate compliance with the majority of X/Open CAE specifications once these specifications are referenced in an X/Open component or profile definition and included in the X/Open branding programme.

  CAE specifications are published as soon as they are developed, not published to coincide with the launch of a particular X/Open brand. By making its specifications available in this way, X/Open makes it possible for conformant products to be developed as soon as is practicable, so enhancing the value of the X/Open brand as a procurement aid to users.

- *Preliminary Specifications*

  These specifications, which often address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations, are released in a controlled manner for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification. In fact, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE specification.

  Preliminary specifications are analogous to the *trial-use* standards issued by formal standards organisations, and product development teams are encouraged to develop products on the basis of them. However, because of the nature of the technology that a Preliminary specification is addressing, it may be untried in multiple independent implementations, and may therefore change before being published as a CAE specification. There is always the intent to progress to a corresponding CAE specification, but the ability to do so depends on consensus among X/Open members. In all cases, any resulting CAE specification is made as upwards-compatible as possible. However, complete upwards-compatibility from the Preliminary to the CAE specification cannot be guaranteed.

In addition, X/Open publishes:

- *Guides*

  These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

- *Technical Studies*

  X/Open Technical Studies present results of analyses performed by X/Open on subjects of interest in areas relevant to X/Open's Technical Programme. They are intended to communicate the findings to the outside world and, where appropriate, stimulate discussion and actions by other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of an X/Open technical activity. Although at the time of its publication, there may be an intention to progress the activity towards publication of a Specification, Guide or Technical Study, X/Open is a consensus organisation, and makes no commitment regarding future development and further publication. Similarly, a Snapshot does not represent any commitment by X/Open members to develop any specific products.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

- a new *Version* indicates that this publication includes all the same (unchanged) definitive information from the previous publication of that title, but also includes extensions or additional information. As such, it *replaces* the previous publication.

- a new *Issue* does include changes to the definitive information contained in the previous publication of that title (and may also include extensions or additional information). As such, X/Open maintains *both* the previous and new issue as current publications.

Corrigenda

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done in any one of the following ways:

- anonymous ftp to ftp.xopen.org
- ftpmail (see below)
- reference to the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information using ftpmail, send a message to ftpmail@xopen.org with the following four lines in the body of the message:

```
open
cd pub/Corrigenda
get index
quit
```

This will return the index of publications for which Corrigenda exist. Use the same email address to request a copy of the full corrigendum information following the email instructions.

This Document

This document provides specifications for each construct in the XANDF format, and some general notes on various aspects of XANDF.

The Architecture Neutral Distribution Format (ANDF) is a software porting technology making it possible to develop shrink-wrapped software for open systems, independent of any particular processor architecture. ANDF intends to reduce the effort needed for porting of applications while at the same time making it possible to fully utilise the particular features of a platform.

The XANDF specification defines an integration interface between the two major components of a multi-platform cross-compilation system. The compilation of the source code is turned into a two stage process. In the first stage, the application is transcribed into a format which utilises generalised declarations of the API calls used, together with generalised definitions of data types, constants, etc. This is the Architecture Neutral Distribution Format and the piece of software producing it is called ANDF producer. XANDF is in fact an abstract algebra.

This format has a value in itself, even if the second phase of the compilation is never entered. It can namely be examined to determine how portable the code really is, through comparing it against lists of standardised API calls, and issuing warnings when such a search fails. X/Open is currently examining the possibilities offered by these features to its testing programme.

In the second phase of the compilation, the entities generated in the first phase are first linked together and then mapped onto a concrete machine through the use of processor-specific libraries implementing the API calls and data formats. The software performing this task is called ANDF Installer.

XANDF tokens (see Chapter 9) offer a general encapsulation and expansion mechanism which allows any implementation detail to be delayed to the most appropriate stage of program translation. This provides a means for encapsulating any target dependencies in a neutral form, with specific implementations defined through standard XANDF features. The token register records the names and specifications of tokens, using a consistent naming scheme to avoid ambiguity between tokens.

Intended Audience

application writers who wish to examine the portability of their code will be interested in ANDF Producer.

Suppliers and Users who want the convenience of running shrink-wrapped applications will be interested in ANDF Installer.

This specification is also intended for readers who are aware of the general background to XANDF but require more detailed information for implementation purposes.

Structure

- **Chapter 1** explains the positioning and purpose of XANDF.
- **Chapter 2** explains the basic structure of XANDF, its use of tokens and tags, and its extensibility.
- **Chapter 3** describes XANDF structure representation.
- **Chapter 4** describes the behaviour of XANDF installers.
- **Chapter 5** gives the definitions for all the XANDF constructs.
- **Chapter 6** gives the definitions for supplementary UNITs.
- **Chapter 7** gives implementation notes and guidance.
- **Chapter 8** explains the bit-encoding of XANDF.
- **Chapter 9** explains the XANDF token register.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for commands, keywords, type names, and data structures.
- *Italic* font is used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - — command operands, command option-arguments or variable names
  - — environment variables
- Normal font is used for the names of constants and literals.
- Names surrounded by braces { } represent the set containing what is inside the braces.

Trade Marks

UNIX® is a registered trade mark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open® is a registered trade mark, and the ''X'' device is a trade mark, of X/Open Company Limited.

Acknowledgements

The source for this XANDF Specification and its companion XANDF Guide was contributed by the United Kingdom Defence Evaluation and Research Agency (DERA).

Referenced Documents

The following documents are referenced in this Specification:

XANDF_G
X/Open Guide, January 1996, Architecture Neutral Distribution Format (XANDF) (ISBN: 1-85912-141-1, G508).

Chapter 1

Introduction

1.1 Scope and Purpose

The Architecture Neutral Distribution Format (ANDF) is a software porting technology making it possible to develop shrink-wrapped software for open systems, independent of any particular processor architecture. ANDF intends to reduce the effort needed for porting of applications while at the same time making it possible to fully utilise the particular features of a platform.

The XANDF specification defines an integration interface between the two major components of a multi-platform cross-compilation system. The compilation of the source code is turned into a two stage process. In the first stage, the application is transcribed into a format which utilises generalised declarations of the API calls used, together with generalised definitions of data types, constants, and so on. This is the Architecture Neutral Distribution Format and the piece of software producing it is called ANDF producer. XANDF is in fact an abstract algebra.

This format has a value in itself, even if the second phase of the compilation is never entered. It can be examined to determine how portable the code really is, through comparing it against lists of standardised API calls, and issuing warnings when such a search fails. X/Open is currently examining the possibilities offered by these features to its testing programme.

In the second phase of the compilation, the entities generated in the first phase are first linked together and then mapped onto a concrete machine through the use of processor-specific libraries implementing the API calls and data formats. The software performing this task is called ANDF Installer.

While ANDF producers are used by application writers, installers are provided on systems intending to offer the customers the convenience of running shrink-wrapped applications.

1.2 This Specification

This document provides specifications for each construct in the XANDF format and some general notes on various aspects of XANDF. It is intended for readers who are aware of the general background to XANDF but require more detailed information.

1.3 Terminology

The following terms are used with a precise meaning which is particular to this document:

compiling
The production of ANDF from some source language.

producing
Same meaning as ‘‘compiling’’.

translating
Making a program for some specific platform from ANDF.

1.4 Using XANDF for Porting

Software vendors, when they port their programs to several platforms, usually wish to take advantage of the particular features of each platform. That is, they wish the versions of their programs on each platform to be functionally equivalent, but not necessarily algorithmically identical. XANDF is intended for porting in this sense. It is designed so that a program in its XANDF form can be systematically modified when it arrives at the target platform to achieve the intended functionality and to use the algorithms and data structures which are appropriate and efficient for the target machine. A fully efficient program, specialised to each target, is a necessity if independent software vendors are to take up a porting technology.

These modifications are systematic because, on the source machine, programmers work with generalised declarations of the APIs they are using. The declarations express the requirements of the APIs without giving their implementation. The declarations are specified in terms of XANDF’s tokens, and the XANDF which is produced contains uses of these tokens. On each target machine the tokens are used as the basis for suitable substitutions and alterations.

Using XANDF for porting places extra requirements on software vendors and API designers. Software vendors must write their programs scrupulously in terms of APIs and nothing more. API designers need to produce an interface which can be specialised to efficient data structures and constructions on all relevant machines.

XANDF is neutral with respect to the set of languages which has been considered. The design of C, C++, Fortran and Pascal is quite conventional, in the sense that they are sufficiently similar for XANDF constructions to be devised to represent them all. These XANDF constructions can be chosen so that they are, in most cases, close to the language constructions. Other languages, such as Lisp, are likely to need a few extensions. To express novel language features XANDF will probably have to be more seriously extended. But the time to do so is when the feature in question has achieved sufficient stability. Tokens can be used to express the constructs until the time is right. For example, there is a lack of consensus about the best constructions for parallel languages, so that at present XANDF would either have to use low level constructions for parallelism or back what might turn out to be the wrong system. In other words it is not yet the time to make generalisations for parallelism as an intrinsic part of XANDF.

XANDF is neutral with respect to machine architectures. In designing XANDF, the aim has been to retain the information which is needed to produce and optimise the machine code, while discarding identifier and syntactic information. So XANDF has constructions which are closely related to typical language features and it has an abstract model of memory. It is expected that programs expressed in the considered languages can be translated into code which is as efficient as that produced by native compilers for those languages.

Because of these porting features, XANDF supports shrink-wrapping, distribution and installation. Installation does not have to be left to the end-user; the production of executables can be done anywhere in the chain from software vendor, through dealer and network manager to the end-user.

Chapter 2

Structure of XANDF

Each piece of XANDF program is classified as being of a particular SORT. Some pieces of are LABELs, some are TAGs, some are ERROR_TREATMENTs and so on (to list some of the more transparently named SORTs). The SORTs of the arguments and result of each construct of the XANDF format are specified. For instance, *plus* is defined to have three arguments — an ERROR_TREATMENT and two EXPs (short for ‘‘expression’’) — and to produce an EXP; *goto* has a single LABEL argument and produces an EXP. The specification of the SORTs of the arguments and results of each construct constitutes the syntax of the XANDF format. When XANDF is represented as a parsed tree, it is structured according to this syntax. When it is constructed and read, it is in terms of this syntax.

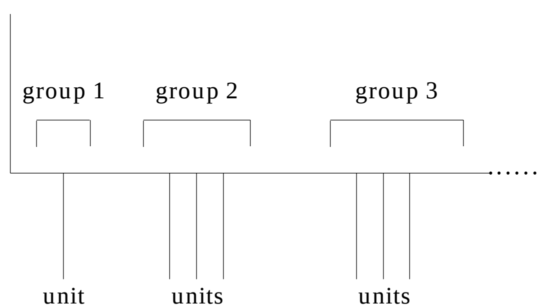
2.1 Overall Structure

A separable piece of XANDF is called a CAPSULE. A producer generates a CAPSULE; the XANDF linker links CAPSULEs together to form a CAPSULE, and the final translation process turns a CAPSULE into an object file.

The structure of capsules is designed so that the process of linking two or more capsules consists almost entirely of copying large byte-aligned sections of the source files into the destination file, with out changing or even examining these sections. Only a small amount of interface information has to be modified and this is made easily accessible. The translation process only requires an extra indirection to account for this interface information, so it is also fast. The description of XANDF at the capsule level is almost all about the organisation of the interface information.

There are three major kinds of entity which are used inside a capsule to name its constituents. The first are called tags: they are used to name the procedures, functions, values and variables which are the components of the program. The second are called tokens: they identify pieces of XANDF which can be used for substitution — a little like macros. The third are the alignment tags, used to name alignments so that circular types can be described. Because these internal names are used for linking pieces of XANDF together, they are collectively called *linkable entities*. The interface information relates these linkable entities to each other and to the world outside the capsule.

The most important part of a capsule, the part which contains the real information, consists of a sequence of groups of units. Each group contains units of the same kind, and all the units of the same kind are in the same group. The groups always occur in the same order, though it is not necessary for each kind to be present.



The order is as follows:

- **tld** unit.

  Every capsule has exactly one **tld** unit. It gives information to the XANDF linker about those items in the capsule which are visible externally.

- **versions** unit.

  These units contain information about the versions of XANDF used. Every capsule will have at least one such unit.

- **tokdec** units.

  These units contain declarations for tokens. They bear the same relationship to the following **tokdef** units that C declarations do to C definitions. However, they are not necessary for the translator, and the current ANSI C producer does not provide them.

- **tokdef** units.

  These units contain definitions of tokens.

- **aldef** units.

  These units give the definitions of alignment tags.

- **diagtype** units.

  These units give diagnostic information about types.

- **tagdec** units.

  These units contain declarations of tags, which identify values, procedures and run-time objects in the program. The declarations give information about the size, alignment and other properties of the values. They bear the same relationship to the following tagdef units that C declarations do to C definitions.

- **diagdef** units.

  These units give diagnostic information about the values and procedures defined in the capsule.

- **tagdef** units.

  These units contain the definitions of tags, and so describe the procedures and the values they manipulate.

- **linkinfo** units.

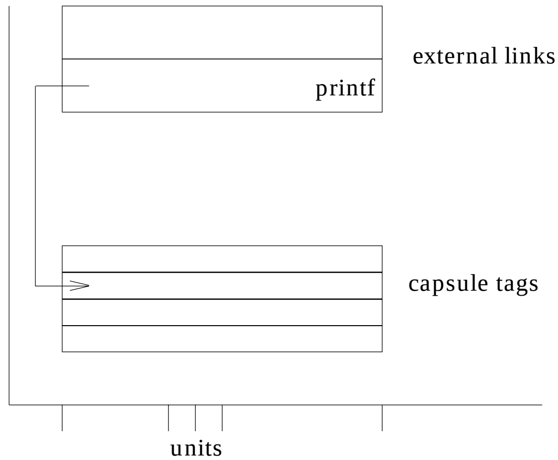
  These units give information about the linking of objects.

This organisation is imposed to help installers, by ensuring that the information needed to process a unit has been provided before that unit arrives. For example, the token definitions occur before any tag definition, so that, during translation, the tokens may be expanded as the tag definitions are being read[1].

The tags and tokens in a capsule have to be related to the outside world. For example, there might be a tag standing for **printf** used in the appropriate way inside the capsule. When an

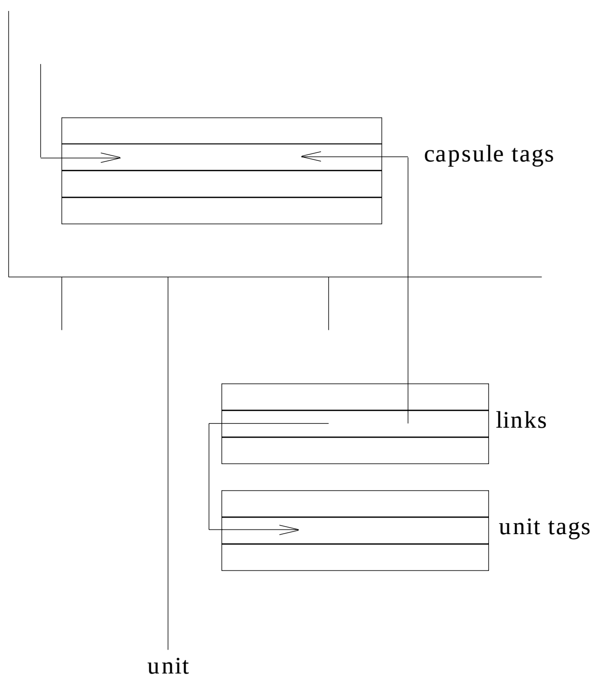
1. In a capsule which is ready for translation all tokens used must be defined, but this need not apply to an arbitrary capsule.

object file is produced from the capsule the identifier **printf** must occur in it, so that the system linker can associate it with the correct library procedure. In order to do this, the capsule has a table of tags at the capsule level, and a set of external links which provide external names for some of these tags.



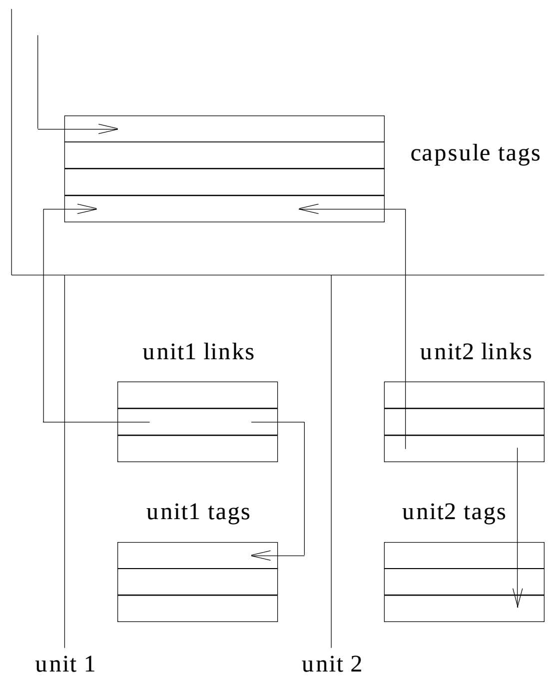
In just the same way, there are tables of tokens and alignment tags at the capsule level, and external links for these as well.

The tags used inside a unit have to be related to these capsule tags, so that they can be properly named. A similar mechanism is used, with a table of tags at the unit level, and links between these and the capsule level tags.



Again the same technique is used for tokens and alignment tags.

It is also necessary for a tag used in one unit to refer to the same thing as a tag in another unit. To do this a tag at the capsule level is used, which may or may not have an external link.



The same technique is used for tokens and alignment tags.

So, when the XANDF linker is joining two capsules, it has to perform the following tasks:

- It creates new sets of capsule level tags, tokens and alignment tags by identifying those which have the same external name, and otherwise creating different entries.
- It similarly joins the external links, suppressing any names which are no longer to be external.
- It produces new link tables for the units, so that the entities used inside the units are linked to the new positions in the capsule level tables.
- It reorganises the units so that the correct order is achieved.

This can be done without looking into the interior of the units (except for the **tld** unit), simply copying the units into their new place.

During the process of installation the values associated with the linkable entities can be accessed by indexing into an array followed by one indirection. These are the kinds of object which in a programming language are referred to by using identifiers, which involves using hash tables for access. This is an example of a general principle of the design of XANDF: speed is required in the linking and installing processes, if necessary at the expense of time in the production of XANDF.

2.2 Tokens

Tokens are used (applied) in the XANDF at the point where substitutions are to be made. Token definitions provide the substitutions and usually reside on the target machine and are linked in there.

A typical token definition has parameters from various SORTS and produces a result of a given SORT. As an example of a simple token definition, written here in a C-like notation, consider the following.

```
EXP ptr_add(EXP par0,EXP par1, SHAPE par2)
{add_to_ptr(
    par0,
    offset_mult(
       offset_pad(
          alignment(par2),
          shape_offset(par2))
       par1))
}
```

This defines the token **ptr_add**, to produce something of SORT EXP. It has three parameters, of SORTS EXP, EXP and SHAPE. The **add_to_ptr**, **offset_mult**, **offset_pad**, **alignment** and **shape_offset** constructions are XANDF constructions producing respectively an EXP, an EXP, an EXP, an ALIGNMENT and an EXP.

A typical use of this token is:

```
ptr_add(
     obtain_tag(tag41),
     contents(integer(signed_int),
              obtain_tag(tag62)),
     integer(char))
```

The effect of this use is to produce the XANDF of the definition with *par0*, *par1* and *par2* substituted by the actual parameters.

There is no way of obtaining anything like a side-effect. A token without parameters is therefore just a constant.

Tokens can be used for various purposes. They are used to make the XANDF shorter by using tokens for commonly used constructions (**ptr_add** is an example of this use). They are used to make target dependent substitutions (*char* in the use of **ptr_add** is an example of this, since *char* may be signed or unsigned on the target).

A particularly important use is to provide definitions appropriate to the translation of a particular language. Another is to abstract those features which differ from one ABI to another. This kind of use requires that sets of tokens should be standardised for these purposes, since otherwise there will be a proliferation of such definitions.

2.3 Tags

Tags are used to identify the actual program components. They can be declared or defined. A declaration gives the SHAPE of a tag (a SHAPE is the analogue of a type). A definition gives an EXP for the tag (an EXP describes how the value is to be made up).

2.4 Extending the Format

XANDF can be extended for two major reasons.

First, as part of the evolution of XANDF, new features will from time to time be identified. It is highly desirable that these can be added without disturbing the current encoding, so that old XANDF can still be installed by systems which recognise the new constructions. Such changes should only be made infrequently and with great care, for stability reasons, but nevertheless they must be allowed for in the design.

Second, it may be required to add extra information to XANDF to permit special processing. XANDF is a way of describing programs and it clearly may be used for other reasons than portability and distribution. In these uses it may be necessary to add extra information which is closely integrated with the program. Diagnostics and profiling can serve as examples. In these cases the extra kinds of information may not have been allowed for in the XANDF encoding.

Some extension mechanisms are described below and are related to these reasons.

1. The encoding of every SORT in XANDF can be extended indefinitely (except for certain auxiliary SORTS). This mechanism should only be used for extending standard XANDF to the next standard, since otherwise extensions made by different groups of people might conflict with each other. See Section 8.2.8 on page 161.

2. Basic XANDF has three kinds of linkable entity and seven kinds of unit. It also contains a mechanism for extending these so that other information can be transmitted in a capsule and properly related to basic XANDF. The rules for linking this extra information are also laid down. See Section 5.11.1 on page 33.

   If a new kind of unit is added, it can contain any information, but if it is to refer to the tags and tokens of other units it must use the linkable entities. Since new kinds of unit might need extra kinds of linkable entity, a method for adding these is also provided. All this works in a uniform way, with capsule level tables of the new entities, and external and internal links for them. If new kinds of unit are added, the order of groups must be the same in any capsules which are linked together.

   As an example of the use of this kind of extension, the diagnostic information is introduced in just this way. It uses two extra kinds of unit and one extra kind of linkable entity. The extra units need to refer to the tags in the other units, since these are the object of the diagnostic information.

   This mechanism can be used for both purposes.

3. The parameters of tokens are encoded in such a way that foreign information (that is, information which cannot be expressed in the XANDF SORTS) can be supplied. This mechanism should only be used for the second purpose, though it could be used to experiment with extensions for future standards. See Section 8.2.5 on page 160.

Chapter 3

Describing the Structure

The following examples show how XANDF constructs are described in this document. The first is the construct **floating** (see Section 5.31.6 on page 103):

fv:	FLOATING_VARIETY
	→ SHAPE

The constructs' arguments (one in this case) precede the ''→'' and the result follows it. Each argument is shown as follows:

name:	SORT

The name standing before the colon is for use in the accompanying English description within the specification. It has no other significance.

The example given above indicates that **floating** takes one argument. This argument, *v*, is of SORT FLOATING_VARIETY. After the → comes the SORT of the result of **floating.** It is a SHAPE.

In the case of **floating** the formal description supplies the syntax and the accompanying English text supplies the semantics. However, in the case of some constructs it is convenient to specify more information in the formal section. For example, the specification of the construct **floating_negate** (see Section 5.16.39 on page 53) not only states that it has an EXP argument and an EXP result:

flpt_err:	ERROR_TREATMENT
arg1:	EXP FLOATING(*f*)
	→ EXP FLOATING(*f*)

but also supplies additional information about those EXPs; it specifies that these expressions will be floating point numbers of the same kind.

In some constructs, arguments are optional. This is denoted as follows (the example is from Section 5.16.6 on page 40) :

result_shape:	SHAPE
p:	EXP PROC
params:	LIST(EXP)
var_param:	OPTION(EXP)
	→ EXP *result_shape*

In this example, *var_param* is an optional argument to the **apply_proc** construct.

Some constructs take a varying number of arguments, for example *params* in the above construct. These are denoted by LIST. There is a similar construction, SLIST, which differs only in having a different encoding.

Some constructs' results are governed by the values of their arguments. This is denoted by the ? formation shown in the specification of the case (see Section 5.16.14 on page 44) construct shown below:

exhaustive:	BOOL
control:	EXP INTEGER(*v*)
branches:	LIST(CASELIM)
	→ EXP (*exhaustive* ?BOTTOM:TOP)

If *exhaustive* is true, the resulting EXP has the SHAPE BOTTOM ; otherwise it is TOP.

Depending on an XANDF-processing tool's purpose, not all of some constructs' arguments need necessarily be processed. For instance, installers do not need to process one of the arguments of the **x_cond** constructs (where *x* stands for a SORT, for example, **exp_cond** on Section 5.16.2 on page 39). Secondly, standard tools might want to ignore embedded fragments of XANDF adhering to some private standard. In these cases it is desirable for tools to be able to skip the irrelevant pieces of XANDF. BITSTREAMs and BYTESTREAMs are formations which permit this. In the encoding they are prefaced with information about their length.

Some constructs' arguments are defined as being BITSTREAMs or BYTESTREAMs, even though the constructs specify them to be of a particular SORT. In these cases the argument's SORT is denoted as, for example:

BITSTREAM FLOATING_VARIETY

This construct must have a FLOATING_VARIETY argument, but certain XANDF-processing tools may benefit from being able to skip past the argument (which might itself be a very large piece of XANDF) without having to read its content.

The nature of the UNITS in a GROUP is determined by unit identifications. These occur in **make_capsule.** The values used for unit identifications are specified in the text as follows:

Unit identification: *some_name*

where *some_name* might be **tokdec**, **tokdef**, and so on.

The kinds of linkable entity used are determined by linkable entity identifications. These occur in **make_capsule.** The values used for linkable entity identification are specified in the text as follows:

Linkable entity identification: *some_name*

where *some_name* might be **tag**, **token**, and so on.

The bit encodings are also specified in this document. The details are given in Chapter 8. This section describes the encoding in terms of information given with the descriptions of the SORTS and constructs.

With each SORT the number of bits used to encode the constructs is given in the following form:

Number of encoding bits: *n*

This number may be zero; if so, the encoding is non-extendable. If it is non-zero, the encoding may be extendible or non-extendible. This is specified in the following form:

Is coding extendible? Yes or No

With each construct the number used to encode it is given in the following form:

Encoding number: *n*

If the number of encoding bits is zero, *n* will be zero.

There may be a requirement that a component of a construct should start on a byte boundary in the encoding. This is denoted by inserting:

> *byte_align*

before the component.

Chapter 4

Installer Behaviour

4.1 Definition of Terms

In this document the behaviour of XANDF installers is described in a precise manner. Certain words are used with very specific meanings. These are:

undefined	Installers can perform any action, including refusing to translate the program. It can produce code with any effect, meaningful or meaningless.
shall	When the phrase ''P shall be done'' (or similar phrases involving ''shall'') is used, every installer must perform P.
should	When the phrase ''P should be done'' (or similar phrase involving ''should'') is used, installers are advised to perform P, and producer writers may assume it will be done if possible. This usage generally relates to optimisations which are recommended.
will	When the phrase ''P will be true'' (or similar phrases involving ''will'') is used to describe the composition of a XANDF construct, the installer may assume that P holds without having to check it. If, in fact, a producer has produced XANDF for which P does not hold, the effect is undefined.
target-defined	Behaviour will be defined, but it varies from one target machine to another. Each target installer shall define everything which is said to be ''target-defined''.

4.2 Properties of Installers

All installers must implement all of the constructions of XANDF. There are some constructions where the installers may impose limits on the ranges of values which are implemented. In these cases the description of the installer must specify these limits.

Installers are not expected to check that the XANDF they are processing is well-formed, nor that undefined constructs are absent. If the XANDF is not well-formed, any effect is permitted.

Installers shall only implement optimisations which are correct in all circumstances. This correctness can only be shown by demonstrating the equivalence of the transformed program, from equivalences deducible from this specification or from the ordinary laws of arithmetic. No statements are made in this specification of the form ''such-and-such an optimisation is permitted''.

Note: Fortran90 has a notion of mathematical equivalence which is not the same as XANDF equivalence. It can be applied to transform programs provided parentheses in the text are not crossed. XANDF does not acknowledge this concept. Such transformations would have to be applied in a context where the permitted changes are known.

Chapter 5

Specification of XANDF Constructs

5.1 ACCESS

Number of encoding bits: 4

Is coding extendable? Yes

An ACCESS describes properties a variable or identity may have which may constrain or describe the ways in which the variable or identity is used.

Each construction which needs an ACCESS uses it in the form OPTION (ACCESS). If the option is absent, the variable or identity has no special properties.

An ACCESS acts like a set of the values **constant**, **long_jump_access**, **no_other_read**, **no_other_write**, **register**, **out_par**, **used_as_volatile,** and **visible.** **standard_access** acts like the empty set. **add_accesses** is the set union operation.

5.1.1 access_apply_token

Encoding number: 1

token_value:	TOKEN
token_args:	BITSTREAM *param_sorts(token_value)*
	→ ACCESS

The token is applied to the arguments encoded in the BITSTREAM *token_args* to give an ACCESS.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.1.2 access_cond

Encoding number: 2

control:	EXP INTEGER(*v*)
e1:	BITSTREAM ACCESS
e2:	BITSTREAM ACCESS
	→ ACCESS

control is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

5.1.3 add_accesses

Encoding number: 3

a1: ACCESS
a2: ACCESS
→ ACCESS

A construction qualified with **add_accesses** has both ACCESS properties *a1* and *a2*. This operation is associative and commutative.

5.1.4 constant

Encoding number: 4

→ ACCESS

Only a variable (not an identity) may be qualified with **constant**. A variable qualified with **constant** will retain its initialising value unchanged throughout its lifetime.

5.1.5 long_jump_access

Encoding number: 5

→ ACCESS

An object must also have this property if it is to have a defined value when a **long_jump** returns to the procedure declaring the object.

5.1.6 no_other_read

Encoding number: 6

→ ACCESS

This property refers to a POINTER, *p*. It says that within the lifetime of the declaration being qualified, there are no **contents**, **contents_with_mode** or **move_some** source accesses to any pointer not derived from *p* which overlap with any of the **contents**, **contents_with_mode**, **assign**, **assign_with_mode** or **move_some** accesses to pointers derived from *p*.

The POINTER being described is that obtained by applying **obtain_tag** to the TAG of the declaration. If the declaration is an **identity**, the SHAPE of the TAG will be a POINTER.

5.1.7 no_other_write

Encoding number: 7

→ ACCESS

This property refers to a POINTER, *p*. It says that, within the lifetime of the declaration being qualified, there are no **assign**, **assign_with_mode** or **move_some** destination accesses to any pointer not derived from *p* which overlap with any of the **contents**, **contents_with_mode**, **assign**, **assign_with_mode** or **move_some** accesses to pointers derived from *p*.

The POINTER being described is that obtained by applying **obtain_tag** to the TAG of the declaration. If the declaration is an **identity**, the SHAPE of the TAG will be a POINTER.

5.1.8 out_par

Encoding number: 8

→ ACCESS

An object qualified by **out_par** will be an output parameter in a **make_general_proc** construct. This will indicate that the final value of the parameter is required in the *postlude* part of an **apply_general_proc** of this procedure.

5.1.9 preserve

Encoding number: 9

→ ACCESS

This property refers to a global object. It says that the object will be included in the final program, whether or not all possible accesses to that object are optimised away, for example by inlining all possible uses of procedure object.

5.1.10 register

Encoding number: 10

→ ACCESS

Indicates that an object with this property is frequently used. This can be taken as a recommendation to place it in a register.

5.1.11 standard_access

Encoding number: 11

→ ACCESS

An object qualified as having **standard_access** has normal (that is, no special) access properties.

5.1.12 used_as_volatile

Encoding number: 12

→ ACCESS

An object qualified as having **used_as_volatile** will be used in a **move_some**, **contents_with_mode** or an **assign_with_mode** construct with TRANSFER_MODE volatile.

5.1.13 visible

Encoding number: 13

→ ACCESS

An object qualified as **visible** may be accessed when the procedure in which it is declared is not the current procedure. A TAG must have this property if it is to be used by **env_offset.**

5.2 AL_TAG

Number of encoding bits: 1

Is coding extendable? Yes

Linkable entity identification: alignment

AL_TAGs name ALIGNMENTs. They are used so that circular definitions can be written in XANDF. However, because of the definition of alignments, intrinsic circularities cannot occur.

For example, the following equation has a circular form:

x = alignment(pointer(alignment(x)))

and it or a similar equation might occur in XANDF. However, since *alignment(pointer(x))* is *{pointer}*, this reduces to *x = {pointer}*.

5.2.1 al_tag_apply_token

Encoding number: 2

token_value:	TOKEN
token_args:	BITSTREAM *param_sorts(token_value)*
	→ AL_TAG

The token is applied to the arguments encoded in the BITSTREAM *token_args* to give an AL_TAG.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.2.2 make_al_tag

Encoding number: 1

al_tagno:	TDFINT
	→ AL_TAG

make_al_tag constructs an AL_TAG identified by *al_tagno*.

5.3 AL_TAGDEF

Number of encoding bits: 1

Is coding extendable? Yes

An AL_TAGDEF gives the definition of an AL_TAG for incorporation into a AL_TAGDEF_PROPS.

5.3.1 make_al_tagdef

Encoding number: 1

t: TDFINT
a: ALIGNMENT
→ AL_TAGDEF

The AL_TAG identified by *t* is defined to stand for the ALIGNMENT *a*. All the AL_TAGDEFs in a CAPSULE must be considered together as a set of simultaneous equations defining ALIGNMENT values for the AL_TAGs. No order is imposed on the definitions.

In any particular CAPSULE the set of equations may be incomplete, but a CAPSULE which is being translated into code will have a set of equations which defines all the AL_TAGs which it uses.

The result of the evaluation of the *control* argument of any **_cond** construction (for example, **alignment_cond**) used in *a* shall be independent of any AL_TAGs used in the control. Simultaneous equations defining ALIGNMENTs can then always be solved.

See also Section 7.13.3 on page 149.

5.4 AL_TAGDEF_PROPS

Number of encoding bits: 0

Unit identification: aldef

5.4.1 make_al_tagdefs

Encoding number: 0

no_labels:	TDFINT
tds:	SLIST(AL_TAGDEF)
	→ AL_TAGDEF_PROP

no_labels is the number of local LABELs used in *tds*. *tds* is a list of AL_TAGDEFs which define the bindings for **al_tags**.

5.5 ALIGNMENT

Number of encoding bits: 4

Is coding extendable? Yes

An ALIGNMENT gives information about the layout of data in memory and hence is a parameter for the POINTER and OFFSET SHAPES (see Section 7.13 on page 147). This information consists of a set of elements.

The possible values of the elements in such a set are **proc**, **code**, **pointer**, **offset,** all VARIETIES, all FLOATING_VARIETIES and all BITFIELD_VARIETIES. The sets are written here as, for example, { *pointer*, *proc* } meaning the set containing *pointer* and *proc*.

In addition, there are ''special'' ALIGNMENTs **alloca_alignment**, **callers_alignment**, **callees_alignment**, **locals_alignment** and **var_param_alignment**. Each of these are considered to be sets which include all of the ''ordinary'' ALIGNMENTs above.

There is a function, **alignment,** which can be applied to a SHAPE to give an ALIGNMENT (see the definition below). The interpretation of a POINTER to an ALIGNMENT, *a*, is that it can serve as a POINTER to any SHAPE, *s*, such that *alignment* (*s*) is a subset of the set *a*.

So given a POINTER ({ *proc*, *pointer* }) it is permitted to assign a PROC or a POINTER to it, or indeed a compound containing only PROCS and POINTERS. This permission is valid only in respect of the space being of the right kind; it may or may not be big enough for the data.

The most usual use for ALIGNMENT is to ensure that addresses of *int* values are aligned on 4-byte boundaries, *float* values are aligned on 4-byte boundaries, *doubles* on 8-bit boundaries, and so on, and whatever may be implied by the definitions of the machines and languages involved.

In the specification the phrase ''*a* will include *b*'' where *a* and *b* are ALIGNMENTS, means that the set *b* will be a subset of *a* (or equal to *a*).

5.5.1 alignment_apply_token

Encoding number: 1

token_value:	TOKEN
token_args:	BITSTREAM *param_sorts(token_value)*
	→ ALIGNMENT

The token is applied to the arguments encoded in the BITSTREAM *token_args* to give an ALIGNMENT.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.5.2 alignment_cond

Encoding number: 2

control: EXP INTEGER(*v*)
e1: BITSTREAM ALIGNMENT
e2: BITSTREAM ALIGNMENT
→ ALIGNMENT

control is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If control is zero then *e2* is installed at this point and *e1* is ignored and never processed.

5.5.3 alignment

Encoding number: 3

sha: SHAPE
→ ALIGNMENT

The **alignment** construct is defined as follows:

- If *sha* is PROC then the resulting ALIGNMENT is { *proc* }.
- If *sha* is INTEGER (*v*) then the resulting ALIGNMENT is { *v* }.
- If *sha* is FLOATING (*v*) then the resulting ALIGNMENT is { *v* }.
- If *sha* is BITFIELD (*v*) then the resulting ALIGNMENT is { *v* }.
- If *sha* is TOP the resulting ALIGNMENT is { } — the empty set.
- If *sha* is BOTTOM the resulting ALIGNMENT is undefined.
- If *sha* is POINTER (*x*) or OFFSET (*x*, *y*) then the resulting ALIGNMENT is { *pointer* } or { *offset* } respectively.
- If *sha* is NOF (*n*, s) the resulting ALIGNMENT is **alignment** (*s*).
- If *sha* is COMPOUND (EXP OFFSET (*x*, *y*)) then the resulting ALIGNMENT is *x*.

5.5.4 alloca_alignment

Encoding number: 4

→ ALIGNMENT

This delivers the ALIGNMENT of POINTERS produced from **local_alloc**.

5.5.5 callees_alignment

Encoding number: 5

var: BOOL
→ ALIGNMENT

If *var* is true the ALIGNMENT is that of callee parameters qualified by the PROCPROPS **var_callees**. If *var* is false, the ALIGNMENT is that of callee parameters not qualified by PROCPROPS **var_callees**.

This delivers the base ALIGNMENT of OFFSETS from a frame-pointer to a CALLEE parameter. Values of such OFFSETS can only be produced by **env_offset** applied to CALLEE parameters, or offset arithmetic operations applied to existing OFFSETS.

5.5.6 callers_alignment

Encoding number: 6

var: BOOL
→ ALIGNMENT

If *var* is true the ALIGNMENT is that of caller parameters qualified by the PROCPROPS **var_callers**. If *var* is false, the ALIGNMENT is that of caller parameters not qualified by PROCPROPS **var_callers**.

This delivers the base ALIGNMENT of OFFSETS from a frame-pointer to a CALLER parameter. Values of such OFFSETS can only be produced by **env_offset** applied to CALLER parameters, or offset arithmetic operations applied to existing OFFSETS.

5.5.7 code_alignment

Encoding number: 7

→ ALIGNMENT

This delivers { *code* }, the ALIGNMENT of the POINTER produced by **make_local_lv**.

5.5.8 locals_alignment

Encoding number: 8

→ ALIGNMENT

This delivers the base ALIGNMENT of OFFSETS from a frame-pointer to a value defined by *variable* or *identify*. Values of such OFFSETS can only be produced by **env_offset** applied to TAGS so defined, or offset arithmetic operations applied to existing OFFSETS.

5.5.9 obtain_al_tag

Encoding number: 9

at:	AL_TAG
	→ ALIGNMENT

obtain_al_tag produces the ALIGNMENT with which the AL_TAG *at* is bound.

5.5.10 parameter_alignment

Encoding number: 10

sha:	SHAPE
	→ ALIGNMENT

This delivers the ALIGNMENT of a parameter of a procedure of SHAPE *sha*.

5.5.11 unite_alignments

Encoding number: 11

a1:	ALIGNMENT
a2:	ALIGNMENT
	→ ALIGNMENT

unite_alignments produces the alignment at which all the members of the ALIGNMENT sets *a1* and *a2* can be placed — in other words the ALIGNMENT set which is the union of *a1* and *a2*.

5.5.12 var_param_alignment

Encoding number: 12

→ ALIGNMENT

This delivers the ALIGNMENT used in the *var_param* argument of **make_proc**.

5.6 BITFIELD_VARIETY

Number of encoding bits: 2

Is coding extendable? Yes

These describe runtime bitfield values. The intention is that these values are usually kept in memory locations which need not be aligned on addressing boundaries.

There is no limit on the size of bitfield values in XANDF, but an installer may specify limits. See Section 7.24 on page 154 and Section 7.25 on page 155.

5.6.1 bfvar_apply_token

Encoding number: 1

token_value:	TOKEN
token_args:	BITSTREAM *param_sorts(token_value)*
	→ BITFIELD_VARIETY

The token is applied to the arguments encoded in the BITSTREAM *token_args* to give a BITFIELD_VARIETY.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.6.2 bfvar_cond

Encoding number: 2

control:	EXP INTEGER(*v*)
e1:	BITSTREAM BITFIELD_VARIETY
e2:	BITSTREAM BITFIELD_VARIETY
	→ BITFIELD_VARIETY

control is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If control is zero then *e2* is installed at this point and *e1* is ignored and never processed.

5.6.3 bfvar_bits

Encoding number: 3

issigned:	BOOL
bits:	NAT
	→ BITFIELD_VARIETY

bfvar_bits constructs a BITFIELD_VARIETY describing a pattern of *bits* bits. If *issigned* is true, the pattern is considered to be a twos-complement signed number; otherwise it is considered to be unsigned.

5.7 BITSTREAM

A BITSTREAM consists of an encoding of any number of bits. This encoding is such that any program reading XANDF can determine how to skip over it. To read it meaningfully, extra knowledge of what it represents may be needed.

A BITSTREAM is used, for example, to supply parameters in a TOKEN application. If there is a definition of this TOKEN available, this will provide the information needed to decode the bitstream.

See Section 8.3 on page 161.

5.8 BOOL

Number of encoding bits: 3

Is coding extendable? Yes

A BOOL is a piece of XANDF which can take two values: *true* or *false*.

5.8.1 bool_apply_token

Encoding number: 1

token_value:	TOKEN
token_args:	BITSTREAM *param_sorts(token_value)*
	→ BOOL

The token is applied to the arguments encoded in the BITSTREAM *token_args* to give a BOOL.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.8.2 bool_cond

Encoding number: 2

control:	EXP INTEGER(*v*)
e1:	BITSTREAM BOOL
e2:	BITSTREAM BOOL
	→ BOOL

control is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If control is zero then *e2* is installed at this point and *e1* is ignored and never processed.

5.8.3 false

Encoding number: 3

→ BOOL

false produces a false BOOL.

5.8.4 true

Encoding number: 4

→ BOOL

true produces a true BOOL.

5.9 BYTESTREAM

A BYTESTREAM is analogous to a BITSTREAM, but is encoded to permit fast copying.

See Section 8.3 on page 161.

5.10 CALLEES

Number of encoding bits: 2

Is coding extendable? Yes

This is an auxiliary SORT used in calling procedures by **apply_general_proc** and **tail_call** to provide their actual callee parameters.

5.10.1 make_callee_list

Encoding number: 1

args: LIST(EXP)
$\rightarrow$ CALLEES

The list of EXPS *args* are evaluated in any interleaved order, and the resulting list of values form the actual callee parameters of the call.

5.10.2 make_dynamic_callees

Encoding number: 2

ptr: EXP POINTER(*x*)
sze: EXP OFFSET(*x*,*y*)
$\rightarrow$ CALLEES

The value of size *sze* pointed at by *ptr* forms the actual callee parameters of the call.

The call involved (that is, **apply_general_proc** or **tail_call**) must have a **var_callees** PROCPROPS.

5.10.3 same_callees

Encoding number: 3

$\rightarrow$ CALLEES

The callee parameters of the call are the same as those of the current procedure.

5.11 CAPSULE

Number of encoding bits: 0

Is coding extendable? No

A CAPSULE is an independent piece of XANDF. There is only one construction: **make_capsule.**

5.11.1 make_capsule

Encoding number: 0

prop_names:	SLIST(TDFIDENT)
capsule_linking:	SLIST(CAPSULE_LINK)
external_linkage:	SLIST(EXTERN_LINK)
groups:	SLIST(GROUP)
	$\rightarrow$ CAPSULE

make_capsule brings together UNITs and linking and naming information. See Section 2.1 on page 5.

The elements of the list, *prop_names*, correspond one-to-one with the elements of the list, *groups*. The element of *prop_names* is the unit identification of all the UNITS in the corresponding GROUP. See Section 5.29 on page 99. A CAPSULE need not contain all the kinds of UNIT.

It is intended that new kinds of PROPs with new unit identifications can be added to the standard in a purely additive fashion, either to form a new standard or for private purposes.

The elements of the list, *capsule_linking*, correspond one-to-one with the elements of the list, *external_linkage*. The element of *capsule_linking* gives the linkable entity identification for all the LINKEXTERNS in the element of *external_linkage*. It also gives the number of CAPSULE level linkable entities having that identification.

The elements of the list, *capsule_linking*, also correspond one-to-one with the elements of the lists called *local_vars* in each of the *make_unit* constructions for the UNITS in *groups*. The element of *local_vars* gives the number of UNIT-level linkable entities having the identification in the corresponding member of *capsule_linking*.

It is intended that new kinds of linkable entity can be added to the standard in a purely additive fashion, either to form a new standard or for private purposes.

external_linkage provides a list of lists of LINKEXTERNs. These LINKEXTERNs specify the associations between the names to be used outside the CAPSULE and the linkable entities by which the UNITs make objects available within the CAPSULE.

The list, *groups*, provides the non-linkage information of the CAPSULE.

5.12 CAPSULE_LINK

Number of encoding bits: 0

Is coding extendable? No

This is an auxiliary SORT which gives the number of linkable entities of a given kind at CAPSULE level. It is used only in **make_capsule**.

5.12.1 make_capsule_link

Encoding number: 0

sn: TDFIDENT
n: TDFINT
→ CAPSULE_LINK

n is the number of CAPSULE level linkable entities (numbered from 0 to *n* -1) of the kind given by *sn*. *sn* corresponds to the linkable entity identification.

5.13 CASELIM

Number of encoding bits: 0

Is coding extendable? No

This is an auxiliary SORT which provides lower and upper bounds and the LABEL destination for the **case** construction.

5.13.1 make_caselim

Encoding number: 0

branch:	LABEL
lower:	SIGNED_NAT
upper:	SIGNED_NAT
	→ CASELIM

This makes a triple of destination and limits. The **case** construction (see Section 5.16.14 on page 44) uses a list of CASELIMS. If the control variable of the case lies between *lower* and *upper*, control passes to *branch*.

5.14 ERROR_CODE

Number of encoding bits: 2

Is coding extendable? Yes

5.14.1 nil_access

Encoding number: 1

$\rightarrow$ ERROR_CODE

This delivers the ERROR_CODE arising from an attempt to access a nil pointer in an operation with TRANSFER_MODE **trap_on_nil**.

5.14.2 overflow

Encoding number: 2

$\rightarrow$ ERROR_CODE

This delivers the ERROR_CODE arising from a numerical exceptional result in an operation with ERROR_TREATMENT *trap*(*overflow*).

5.14.3 stack_overflow

Encoding number: 3

$\rightarrow$ ERROR_CODE

This delivers the ERROR_CODE arising from a stack overflow in the call of a procedure defined with PROCPROPS **check_stack**.

5.15 ERROR_TREATMENT

Number of encoding bits: 3

Is coding extendable? Yes

These values describe the way to handle various forms of error which can occur during the evaluation of operations.

It is expected that additional ERROR_TREATMENTS will be needed.

5.15.1 errt_apply_token

Encoding number: 1

token_value:	TOKEN
token_args:	BITSTREAM *param_sorts(token_value)*
	→ ERROR_TREATMENT

The token is applied to the arguments encoded in the BITSTREAM *token_args* to give an ERROR_TREATMENT.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.15.2 errt_cond

Encoding number: 2

control:	EXP INTEGER(*v*)
e1:	BITSTREAM ERROR_TREATMENT
e2:	BITSTREAM ERROR_TREATMENT
	→ ERROR_TREATMENT

control is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

5.15.3 continue

Encoding number: 3

→ ERROR_TREATMENT

If an operation with a continue ERROR_TREATMENT causes an error, some value of the correct SHAPE shall be delivered. This value shall have the same properties as is specified in **make_value.**

5.15.4 error_jump

Encoding number: 4

lab: LABEL
→ ERROR_TREATMENT

error_jump produces an ERROR_TREATMENT which requires that control be passed to *lab* if it is invoked. *lab* will be in scope.

If a construction has an **error_jump** ERROR_TREATMENT and the jump is taken, the canonical order specifies only that the jump occurs after evaluating the construction. It is not specified how many further constructions are evaluated.

Note: This rule implies that a further construction is needed to guarantee that errors have been processed. This is not yet included. The effect of nearby procedure calls or exits also needs definition.

5.15.5 trap

Encoding number: 5

trap_list: LIST(ERROR_CODE)
→ ERROR_TREATMENT

The list of ERROR_CODES in *trap_list* specifies a set of possible exceptional behaviours. If any of these occur in an construction with ERROR_TREATMENT **trap**, the XANDF exception handling is invoked (see Section 7.8).

The observations on canonical ordering in **error_jump** apply equally here.

5.15.6 wrap

Encoding number: 6

→ ERROR_TREATMENT

wrap is an ERROR_TREATMENT which will only be used in constructions with integer operands and delivering EXP INTEGER (v) where either the lower bound of v is zero or the construction is not one of **mult, power, div0, div1, div2, rem0, rem1, rem2**. The result will be evaluated and any bits in the result lying outside the representing VARIETY will be discarded (see Section 7.18 on page 151).

5.15.7 impossible

Encoding number: 7

→ ERROR_TREATMENT

impossible is an ERROR_TREATMENT which means that this error will not occur in the construct concerned.

Note: **impossible** is possibly a misnomer. If an error occurs the result is undefined.

5.16 EXP

Number of encoding bits: 7

Is coding extendable? Yes

EXPs are pieces of XANDF which are translated into program. EXP is by far the richest SORT. There are few primitive EXPs; most of the constructions take arguments which are a mixture of EXPs and other SORTs. There are constructs delivering EXPs that correspond to the declarations, program structure, procedure calls, assignments, pointer manipulation, arithmetic operations, tests, and so on, of programming languages.

5.16.1 exp_apply_token

Encoding number: 1

token_value:	TOKEN
token_args:	BITSTREAM *param_sorts(token_value)*
	→ EXP *x*

The token is applied to the arguments encoded in the BITSTREAM *token_args* to give an EXP.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.16.2 exp_cond

Encoding number: 2

control:	EXP INTEGER(*v*)
e1:	BITSTREAM EXP*x*
e2:	BITSTREAM EXP*y*
	→ EXP *x* or EXP *y*

control is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If control is zero then *e2* is installed at this point and *e1* is ignored and never processed.

5.16.3 abs

Encoding number: 3

ov_err:	ERROR_TREATMENT
arg1:	EXP INTEGER(*v*)
	→ EXP INTEGER(*v*)

The absolute value of the result produced by *arg1* is delivered.

If the result cannot be expressed in the VARIETY being used to represent *v*, an overflow error is caused and is handled in the way specified by *ov_err*.

5.16.4 add_to_ptr

Encoding number: 4

arg1:	EXP POINTER(*x*)
arg2:	EXP OFFSET(*y*,*z*)
	→ EXP POINTER(*x*)

arg1 is evaluated, giving *p,* and *arg2* is evaluated and the results are added to produce the answer. The result is derived from the pointer delivered by *arg1*. The intention is to produce a POINTER displaced from the argument POINTER by the given amount.

x will include *y*.

arg1 may deliver a null POINTER. In this case the result is derived from a null POINTER which counts as an original POINTER. Further OFFSETS may be added to the result, but the only other useful operation on the result of adding a number of OFFSETS to a null POINTER is to *subtract_ptrs* a null POINTER from it.

The result will be less than or equal (in the sense of **pointer_test**) to the result of applying **add_to_ptr** to the original pointer from which *p* is derived and the size of the space allocated for the original pointer.

Note: In the simple representation of POINTER arithmetic (see Section 7.13 on page 147) **add_to_ptr** is represented by addition. The constraint ''*x* includes *y*'' ensures that no padding has to be inserted in this case.

5.16.5 and

Encoding number: 5

arg1:	EXP INTEGER(*v*)
arg2:	EXP INTEGER(*v*)
	→ EXP INTEGER(*v*)

The arguments are evaluated, producing integer values of the same VARIETY, *v*. The result is the bitwise logical ''and'' of the two values in the representing VARIETY. The result is delivered with the same SHAPE as the arguments.

See also Section 7.18 on page 151.

5.16.6 apply_proc

Encoding number: 6

result_shape:	SHAPE
p:	EXP PROC
params:	LIST(EXP)
var_param:	OPTION(EXP)
	→ EXP *result_shape*

p, *params* and *var_param* (if present) are evaluated in any interleaved order. The procedure, *p,* is applied to the parameters. The result of the procedure call, which will have *result_shape*, is delivered as the result of the construction.

The canonical order of evaluation is as if the definition were inlined. That is, the actual parameters are evaluated interleaved in any order and used to initialise variables which are identified by the formal parameters during the evaluation of the procedure body. When this is complete, the body is evaluated. So, **apply_proc** is evaluated like a variable construction, and obeys similar rules for order of evaluation.

If *p* delivers a null procedure, the effect is undefined.

var_param is intended to communicate parameters which vary in SHAPE from call to call. Access to these parameters during the procedure is performed by using OFFSET arithmetic. Note that it is necessary to place these values on **var_param_alignment** because of the definition of **make_proc**.

All calls to the same procedure will yield results of the same SHAPE.

For notes on the intended implementation of procedures, see Section 7.9 on page 143.

5.16.7 apply_general_proc

Encoding number: 7

result_shape:	SHAPE
prcprops:	OPTION(PROCPROPS)
p:	EXP PROC
caller_params:	LIST(OTAGEXP)
callee_params:	CALLEES
postlude:	EXP TOP
	→ EXP *result_shape*

p, *caller_params* and *callee_params* are evaluated in any order. The procedure, *p*, is applied to the parameters. The result of the procedure call, which will have *result_shape,* is delivered as the result of the construction.

If *p* delivers a null procedure, the effect is undefined.

Any TAG introduced by an OTAGEXP in *caller_params* is available in *postlude* which will be evaluated after the application.

postlude will not contain any **local_allocs** or calls of procedures with untidy returns. If *prcprops* include **untidy**, *postlude* will be **make_top**.

The canonical order of evaluation is as if the definition of *p* were inlined in a manner dependent on *prcprops*.

If none of the PROCPROPS **var_callers, var_callees** and **check_stack** are present, the inlining is as follows, supposing that P is the body of the definition of *p*:

> Let R_i be the value of the EXP of the i^{th} OTAGEXP in *caller_params* and T_i be its TAG (if it is present). Let E_i be the i^{th} value in **callee_params**. Let r_i be the i^{th} formal caller parameter TAG of *p*. Let e_i be the i^{th} formal callee parameter TAG of *p*.
>
> Each R_i is used to initialise a variable which is identified by r_i; there will be exactly as many R_i as r_i. The scope of these variable definitions is a sequence consisting of three components — the identification of a TAG *res* with the result of a binding of P, followed by a binding of *postlude,* followed by an **obtain_tag** of *res* giving the result of the inlined procedure call.

The binding of P consists of using each E_i to initialize a variable identified with e_i; there will be exactly as many E_i as e_i. The scope of these variable definitions is P modified so that the first *return* or *untidy_return* encountered in P gives the result of the binding. If it ends with a *return*, any space generated by **local_allocs** within the binding is freed (in the sense of **local_free**) at this point. If it ends with *untidy_return*, no freeing will take place.

The binding of *postlude* consists of identifying each T_i (if present) with the contents of the variable identified by r_i. The scope of these identifications is *postlude*.

If the PROCPROPS **var_callers** is present, the inlining process is modified as follows:

> A compound variable is constructed initialised to R_i in order; the alignment and padding of each individual R_i will be given by an exact application of **parameter_alignment** on the SHAPE of R_i. Each r_i is then identified with a pointer to the copy of R_i within the compound variable; there will be at least as many R_i as r_i. The evaluation then continues as above with the scope of these identifications being the sequence.

If the PROCPROPS **var_callees** is present, the inlining process is modified as follows:

> The binding of P is done by generating (as if by **local_alloc**) a pointer to space for a compound value constructed from each E_i in order (just as for **var_callers**). Each e_i is identified with a pointer to the copy of E_i within the generated space; there will be at least as many e_i as E_i. P is evaluated within the scope of these identifications, as before. Note that the generation of space for these callee parameters is a **local_alloc** with the binding of P, and hence will not be freed if P ends with an *untidy_return*.

5.16.8 assign

Encoding number: 8

arg1:	EXP POINTER(x)
arg2:	EXPy
	$\rightarrow$ EXP TOP

The value produced by *arg2* will be put in the space indicated by *arg1*.

x will include *alignment* (y).

y will not be a BITFIELD.

If the space which the pointer indicates does not lie wholly within the space indicated by the original pointer from which it is derived, the effect is undefined.

If the value delivered by *arg1* is a null pointer, the effect is undefined.

See also Section 7.16 on page 151 and Section 7.17 on page 151.

The constraint ‘‘x will include *alignment*(y)’’ ensures in the simple memory model (see Section 7.13 on page 147) that no change is needed to the POINTER.

5.16.9 assign_with_mode

Encoding number: 9

md:	TRANSFER_MODE
arg1:	EXP POINTER(*x*)
arg2:	EXP*y*
	→ EXP TOP

The value produced by *arg2* will be put in the space indicated by *arg1*. The assignment will be carried out as specified by the TRANSFER_MODE (see Section 5.33.19 on page 110).

If *md* consists of **standard_transfer_mode** only, then **assign_with_mode** is the same as **assign.**

x will include *alignment* (*y*).

y will not be a BITFIELD.

If the space which the pointer indicates does not lie wholly within the space indicated by the original pointer from which it is derived, the effect is undefined.

If the value delivered by *arg1* is a null pointer, the effect is undefined.

See also Section 7.16 on page 151 and Section 7.17 on page 151.

5.16.10 bitfield_assign

Encoding number: 10

arg1:	EXP POINTER(*x*)
arg2:	EXP OFFSET(*y*,*z*)
arg3:	EXP BITFIELD(*v*)
	→ EXP TOP

The value delivered by *arg3* is assigned at a displacement given by *arg2* from the pointer delivered by *arg1*.

x will include *y* and *z* will include *v*.

arg2, BITFIELD (*v*) will be *variety_enclosed* (see also Section 7.24 on page 154).

5.16.11 bitfield_assign_with_mode

Encoding number: 11

md:	TRANSFER_MODE
arg1:	EXP POINTER(*x*)
arg2:	EXP OFFSET(*y*,*z*)
arg3:	EXP BITFIELD(*v*)
	→ EXP TOP

The value delivered by *arg3* is assigned at a displacement given by *arg2* from the pointer delivered by *arg1* by the TRANSFER_MODE (see Section 5.33.19 on page 110).

If *md* consists of **standard_transfer_mode** only, then **bitfield_assign_with_mode** is the same as **bitfield_assign**.

x will include *y* and *z* will include *v*.

arg2, BITFIELD (*v*) will be *variety_enclosed* (see Section 7.24 on page 154).

5.16.12 bitfield_contents

Encoding number: 12

v:	BITFIELD_VARIETY
arg1:	EXP POINTER(*x*)
arg2:	EXP OFFSET(*y*,*z*)
	→ EXP BITFIELD(*v*)

The bitfield of BITFIELD_VARIETY *v*, located at the displacement delivered by *arg2* from the pointer delivered by *arg1*, is extracted and delivered.

x will include *y* and *z* will include *v*.

arg2, BITFIELD (*v*) will be *variety_enclosed* (see Section 7.24 on page 154).

5.16.13 bitfield_contents_with_mode

Encoding number: 13

md:	TRANSFER_MODE
v:	BITFIELD_VARIETY
arg1:	EXP POINTER(*x*)
arg2:	EXP OFFSET(*y*,*z*)
	→ EXP BITFIELD(*v*)

The bitfield of BITFIELD_VARIETY *v*, located at the displacement delivered by *arg2* from the pointer delivered by *arg1*, is extracted and delivered. The operation will be carried out as specified by the TRANSFER_MODE (see Section 5.33.19 on page 110).

If *md* consists of **standard_transfer_mode** only, then **bitfield_contents_with_mode** is the same as **bitfield_contents**.

x will include *y* and *z* will include *v*.

arg2, BITFIELD (*v*) will be *variety_enclosed* (see Section 7.24 on page 154).

5.16.14 case

Encoding number: 14

exhaustive:	BOOL
control:	EXP INTEGER(*v*)
branches:	LIST(CASELIM)
	→ EXP(*exhaustive*?BOTTOM:TOP)

control is evaluated to produce an integer value, *c*. Then *c* is tested to see if it lies inclusively between *lower* and *upper*, for each element of *branches*. If this tests succeeds, control passes to the label *branch* belonging to that CASELIM (see Section 5.13 on page 35). If *c* lies between no pair, the construct delivers a value of SHAPE TOP. The order in which the comparisons are made is undefined.

The sets of SIGNED_NATs in branches will be disjoint.

If *exhaustive* is true, the value delivered by *control* will lie between one of the *lower* / *upper* pairs.

5.16.15 change_bitfield_to_int

Encoding number: 15

v:	VARIETY
arg1:	EXP BITFIELD(*bv*)
	→ EXP INTEGER(*v*)

arg1 is evaluated and converted to a INTEGER (*v*).

If *arg1* exceed the bounds of *v*, the effect is undefined.

5.16.16 change_floating_variety

Encoding number: 16

flpt_err:	ERROR_TREATMENT
r:	FLOATING_VARIETY
arg1:	EXP FLOATING(*f*)
	→ EXP FLOATING(*r*)

arg1 is evaluated and will produce floating point value, *fp*. The value *fp* is delivered, changed to the representation of the FLOATING_VARIETY *r*.

r and *f* will be both real or both complex.

If there is a floating point error it is handled by **flpt_err**.

See also Section 7.21 on page 153.

5.16.17 change_variety

Encoding number: 17

ov_err:	ERROR_TREATMENT
r:	VARIETY
arg1:	EXP INTEGER(*v*)
	→ EXP INTEGER(*r*)

arg1 is evaluated and will produce an integer value, *a*. The value *a* is delivered, changed to the representation of the VARIETY *r*.

If *a* is not contained in the VARIETY being used to represent *r*, an overflow occurs and is handled according to **ov_err**.

5.16.18 change_int_to_bitfield

Encoding number: 18

bv: BITFIELD_VARIETY
arg1: EXP INTEGER(*v*)
→ EXP BITFIELD(*bv*)

arg1 is evaluated and converted to a BITFIELD (*bv*).

If *arg1* exceed the bounds of *bv*, the effect is undefined.

5.16.19 complex_conjugate

Encoding number: 19

c: EXP FLOATING(*cv*)
→ EXP FLOATING(*cv*)

This delivers the complex conjugate of *c*.

cv will be a complex floating variety.

5.16.20 component

Encoding number: 20

sha: SHAPE
arg1: EXP COMPOUND(EXP OFFSET(*x*,*y*))
arg2: EXP OFFSET(*x*,*alignment(sha)*)
→ EXP *sha*

arg1 is evaluated to produce a COMPOUND value. The component of this value at the OFFSET given by *arg2* is delivered. This will have SHAPE *sha*.

arg2 will be a constant and non-negative (see Section 7.3 on page 139).

If *sha* is a BITFIELD then *arg2*, *sha* will be *variety_enclosed* (see Section 7.24 on page 154).

5.16.21 concat_nof

Encoding number: 21

arg1: EXP NOF(*n*,*s*)
arg2: EXP NOF(*m*,*s*)
→ EXP NOF(*n*+*m*,*s*)

arg1 and *arg2* are evaluated and their results concatenated. In the result, the components derived from *arg1* will have lower indices than those derived from *arg2*.

5.16.22 conditional

Encoding number: 22

alt_label_intro:	LABEL
first:	EXP*x*
alt:	EXP*z*
	→ EXP (*x* LUB *z*)

first is evaluated. If *first* produces a result, *f*, this value is delivered as the result of the whole construct, and *alt* is not evaluated.

If **goto** (*alt_label_intro*) or any other jump (including **long_jump**) to *alt_label_intro* is obeyed during the evaluation of *first*, then the evaluation of *first* will stop, *alt* will be evaluated and its result delivered as the result of the construction.

The lifetime of *alt_label_intro* is the evaluation of *first*. *alt_label_intro* will not be used within *alt*.

The actual order of evaluation of the constituents shall be indistinguishable in all observable effects (apart from time) from evaluating all the obeyed parts of *first* before any obeyed part of *alt*. Note that this specifically includes any defined error handling.

For LUB see Section 7.26 on page 155.

5.16.23 contents

Encoding number: 23

s:	SHAPE
arg1:	EXP POINTER(*x*)
	→ EXP *s*

A value of SHAPEs will be extracted from the start of the space indicated by the pointer, and this is delivered.

x will include *alignment* (*s*).

s will not be a BITFIELD.

If the space which the pointer indicates does not lie wholly within the space indicated by the original pointer from which it is derived, the effect is undefined.

If the value delivered by *arg1* is a null pointer the effect is undefined.

Note: The constraint *x* will include *alignment*(*s*) ensures in the simple memory model (see Section 7.13 on page 147) that no change is needed to the POINTER.

5.16.24 contents_with_mode

Encoding number: 24

md:	TRANSFER_MODE
s:	SHAPE
arg1:	EXP POINTER(*x*)
	→ EXP *s*

A value of SHAPEs will be extracted from the start of the space indicated by the pointer, and this is delivered. The operation will be carried out as specified by the TRANSFER_MODE (see Section 5.33.19 on page 110).

If *md* consists of **standard_transfer_mode** only, then **contents_with_mode** is the same as **contents**.

x will include *alignment* (*s*).

s will not be a BITFIELD.

If the space which the pointer indicates does not lie wholly within the space indicated by the original pointer from which it is derived, the effect is undefined.

If the value delivered by *arg1* is a null pointer, the effect is undefined.

5.16.25 current_env

Encoding number: 25

→ EXP POINTER(*frame_alignment*)

A value of SHAPE POINTER (*fa*) is created and delivered. It gives access to the variables, identities and parameters in the current procedure activation which are declared as having ACCESS visible.

If the immediately enclosing procedure is defined by **make_general_proc,** then *fa* is the set union of **local_alignment** and the alignments of the kinds of parameters defined. That is to say, if there are caller parameters, then the alignment includes *callers_alignment*(*x*) where *x* is true if the PROCPROPS **var_callers** is present; if there are callee parameters, the alignment includes *callees_alignment*(*x*) where *x* is true if the PROCPROPS **var_callees** is present.

If the immediately enclosing procedure is defined by **make_proc,** then

fa = {*locals_alignment*, *callers_alignment(false)* }.

If an OFFSET produced by **env_offset** is added to a POINTER produced by **current_env** from an activation of the procedure which contains the declaration of the TAG used by **env_offset,** then the result is an original POINTER, notwithstanding the normal rules for **add_to_ptr** (see Section 7.15 on page 150).

If an OFFSET produced by **env_offset** is added to such a pointer from an inappropriate procedure, the effect is undefined.

5.16.26 div0

Encoding number: 26

div_by_zero_err:	ERROR_TREATMENT
ov_err:	ERROR_TREATMENT
arg1:	EXP INTEGER(*v*)
arg2:	EXP INTEGER(*v*)
	→ EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*, of the same VARIETY, *v*. Either the value *a* D1 *b* or the value *a* D2 *b* is delivered as the result of the construct, with the same SHAPE as the arguments. Different occurrences of **div0** in the same capsule can use D1 or D2 independently.

If *b* is zero, a *div_by_zero* error occurs and is handled by *div_by_zero_err*.

If *b* is not zero and the result cannot be expressed in the VARIETY being used to represent *v*, an overflow occurs and is handled by *ov_err*.

Producers may assume that shifting and **div0** by a constant which is a power of two yield equally good code.

See also Section 7.4 on page 140 for the definitions of D1, D2, M1 and M2.

5.16.27 div1

Encoding number: 27

div_by_zero_err:	ERROR_TREATMENT
ov_err:	ERROR_TREATMENT
arg1:	EXP INTEGER(*v*)
arg2:	EXP INTEGER(*v*)
	→ EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*, of the same VARIETY, *v*. The value *a* D1 *b* is delivered as the result of the construct, with the same SHAPE as the arguments.

If *b* is zero, a *div_by_zero* error occurs and is handled by *div_by_zero_err*.

If *b* is not zero and the result cannot be expressed in the VARIETY being used to represent *v*, an overflow occurs and is handled by *ov_err*.

Producers may assume that shifting and **div1** by a constant which is a power of two yield equally good code.

See also Section 7.4 on page 140 for the definitions of D1, D2, M1 and M2.

5.16.28 div2

Encoding number: 28

div_by_zero_err:	ERROR_TREATMENT
ov_err:	ERROR_TREATMENT
arg1:	EXP INTEGER(*v*)
arg2:	EXP INTEGER(*v*)
	→ EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*, of the same VARIETY, *v*. The value *a* D2 *b* is delivered as the result of the construct, with the same SHAPE as the arguments.

If *b* is zero, a *div_by_zero* error occurs and is handled by *div_by_zero_err*.

If *b* is not zero and the result cannot be expressed in the VARIETY being used to represent *v*, an overflow occurs and is handled by *ov_err*.

Producers may assume that shifting and **div2** by a constant which is a power of two yield equally good code if the lower bound of *v* is zero.

See also Section 7.4 on page 140 for the definitions of D1, D2, M1 and M2.

5.16.29 env_offset

Encoding number: 29

fa:	ALIGNMENT
y:	ALIGNMENT
t:	TAG*x*
	→ EXP OFFSET(*fa*,*y*)

t will be the tag of a variable, identify or procedure parameter with the visible property within a procedure defined by **make_general_proc** or **make_proc**.

If it is defined in a make_general_proc, let P be its associated PROCPROPS; otherwise let P be the PROCPROPS { *locals_alignment*, *caller_alignment* (*false*) }.

If *t* is the TAG of a variable or identify, *fa* will contain *locals_alignment*; if it is a caller parameter *fa* will contain a *caller_alignment* (*b*) where *b* is true if P contains **var_callers**; if it is a callee parameter, *fa* will contain a *callee_alignment* (*b*) where *b* is true if P contains **var_callees**.

If t is the TAG of a variable or parameter, the result is the OFFSET of its position, within any procedure environment which derives from the procedure containing the declaration of the variable or parameter, relative to its environment pointer. In this case *x* will be POINTER (*y*).

If *t* is the TAG of an identify, the result will be an OFFSET of space which holds the value. This pointer will not be used to alter the value. In this case *y* will be *alignment* (*x*).

See also Section 7.10 on page 144.

5.16.30 env_size

Encoding number: 30

proctag: TAG PROC
→ EXP OFFSET(*locals_alignment*,*{}*)

This delivers an OFFSET of a space sufficient to contain all the variables and identifications, explicit or implicit in the procedure identified by *proctag*. This will not include the space required for any **local_allocs** or procedure calls within the procedure.

proctag will be defined in the current CAPSULE by a TAGDEF identification of a **make_proc** or a **make_general_proc**.

5.16.31 fail_installer

Encoding number: 31

message: STRING(*k*,*n*)
→ EXP BOTTOM

Any attempt to use this operation to produce code will result in a failure of the installation process. A message will give information about the reason for this failure; this message should be passed to the installation manager.

5.16.32 float_int

Encoding number: 32

flpt_err: ERROR_TREATMENT
f: FLOATING_VARIETY
arg1: EXP INTEGER(*v*)
→ EXP FLOATING(*f*)

arg1 is evaluated to produce an integer value, which is converted to the representation of *f* and delivered.

If *f* is complex, the real part of the result will be derived from *arg1* and the imaginary part will be zero.

If there is a floating point error, it is handled by *flpt_err*. See also Section 7.21 on page 153.

5.16.33 floating_abs

Encoding number: 33

flpt_err: ERROR_TREATMENT
arg1: EXP FLOATING(*f*)
→ EXP FLOATING(*f*)

arg1 is evaluated and will produce a floating point value, *a*, of the FLOATING_VARIETY, *f*. The absolute value of *a* is delivered as the result of the construct, with the same SHAPE as the argument.

Though **floating_abs** cannot produce an overflow, it can give an invalid operand exception which is handled by *flpt_err*.

f will not be complex.

See also Section 7.23 on page 153.

5.16.34 floating_div

Encoding number: 34

flpt_err:	ERROR_TREATMENT
arg1:	EXP FLOATING(*f*)
arg2:	EXP FLOATING(*f*)
	→ EXP FLOATING(*f*)

arg1 and *arg2* are evaluated and will produce floating point values, *a* and *b*, of the same FLOATING_VARIETY, *f*. The value *a* / *b* is delivered as the result of the construct, with the same SHAPE as the arguments.

If there is a floating point error, it is handled by *flpt_err*.

See also Section 7.21 on page 153 and Section 7.23 on page 153.

5.16.35 floating_minus

Encoding number: 35

flpt_err:	ERROR_TREATMENT
arg1:	EXP FLOATING(*f*)
arg2:	EXP FLOATING(*f*)
	→ EXP FLOATING(*f*)

arg1 and *arg2* are evaluated and will produce floating point values, *a* and *b*, of the same FLOATING_VARIETY, *f*. The value *a* − *b* is delivered as the result of the construct, with the same SHAPE as the arguments.

If there is a floating point error it is handled by *flpt_err*.

See also Section 7.21 on page 153 and Section 7.23 on page 153

5.16.36 floating_maximum

Encoding number: 36

flpt_err:	ERROR_TREATMENT
arg1:	EXP FLOATING(*f*)
arg2:	EXP FLOATING(*f*)
	→ EXP FLOATING(*f*)

The maximum of the values delivered by *arg1* and *arg2* is the result. *f* will not be complex.

If *arg1* and *arg2* are incomparable, *flpt_err* will be invoked.

See also Section 7.23 on page 153.

5.16.37 floating_minimum

Encoding number: 37

flpt_err:	ERROR_TREATMENT
arg1:	EXP FLOATING(*f*)
arg2:	EXP FLOATING(*f*)
	→ EXP FLOATING(*f*)

The minimum of the values delivered by *arg1* and *arg2* is the result. *f* will not be complex.

If *arg1* and *arg2* are incomparable, *flpt_err* will be invoked.

See also Section 7.23 on page 153.

5.16.38 floating_mult

Encoding number: 38

flpt_err:	ERROR_TREATMENT
arg1:	LIST(EXP)
	→ EXP FLOATING(*f*)

The arguments, *arg1*, are evaluated, producing floating point values all of the same FLOATING_VARIETY, *f*. These values are multiplied in any order and the result of this multiplication is delivered as the result of the construct, with the same SHAPE as the arguments.

If there is a floating point error it is handled by *flpt_err*. See also Section 7.21 on page 153.

Note: Separate **floating_mult** operations cannot in general be combined, because rounding errors need to be controlled. The reason for allowing **floating_mult** to take a variable number of arguments is to make it possible to specify that a number of multiplications can be re-ordered.

If *arg1* contains one element, the result is the value of that element. There will be at least one element in *arg1*.

See also Section 7.23 on page 153.

5.16.39 floating_negate

Encoding number: 39

flpt_err:	ERROR_TREATMENT
arg1:	EXP FLOATING(*f*)
	→ EXP FLOATING(*f*)

arg1 is evaluated and will produce a floating point value, *a*, of the FLOATING_VARIETY, *f*. The value $-a$ is delivered as the result of the construct, with the same SHAPE as the argument.

Though **floating_negate** cannot produce an overflow, it can give an invalid operand exception which is handled by *flpt_err*.

See also Section 7.23 on page 153.

5.16.40 floating_plus

Encoding number: 40

flpt_err:	ERROR_TREATMENT
arg1:	LIST(EXP)
	→ EXP FLOATING(*f*)

The arguments, *arg1*, are evaluated, producing floating point values, all of the same FLOATING_VARIETY, *f*. These values are added in any order and the result of this addition is delivered as the result of the construct, with the same SHAPE as the arguments.

If there is a floating point error, it is handled by *flpt_err*.

See also Section 7.21 on page 153.

Note: Separate **floating_plus** operations cannot in general be combined, because rounding errors need to be controlled. The reason for allowing **floating_plus** to take a variable number of arguments is to make it possible to specify that a number of multiplications can be re-ordered.

If *arg1* contains one element, the result is the value of that element. There will be at least one element in *arg1*.

See also Section 7.23 on page 153.

5.16.41 floating_power

Encoding number: 41

flpt_err:	ERROR_TREATMENT
arg1:	EXP FLOATING(*f*)
arg2:	EXP INTEGER(*v*)
	→ EXP FLOATING(*f*)

The result of *arg1* is raised to the power given by *arg2*.

If there is a floating point error it is handled by *flpt_err*.

See also Section 7.21 on page 153 and Section 7.23 on page 153.

5.16.42 floating_test

Encoding number: 42

prob:	OPTION(NAT)
flpt_err:	ERROR_TREATMENT
nt:	NTEST
dest:	LABEL
arg1:	EXP FLOATING(*f*)
arg2:	EXP FLOATING(*f*)
	→ EXP TOP

arg1 and *arg2* are evaluated and will produce floating point values, *a* and *b*, of the same FLOATING_VARIETY, *f*. These values are compared using *nt*.

If *f* is complex then *nt* will be **equal** or **not_equal**.

If *a nt b,* this construction yields TOP. Otherwise control passes to *dest*.

If *prob* is present, *prob* / 100 gives the probability that control will continue to the next construct (that is, not pass to *dest*). If *prob* is absent, this probability is unknown.

If there is a floating point error it is handled by *flpt_err*.

See also Section 7.21 on page 153 and Section 7.23 on page 153.

5.16.43 goto

Encoding number: 43

dest:	LABEL
	→ EXP BOTTOM

Control passes to the EXP labelled *dest.* This construct will only be used where *dest* is in scope.

5.16.44 goto_local_lv

Encoding number: 44

arg1:	EXP POINTER({*code*})
	→ EXP BOTTOM

arg1 is evaluated. The label from which the value delivered by *arg1* was created will be within its lifetime, and this construction will be obeyed in the same activation of the same procedure as the creation of the POINTER({*code*}) by **make_local_lv.** Control passes to this activation of this LABEL.

If *arg1* delivers a null POINTER, the effect is undefined.

5.16.45 identify

Encoding number: 45

opt_access:	OPTION(ACCESS)
name_intro:	TAG*x*
definition:	EXP*x*
body:	EXP*y*
	→ EXP *y*

definition is evaluated to produce a value, *v.* Then body is evaluated. During this evaluation, *v* is bound to *name_intro.* This means that inside body an evaluation of *obtain_tag* (*name_intro*) will produce the value *v*.

The value delivered by **identify** is that produced by body.

The TAG given for *name_intro* will not be reused within the current UNIT. No rules for the hiding of one TAG by another are given; this will not happen. The lifetime of *name_intro* is the evaluation of body.

If *opt_access* contains *visible*, it means that the value must not be aliased while the procedure containing this declaration is not the current procedure. Hence if there are any copies of this value, they will need to be refreshed when the procedure is returned to. The easiest implementation when *opt_access* is *visible* may be to keep the value in memory, but this is not a necessary requirement.

The order in which the constituents of *definition* and *body* are evaluated shall be indistinguishable in all observable effects (apart from time) from completely evaluating definition before starting body. See the note about order in Section 5.16.107 on page 76.

5.16.46 ignorable

Encoding number: 46

arg1: EXP *x*
→ EXP *x*

If the result of this construction is discarded, *arg1* need not be evaluated, though evaluation is permitted. If the result is used, it is the result of *arg1*.

5.16.47 imaginary_part

Encoding number: 47

arg1: EXP*c*
→ EXP FLOATING(*float_of_complex(c)*)

c will be complex. This delivers the imaginary part of the value produced by *arg1*.

5.16.48 initial_value

Encoding number: 48

init: EXP *s*
→ EXP *s*

Any tag used as an argument of an **obtain_tag** in *init* will be global or defined within *init*.

All labels used in *init* will be defined within *init*.

init will be evaluated once only before any procedure application, other than those involved in this or other **initial_value** constructions, but after all load-time constant initialisations of TAGDEFs. The result of this evaluation is the value of the construction.

The order of evaluation of the different **initial_value**s in a program is undefined.

See also Section 7.29 on page 156.

5.16.49 integer_test

Encoding number: 49

prob: OPTION(NAT)
nt: NTEST
dest: LABEL

arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*v*)
→ EXP TOP

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*, of the same VARIETY, *v*. These values are compared using *nt*.

If *a nt b*, this construction yields TOP. Otherwise control passes to *dest*.

If *prob* is present, *prob* / 100 gives the probability that control will continue to the next construct (that is, not pass to *dest*). If *prob* is absent, this probability is unknown.

5.16.50 labelled

Encoding number: 50

placelabs_intro: LIST(LABEL)
starter: EXP*x*
places: LIST(EXP)
→ EXP *w*

The lists *placelabs_intro* and *places* have the same number of elements.

To evaluate the construction, *starter* is evaluated. If its evaluation runs to completion producing a value, then this is delivered as the result of the whole construction. If a **goto** to one of the LABELS in *placelabs_intro*, or any other jump to one of these LABELS, is evaluated, then the evaluation of *starter* stops and the corresponding element of *places* is evaluated. In the canonical ordering, all the operations which are evaluated from *starter* are completed before any from an element of *places* is started. If the evaluation of the member of *places* produces a result, this is the result of the construction.

If a jump to any of the *placelabs_intro* is obeyed then evaluation continues similarly. Such jumping may continue indefinitely, but if any *places* terminates, then the value it produces is the value delivered by the construction.

The SHAPE *w* is the LUB of *x* and all the *places*. See Section 7.26 on page 155.

The actual order of evaluation of the constituents shall be indistinguishable in all observable effects (apart from time) from that described above. Note that this specifically includes any defined error handling.

The lifetime of each of the LABELs in *placelabs_intro* is the evaluation of *starter* and all the elements of *places*.

5.16.51 last_local

Encoding number: 51

x: EXP OFFSET(*y*,*z*)
→ EXP POINTER(*alloca_alignment*)

If the last use of **local_alloc** in the current activation of the current procedure was after the last use of **local_free** or **local_free_all,** then the value returned is the last POINTER allocated with **local_alloc**.

If the last use of **local_free** in the current activation of the current procedure was after the last use of **local_alloc,** then the result is the POINTER last allocated which is still active.

The result POINTER will have been created by **local_alloc** with the value of its *arg1* equal to the value of *x*.

If the last use of **local_free_all** in the current activation of the current procedure was after the last use of **local_alloc,** or if there has been no use of **local_alloc** in the current activation of the current procedure, then the result is undefined.

The ALIGNMENT *alloca_alignment* includes the set union of all the ALIGNMENTs which can be produced by **alignment** from any SHAPE. See Section 7.13.4 on page 149.

5.16.52 local_alloc

Encoding number: 52

arg1: EXP OFFSET(*x*,*y*)
→ EXP POINTER(*alloca_alignment*)

The *arg1* expression is evaluated and space is allocated sufficient to hold a value of the given size. The result is an original pointer to this space.

x will not consist entirely of bitfield alignments.

The initial contents of the space are not specified.

This allocation is as if on the stack of the current procedure, and the lifetime of the pointer ends when the current activation of the current procedure ends with a **return**, **return_to_label** or **tail_call** or if there is a long jump out of the activation. Any use of the pointer thereafter is undefined. Note the specific exclusion of the procedure ending with **untidy_return**; in this case the calling procedure becomes the current activation.

The uses of **local_alloc** within the procedure are ordered dynamically as they occur, and this order affects the meaning of **local_free** and **last_local**.

arg1 may be a zero OFFSET. In this case, suppose the result is *p*; then a subsequent use in the current procedure activation of

local_free (*offset_zero* (*alloca_alignment*) *p*)

will return the *alloca* stack to the state it was in immediately before the use of **local_alloc**.

Note that if a procedure which uses **local_alloc** is inlined, it may be necessary to use **local_free** to get the correct semantics.

See also Section 7.12 on page 146.

5.16.53 local_alloc_check

Encoding number: 53

arg1: EXP OFFSET(*x*,*y*)
→ EXP POINTER(*alloca_alignment*)

If the OFFSET *arg1* can be accommodated within the limit of the *local_alloc* stack (see Section 5.16.108 on page 76), the action is precisely the same as **local_alloc**.

If not, normal action is stopped and an XANDF exception is raised with ERROR_CODE *stack_overflow*.

5.16.54 local_free

Encoding number: 54

a: EXP OFFSET(*x*,*y*)
p EXP POINTER(*alloca_alignment*)
→ EXP TOP

The POINTER, *p*, will be an original pointer to space allocated by **local_alloc** within the current call of the current procedure. It and all spaces allocated after it by **local_alloc** will no longer be used. This POINTER will have been created by **local_alloc** with the value of its *arg1* equal to the value of *a*.

Any subsequent use of pointers to the spaces no longer used will be undefined.

5.16.55 local_free_all

Encoding number: 55

→ EXP TOP

Every space allocated by **local_alloc** within the current call of the current procedure will no longer be used.

Any use of a pointer to space allocated before this operation within the current call of the current procedure is undefined.

Note that if a procedure which uses **local_free_all** is inlined, it may be necessary to use **local_free** to get the correct semantics.

5.16.56 long_jump

Encoding number: 56

arg1: EXP POINTER(*fa*)
arg2: EXP POINTER({*code*})
→ EXP BOTTOM

arg1 will be a pointer produced by an application of *curent_env* in a currently active procedure.

The frame produced by *arg1* is reinstated as the current procedure. This frame will still be active. Evaluation recommences at the label given by *arg2*. This operation will only be used during the lifetime of that label.

Only TAGs declared to have **long_jump_access** will be defined at the re-entry.

If *arg2* delivers a null POINTER({ *code* }), the effect is undefined.

5.16.57 make_complex

Encoding number: 57

c:	FLOATING_VARIETY
arg1:	EXP FLOATING(*f*)
arg2:	EXP FLOATING(*f*)
	→ EXP FLOATING(*c*)

c will be complex and derived from the same parameters as *f*.

This delivers a complex number with *arg1* delivering the real part and *arg2* the imaginary part.

5.16.58 make_compound

Encoding number: 58

arg1:	EXP OFFSET(*base*,*y*)
arg2:	LIST(EXP)
	→ EXP COMPOUND(*arg1*)

Let the i^{th} component (*i* starts at one) of *arg2* be *x* [*i*]. The list may be empty.

The components *x* [2 * *k*] are values which are to be placed at OFFSETs given by *x* [2 * *k* - 1]. These OFFSETs will be constants and non-negative.

The OFFSET *x* [2 * *k* - 1] will have the SHAPE OFFSET (z_k, *alignment* (*shape* (*x* [2 * *k*]))), where *shape* gives the SHAPE of the component and *base* includes z_k.

arg1 will be a constant non-negative OFFSET. See also Section 5.16.87 on page 69.

The values *x* [2 * *k* - 1] will be such that the components when in place either do not overlap or exactly coincide, in the sense that the OFFSETS are equal and the values have the same SHAPE. If they coincide, the corresponding values *x* [2 * *k*] will have VARIETY SHAPES and will be logically ‘‘OR’d’’ together.

The SHAPE of a *x* [2 * *k*] component can be TOP. In this case, the component is evaluated but no value is placed at the corresponding OFFSET.

If *x* [2 * *k*] is a BITFIELD, then *x* [2 * *k* - 1], *shape* (*x* [2* *k*]) will be *variety_enclosed* (see Section 7.24 on page 154).

5.16.59 make_floating

Encoding number: 59

f:	FLOATING_VARIETY
rm:	ROUNDING_MODE
negative:	BOOL
mantissa:	STRING(*k*,*n*)
base:	NAT
exponent:	SIGNED_NAT
	→ EXP FLOATING(*f*)

f will not be complex.

mantissa will be a STRING of 8-bit integers, each of which is either 46 or is greater than or equal to 48. Those values, *c*, which lie between 48 and 63 will represent the digit $c - 48$. A decimal point is represented by 46.

The BOOL *negative* determines the sign of the result: if true, the result will be negative; if false, it will be positive.

A floating point number, mantissa *($base^{exponent}$) is created and rounded to the representation of *f* as specified by *rm*.

rm will not be *round_as_state*. *mantissa* is read as a sequence of digits to base *base* and may contain one point symbol.

base will be one of the numbers 2, 4, 8, 10, 16. Note that in base 16, the digit 10 is represented by the character number 58, and so on.

The result will lie in *f*.

5.16.60 make_general_proc

Encoding number: 60

result_shape:	SHAPE
prcprops:	OPTION(PROCPROPS)
caller_intro:	LIST(TAGSHACC)
callee_intro:	LIST(TAGSHACC)
body:	EXP BOTTOM
	→ EXP PROC

Evaluation of **make_general_proc** delivers a PROC. When this procedure is applied to parameters using **apply_general_proc,** space is allocated to hold the actual values of the parameters *caller_intro* and *callee_intro.* The values produced by the actual parameters are used to initialise these spaces. Then *body* is evaluated. During this evaluation the TAGS in *caller_intro* and *callee_intro* are bound to original POINTERs to these spaces. The lifetime of these TAGS is the evaluation of *body*.

The SHAPE of *body* will be BOTTOM. *caller_intro* and *callee_intro* may be empty.

The TAGs introduced in the parameters will not be reused within the current UNIT.

The SHAPES in the parameters specify the SHAPE of the corresponding TAGS.

The OPTION(ACCESS) (in *params_intro*) specifies the ACCESS properties of the corresponding parameter, just as for a variable declaration.

In *body*, the only TAGs which may be used as an argument of **obtain_tag** are those which are declared by **identify** or **variable** constructions in *body* and which are in scope, or TAGs which are declared by **make_id_tagdef**, **make_var_tagdef** or **common_tagdef**, or are in *caller_intro* or *callee_intro.* If a **make_proc** occurs in *body*, its TAGs are not in scope.

The argument of every **return** or **untidy_return** construction in *body* will have SHAPE *result_shape*. Every **apply_general_proc** using the procedure will specify the SHAPE of its result to be *result_shape*.

The presence or absence of each of the PROC PROPS **var_callers**, **var_callees, check_stack** and **untidy** in *prcprops* will be reflected in every **apply_general_proc** or **tail_call** on this procedure.

The definition of the canonical ordering of the evaluation of **apply_general_proc** gives the definition of these PROCPROPS.

If *prcprocs* contains **check_stack,** an XANDF exception will be raised if the static space required for the procedure call (in the sense of **env_size**) would exceed the limit given by **set_stack_limit**.

If *prcprops* contains **no_long_jump_dest,** the body of the procedure will never contain the destination label of a **long_jump**.

For notes on the intended implementation of procedures, see Section 7.9 on page 143.

5.16.61 make_int

Encoding number: 61

v: VARIETY
value: SIGNED_NAT
→ EXP INTEGER(*v*)

An integer value is delivered, the value of which is given by *value*, and the VARIETY by *v*. The SIGNED_NAT value will lie between the bounds of *v*.

5.16.62 make_local_lv

Encoding number: 62

lab: LABEL
→ EXP POINTER({*code*})

A POINTER({ *code* }) *lv* is created and delivered. It can be used as an argument to **goto_local_lv** or **long_jump.** If and when one of these is evaluated with *lv* as an argument, control will pass to *lab*.

5.16.63 make_nof

Encoding number: 63

arg1: LIST(EXP)
→ EXP NOF(*n*,*s*)

This creates an array of *n* values of SHAPEs, containing the given values produced by evaluating the members of *arg1* in the same order as they occur in the list.

n will not be zero.

5.16.64 make_nof_int

Encoding number: 64

v: VARIETY
str: STRING(*k*,*n*)
→ EXP NOF(*n*, INTEGER(*v*))

An NOF INTEGER is delivered. The conversions are carried out as if the elements of *str* were INTEGER (*var_limits*(*0, 2 k* − *1*)). *n* may be zero.

5.16.65 make_null_local_lv

Encoding number: 65

→ EXP POINTER({*code*})

Makes a null POINTER ({ *code* }) which can be detected by **pointer_test.** The effect of **goto_local_lv** or **long_jump** applied to this value is undefined.

All null POINTER ({ *code* }) are equal to each other and unequal to any other POINTERS.

5.16.66 make_null_proc

Encoding number: 66

→ EXP PROC

A null PROC is created and delivered. The null PROC may be tested for by using **proc_test.** The effect of using it as the first argument of **apply_proc** is undefined.

All null PROC are equal to each other and unequal to any other PROC.

5.16.67 make_null_ptr

Encoding number: 67

a: ALIGNMENT
→ EXP POINTER(*a*)

A null POINTER (*a*) is created and delivered. The null POINTER may be tested for by **pointer_test**.

a will not include code.

All null POINTER (*x*) are equal to each other and unequal to any other POINTER (*x*).

5.16.68 make_proc

Encoding number: 68

result_shape: SHAPE
params_intro: LIST(TAGSHACC)
var_intro: OPTION(TAGACC)
body: EXP BOTTOM
→ EXP PROC

Evaluation of **make_proc** delivers a PROC. When this procedure is applied to parameters using **apply_proc,** space is allocated to hold the actual values of the parameters *params_intro* and *var_intro* (if present). The values produced by the actual parameters are used to initialise these spaces. Then *body* is evaluated. During this evaluation the TAGS in *params_intro* and *var_intro* are bound to original POINTERs to these spaces. The lifetime of these TAGS is the evaluation of *body*.

If *var_intro* is present then all uses of **apply_proc** which have the effect of calling this procedure will have their *var_param* option present. The ALIGNMENT **var_param_alignment** includes the set union of all the ALIGNMENTs which can be produced by **alignment** from any SHAPE. Note that *var_intro* does not contain an ACCESS component and so cannot be marked visible. Hence it is not a possible argument of **env_offset.** If present, *var_intro* is an original pointer.

The SHAPE of body will be BOTTOM. *params_intro* may be empty.

The TAGs introduced in the parameters will not be reused within the current UNIT.

The SHAPES in the parameters specify the SHAPE of the corresponding TAGS.

The OPTION(ACCESS) (in *params_intro*) specifies the ACCESS properties of the corresponding parameter, just as for a variable declaration.

In *body*, the only TAGs which may be used as an argument of **obtain_tag** are those which are declared by **identify** or **variable** constructions in *body* and which are in scope, or TAGs which are declared by **make_id_tagdef**, **make_var_tagdef** or **common_tagdef**, or are in *params_intro* or *var_intro.* If a **make_proc** occurs in *body*, its TAGs are not in scope.

The argument of every **return** construction in *body* will have SHAPE *result_shape.* Every **apply_proc** using the procedure will specify the SHAPE of it result to be *result_shape*.

For notes on the intended implementation of procedures, see Section 7.9 on page 143.

5.16.69 make_stack_limit

Encoding number: 116

stack_base:	EXP POINTER(*fa*)
frame_size:	EXP OFFSET(*locals_alignment*,*x*)
alloc_size:	EXP OFFSET(*alloca_alignment*,*y*)
	→ EXP POINTER(*fb*)

This creates a POINTER suitable for use with **set_stack_limit**.

fa and *fb* will include *locals_alignment* and, if *alloc_size* is not the zero offset, will also contain *alloca_alignment* .

The result will be the same as if given by the following:

> Assume *stack_base* is the current frame-pointer as given by **current_env** in a hypothetical procedure P with *env_size* equal to *frame_size* and which has generated *alloc_size* by a **local_alloc.** If P then calls Q, the result will be the same as that of a **current_env** performed immediately in the body of Q.
>
> If the following construction is performed:
>
> *set_stack_limit(make_stack_limit(current_env,F,A))*
>
> the *frame* space and *local_alloc* space that would be available for use by this supposed call of Q will not be reused by procedure calls with **check_stack** or uses of **local_alloc_check** after the **set_stack_limit**. Any attempt to do so will raise an XANDF exception, *stack_overflow*.

5.16.70 make_top

Encoding number: 69

$\rightarrow$ EXP TOP

make_top delivers a value of SHAPE TOP (that is, void).

5.16.71 make_value

Encoding number: 70

s: SHAPE
$\rightarrow$ EXP *s*

This EXP creates some value with the representation of the SHAPE *s*. This value will have the correct size, but its representation is not specified. It can be assigned, be the result of a contents, a parameter or result of a procedure, or the result of any construction (like **sequence**) which delivers the value delivered by an internal EXP. However, if it is used for arithmetic or as a POINTER for taking **contents** or **add_to_ptr**, and so on, the effect is undefined.

Installers will usually be able to implement this operation by producing no code.

Note: A floating point NAN is a possible value for this purpose.

The SHAPE *s* will not be BOTTOM.

5.16.72 maximum

Encoding number: 71

arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*v*)
$\rightarrow$ EXP INTEGER(*v*)

The arguments will be evaluated and the maximum of the values delivered is the result.

5.16.73 minimum

Encoding number: 72

arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*v*)
$\rightarrow$ EXP INTEGER(*v*)

The arguments will be evaluated and the minimum of the values delivered is the result.

5.16.74 minus

Encoding number: 73

ov_err:	ERROR_TREATMENT
arg1:	EXP INTEGER(*v*)
arg2:	EXP INTEGER(*v*)
	→ EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*, of the same VARIETY, *v*. The difference $a - b$ is delivered as the result of the construct, with the same SHAPE as the arguments.

If the result cannot be expressed in the VARIETY being used to represent *v*, an overflow error is caused and is handled in the way specified by **ov_err**.

5.16.75 move_some

Encoding number: 74

md:	TRANSFER_MODE
arg1:	EXP POINTER(*x*)
arg2:	EXP POINTER(*yx*)
arg3:	EXP OFFSET(*z*,*t*)
	→ EXP TOP

The arguments are evaluated to produce *p1*, *p2*, and *sz* respectively. A quantity of data measured by *sz* in the space indicated by *p1* is moved to the space indicated by *p2*. The operation will be carried out as specified by the TRANSFER_MODE (see Section 5.33.19 on page 110).

x will include *z* and *y* will include *z*.

sz will be a non-negative OFFSET (see Section 5.16.87 on page 69).

If the spaces of size *sz* to which *p1* and *p2* point do not lie entirely within the spaces indicated by the original pointers from which they are derived, the effect of the operation is undefined.

If the value delivered by *arg1* or *arg2* is a null pointer, the effect is undefined.

See also Section 7.16 on page 151 and Section 7.17 on page 151.

5.16.76 mult

Encoding number: 75

ov_err:	ERROR_TREATMENT
arg1:	EXP INTEGER(*v*)
arg2:	EXP INTEGER(*v*)
	→ EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*, of the same VARIETY, *v*. The product $a * b$ is delivered as the result of the construct, with the same SHAPE as the arguments.

If the result cannot be expressed in the VARIETY being used to represent *v*, an overflow error is caused and is handled in the way specified by **ov_err**.

5.16.77 n_copies

Encoding number: 76

n:	NAT
arg1:	EXP*x*
	→ EXP NOF(*n*,*x*)

arg1 is evaluated and an NOF value is delivered which contains *n* copies of this value. *n* can be zero or one or greater.

Producers are encouraged to use **n_copies** to initialise arrays of known size.

5.16.78 negate

Encoding number: 77

ov_err:	ERROR_TREATMENT
arg1:	EXP INTEGER(*v*)
	→ EXP INTEGER(*v*)

arg1 is evaluated and will produce an integer value, *a*. The value $-a$ is delivered as the result of the construct, with the same SHAPE as the argument.

If the result cannot be expressed in the VARIETY being used to represent *v*, an overflow error is caused and is handled in the way specified by **ov_err**.

5.16.79 not

Encoding number: 78

arg1:	EXP INTEGER(*v*)
	→ EXP INTEGER(*v*)

The argument is evaluated producing an integer value, of VARIETY *v*. The result is the bitwise logical NOT of this value in the representing VARIETY. The result is delivered as the result of the construct, with the same SHAPE as the arguments.

See also Section 7.18 on page 151.

5.16.80 obtain_tag

Encoding number: 79

t:	TAG *x*
	→ EXP *x*

The value with which the TAG *t* is bound is delivered. The SHAPE of the result is the SHAPE of the value with which the TAG is bound.

5.16.81 offset_add

Encoding number: 80

arg1: EXP OFFSET(*x*,*y*)
arg2: EXP OFFSET(*z*,*t*)
→ EXP OFFSET(*x*,*t*)

The two arguments deliver OFFSETs. The result is the sum of these OFFSETs, as an OFFSET.

y will include *z*.

Notes:

1. The effect of the constraint ‘‘*y* will include *z*’’is that, in the simple representation of pointer arithmetic (see Section 7.13 on page 147), this operation can be represented by addition.
2. **offset_add** can lose information, so that **offset_subtract** does not have the usual relation with it.

5.16.82 offset_div

Encoding number: 81

v: VARIETY
arg1: EXP OFFSET(*x*,*x*)
arg2: EXP OFFSET(*x*,*x*)
→ EXP INTEGER(*v*)

The two arguments deliver OFFSETs, *a* and *b*. The result is *a/b*, as an INTEGER of VARIETY *v*. Division is interpreted in the same sense (with respect to remainder) as in **div0**.

The value produced by *arg2* will be non-zero.

5.16.83 offset_div_by_int

Encoding number: 82

arg1: EXP OFFSET(*x*,*x*)
arg2: EXP INTEGER(*v*)
→ EXP OFFSET(*x*,*x*)

The result is the OFFSET produced by *arg1* divided by *arg2*, as an OFFSET (*x*, *x*).

The value produced by *arg2* will be greater than zero.

The following identity will apply for all *A* and *n*:

offset_mult(**offset_div_by_int**(*A*, *n*), *n*) = *A*

5.16.84 offset_max

Encoding number: 83

arg1: EXP OFFSET(*x*,*y*)
arg2: EXP OFFSET(*z*,*y*)
→ EXP OFFSET(*unite_alignments(x,z)*,*y*)

The two arguments deliver OFFSETs. The result is the maximum of these OFFSETs, as an OFFSET.

See Section 7.13.2 on page 148.

Note: In the simple memory model (see Section 7.13 on page 147), this operation is represented by maximum. The constraint that the second ALIGNMENT parameters are both *y* is to permit the representation of OFFSETS in installers by a simple homomorphism.

5.16.85 offset_mult

Encoding number: 84

arg1: EXP OFFSET(*x*,*x*)
arg2: EXP INTEGER(*v*)
→ EXP OFFSET(*x*,*x*)

The first argument gives an OFFSET, *off*, and the second an integer, *n*. The result is the product of these, as an offset.

The result shall be equal to ‘‘offset_adding’’ *off* a total of *n* times to **offset_zero***(x)*.

5.16.86 offset_negate

Encoding number: 85

arg1: EXP OFFSET(*x*,*x*)
→ EXP OFFSET(*x*,*x*)

The inverse of the argument is delivered.

In the simple memory model (see Section 7.13 on page 147), this can be represented by **negate**.

5.16.87 offset_pad

Encoding number: 86

a: ALIGNMENT
arg1: EXP OFFSET(*z*,*t*)
→ EXP OFFSET(*unite_alignments(z,a)*,*a*)

arg1 is evaluated giving *off*. The next greater or equal OFFSET at which a value of ALIGNMENT *a* can be placed is delivered. That is, there shall not exist an OFFSET of the same SHAPE as the result which is greater than or equal to *off* and less than the result, in the sense of **offset_test**.

off will be a non-negative OFFSET, that is it will be greater than or equal to a zero OFFSET of the same SHAPE in the sense of **offset_test**.

Note: In the simple memory model (see Section 7.13 on page 147), this operation can be represented by *((off + a – 1) / a) * a*. In the simple model, this is the only operation which is not represented by a simple corresponding integer operation.

5.16.88 offset_subtract

Encoding number: 87

arg1: EXP OFFSET(*x*,*y*)
arg2: EXP OFFSET(*x*,*z*)
→ EXP OFFSET(*z*,*y*)

The two arguments deliver offsets, *p* and *q*. The result is *p – q*, as an offset.

Note that *x* will include *y*, *x* will include *z* and *z* will include *y*, by the constraints on OFFSETS.

Note: **offset_subtract** and **offset_add** do not have the conventional relationship because **offset_add** can lose information, which cannot be regenerated by **offset_subtract**.

5.16.89 offset_test

Encoding number: 88

prob: OPTION(NAT)
nt: NTEST
dest: LABEL
arg1: EXP OFFSET(*x*,*y*)
arg2: EXP OFFSET(*x*,*y*)
→ EXP TOP

arg1 and *arg2* are evaluated and will produce offset values, *a* and *b*. These values are compared using *nt*.

If *a nt b*, this construction yields TOP. Otherwise control passes to *dest*.

If *prob* is present, *prob/100* gives the probability that control will continue to the next construct (that is, not pass to *dest*). If *prob* is absent this probability is unknown.

a **greater_than_or_equal** *b* is equivalent to **offset_max** *(a,b)* = *a*, and similarly for the other comparisons.

In the simple memory model (see Section 7.13 on page 147), this can be represented by **integer_test**.

5.16.90 offset_zero

Encoding number: 89

a: ALIGNMENT
→ EXP OFFSET(*a*,*a*)

A zero offset of SHAPE OFFSET (*a*, a).

offset_pad(*b*, **offset_zero**(*a*)) is a zero offset of SHAPE OFFSET **(unite_alignments**(*a*, *b*), *b*).

5.16.91 or

Encoding number: 90

arg1:	EXP INTEGER(*v*)
arg2:	EXP INTEGER(*v*)
	→ EXP INTEGER(*v*)

The arguments are evaluated, producing integer values of the same VARIETY, *v*. The result is the bitwise logical OR of these two integers in the representing VARIETY. The result is delivered as the result of the construct, with the same SHAPE as the arguments.

See also Section 7.18 on page 151.

5.16.92 plus

Encoding number: 91

ov_err:	ERROR_TREATMENT
arg1:	EXP INTEGER(*v*)
arg2:	EXP INTEGER(*v*)
	→ EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*, of the same VARIETY, *v*. The sum *a* + *b* is delivered as the result of the construct, with the same SHAPE as the arguments.

If the result cannot be expressed in the VARIETY being used to represent *v*, an overflow error is caused and is handled in the way specified by **ov_err**.

5.16.93 pointer_test

Encoding number: 92

prob:	OPTION(NAT)
nt:	NTEST
dest:	LABEL
arg1:	EXP POINTER(*x*)
arg2:	EXP POINTER(*x*)
	→ EXP TOP

arg1 and *arg2* are evaluated and will produce pointer values, *a* and *b*, which will be derived from the same original pointer. These values are compared using *nt*.

If *a nt b*, this construction yields TOP. Otherwise control passes to *dest*.

If *prob* is present, *prob/100* gives the probability that control will continue to the next construct (that is, not pass to *dest*). If *prob* is absent, this probability is unknown.

The effect of this construction is the same as:

offset_test*(prob, nt, dest,* **subtract_ptrs***(arg1, arg2),* **offset_zero***(x))*

Note: In the simple memory model (see Section 7.13 on page 147), this construction can be represented by **integer_test**.

5.16.94 power

Encoding number: 93

ov_err: ERROR_TREATMENT
arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*w*)
→ EXP INTEGER(*v*)

arg2 will be non-negative. The result is the result of *arg1* raised to the power given by *arg2*.

If the result cannot be expressed in the VARIETY being used to represent *v*, an overflow error is caused and is handled in the way specified by **ov_err**.

5.16.95 proc_test

Encoding number: 94

prob: OPTION(NAT)
nt: NTEST
dest: LABEL
arg1: EXP PROC
arg2: EXP PROC
→ EXP TOP

arg1 and *arg2* are evaluated and will produce PROC values, *a* and *b*. These values are compared using *nt*. The only permitted values of *nt* are **equal** and **not_equal**.

If *a nt b*, this construction yields TOP. Otherwise control passes to *dest*.

If *prob* is present, *prob/100* gives the probability that control will continue to the next construct (that is, not pass to *dest*). If *prob* is absent this probability is unknown.

Two PROCs are equal if they were produced by the same instantiation of **make_proc** or if they were both made with **make_null_proc**. Otherwise they are unequal.

5.16.96 profile

Encoding number: 95

uses: NAT
→ EXP TOP

The integer *uses* gives the number of times which this construct is expected to be evaluated.

All uses of **profile** in the same capsule are to the same scale. They will be mutually consistent.

5.16.97 real_part

Encoding number: 96

arg1:	EXP*c*
	→ EXP FLOATING(*float_of_complex(c)*)

c will be complex. This delivers the real part of the value produced by *arg1*.

5.16.98 rem0

Encoding number: 97

div_by_zero_err:	ERROR_TREATMENT
ov_err:	ERROR_TREATMENT
arg1:	EXP INTEGER(*v*)
arg2:	EXP INTEGER(*v*)
	→ EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*, of the same VARIETY, *v*. The value *a* M1 *b* or the value *a* M2 *b* is delivered as the result of the construct, with the same SHAPE as the arguments. Different occurrences of **rem0** in the same capsule can use M1 or M2 independently.

The following equivalence shall hold:

x = plus(mult(div0(x, y), y), rem0(x, y))

if all the ERROR_TREATMENTS are impossible, and *x* and *y* have no side effects.

If *b* is zero, a *div_by_zero* error occurs and is handled by *div_by_zero_err*.

If *b* is not zero and **div0** (*a*,*b*) cannot be expressed in the VARIETY being used to represent *v*, an overflow may occur, in which case it is handled by *ov_err*.

Producers may assume that suitable masking and **rem0** by a power of two yield equally good code.

See Section 7.4 on page 140 for the definitions of D1, D2, M1 and M2.

5.16.99 rem1

Encoding number: 98

div_by_zero_err:	ERROR_TREATMENT
ov_err:	ERROR_TREATMENT
arg1:	EXP INTEGER(*v*)
arg2:	EXP INTEGER(*v*)
	→ EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*, of the same VARIETY, *v*. The value *a* M1 *b* is delivered as the result of the construct, with the same SHAPE as the arguments.

If *b* is zero, a *div_by_zero* error occurs and is handled by *div_by_zero_err*.

If *b* is not zero and **div1** (*a*,*b*) cannot be expressed in the VARIETY being used to represent *v*, an overflow may occur, in which case it is handled by *ov_err*.

Producers may assume that suitable masking and **rem1** by a power of two yield equally good code.

See Section 7.4 on page 140 for the definitions of D1, D2, M1 and M2.

5.16.100 rem2

Encoding number: 99

div_by_zero_err:	ERROR_TREATMENT
ov_err:	ERROR_TREATMENT
arg1:	EXP INTEGER(*v*)
arg2:	EXP INTEGER(*v*)
	→ EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b,* of the same VARIETY, *v*. The value *a* M2 *b* is delivered as the result of the construct, with the same SHAPE as the arguments.

If *b* is zero, a *div_by_zero* error occurs and is handled by *div_by_zero_err*.

If *b* is not zero and **div2** (*a,b*) cannot be expressed in the VARIETY being used to represent *v*, an overflow may occur, in which case it is handled by *ov_err*.

Producers may assume that suitable masking and **rem2** by a power of two yield equally good code if the lower bound of *v* is zero.

See Section 7.4 on page 140 for the definitions of D1, D2, M1 and M2.

5.16.101 repeat

Encoding number: 100

repeat_label_intro:	LABEL
start:	EXP TOP
body:	EXP*y*
	→ EXP *y*

start is evaluated. Then *body* is evaluated.

If *body* produces a result, this is the result of the whole construction. However if **goto** or any other jump to *repeat_label_intro* is encountered during the evaluation then the current evaluation stops and *body* is evaluated again. In the canonical order, all evaluated components are completely evaluated before any of the next iteration of *body*. The lifetime of *repeat_label_intro* is the evaluation of *body*.

The actual order of evaluation of the constituents shall be indistinguishable in all observable effects (apart from time) from that described above. Note that this specifically includes any defined error handling.

5.16.102 return

Encoding number: 101

arg1: EXP *x*
→ EXP BOTTOM

arg1 is evaluated to produce a value *v*. The evaluation of the immediately enclosing procedure ceases and *v* is delivered as the result of the procedure.

Since the **return** construct can never produce a value, the SHAPE of its result is BOTTOM.

All uses of **return** in the *body* of a **make_proc** or **make_general_proc** will have *arg1* with the same SHAPE.

5.16.103 return_to_label

Encoding number: 102

lab_val: EXP POINTER(*code_alignment*)
→ EXP BOTTOM

lab_val will be a label value in the calling procedure.

The evaluation of the immediately enclosing procedure ceases and control is passed to the calling procedure at the label given by *lab_val*.

5.16.104 round_with_mode

Encoding number: 103

flpt_err: ERROR_TREATMENT
mode: ROUNDING_MODE
r: VARIETY
arg1: EXP FLOATING(*f*)
→ EXP INTEGER(*r*)

arg1 is evaluated to produce a floating point value, *v*. This is rounded to an integer of VARIETY *r*, using the ROUNDING_MODE *mode*. This is the result of the construction.

If *f* is complex, the result is derived from the real part of *arg1*.

If there is a floating point error, it is handled by *flpt_err*. See also Section 7.21 on page 153.

5.16.105 rotate_left

Encoding number: 104

arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*w*)
→ EXP INTEGER(*v*)

The value delivered by *arg1* is rotated left *arg2* places.

arg2 will be non-negative.

The effect of rotating more places than the number of bits needed to represent *v* is undefined .

The use of this construct assumes knowledge of the representational variety of *v*.

5.16.106 rotate_right

Encoding number: 105

arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*w*)
→ EXP INTEGER(*v*)

The value delivered by *arg1* is rotated right *arg2* places.

arg2 will be non-negative.

The effect of rotating more places than the number of bits needed to represent *v* is undefined.

The use of this construct assumes knowledge of the representational variety of *v*.

5.16.107 sequence

Encoding number: 106

statements: LIST(EXP)
result: EXP*x*
→ EXP *x*

The statements are evaluated in the same order as the list, *statements*, and their results are discarded. Then *result* is evaluated and its result forms the result of the construction.

A canonical order is one in which all the components of each statement are completely evaluated before any component of the next statement is started. A similar constraint applies between the last statement and *result*. The actual order in which the statements and their components are evaluated shall be indistinguishable in all observable effects (apart from time) from a canonical order.

Note: This ordering specifically includes any defined error handling. However, if in any canonical order the effect of the program is undefined, the actual effect of the sequence is undefined.

Hence constructions with impossible error handlers may be performed before or after those with specified error handlers, if the resulting order is otherwise acceptable.

5.16.108 set_stack_limit

Encoding number: 107

lim: EXP POINTER({*locals_alignment, alloca_alignment*})
→ EXP TOP

set_stack_limit sets the limits of remaining free stack space to *lim*. This include both the frame stack limit and the *local_alloc* stack. Note that, in implementations where the frame stack and *local_alloc* stack are distinct, this pointer will have a special representation, appropriate to its frame alignment. Thus the pointer should always be generated using **make_stack_limit** or its equivalent formation.

Any later **apply_general_proc** with PROCPROPS including **check_stack** up to the dynamically next **set_stack_limit** will check that the frame required for the procedure will be within the frame stack limit. If it is not, normal execution is stopped and an XANDF exception with ERROR_CODE stack_overflow is raised.

Any later **local_alloc_check** will check that the locally allocated space required is within the local_alloc stack limit. If it is not, normal execution is stopped and an XANDF exception with ERROR_CODE *stack_overflow* is raised.

5.16.109 shape_offset

Encoding number: 108

s: SHAPE
→ EXP OFFSET(*alignment(s)*,{})

This construction delivers the size of a value of the given SHAPE.

Suppose that a value of SHAPE, *s*, is placed in a space indicated by a POINTER (*x*), *p*, where *x* includes **alignment** (*s*). Suppose that a value of SHAPE, *t*, where *a* is **alignment** (*t*) and *x* includes *a*, is placed at

add_to_ptr(p, offset_pad(a, shape_offset(s)))

then the values shall not overlap. This shall be true for all legal *s*, *x* and *t*.

5.16.110 shift_left

Encoding number: 109

ov_err: ERROR_TREATMENT
arg1: EXP INTEGER(*v*)
arg2: EXP INTEGER(*w*)
→ EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*. The value *a* shifted left *b* places is delivered as the result of the construct, with the same SHAPE as *a*.

b will be non-negative.

If the result cannot be expressed in the VARIETY being used to represent *v*, an overflow error is caused and is handled in the way specified by **ov_err**.

Producers may assume that **shift_left** and multiplication by a power of two yield equally efficient code.

The effect of shifting more places than the number of bits needed to represent *v* is undefined.

5.16.111 shift_right

Encoding number: 110

arg1:	EXP INTEGER(*v*)
arg2:	EXP INTEGER(*w*)
	→ EXP INTEGER(*v*)

arg1 and *arg2* are evaluated and will produce integer values, *a* and *b*. The value *a* shifted right *b* places is delivered as the result of the construct, with the same SHAPE as *arg1*.

b will be non-negative.

If the lower bound of *v* is negative, the sign will be propagated.

The efffect of shifting more places than the number of bits needed to represent *v* is undefined.

5.16.112 subtract_ptrs

Encoding number: 111

arg1:	EXP POINTER(*y*)
arg2:	EXP POINTER(*x*)
	→ EXP OFFSET(*x*,*y*)

arg1 and *arg2* are evaluated to produce pointers *p1* and *p2*, which will be derived from the same original pointer. The result, *r*, is the OFFSET from *p2* to *p1*. Both arguments will be derived from the same original pointer.

Note that *add_to_ptr(p2 ,r) = p1*.

5.16.113 tail_call

Encoding number: 112

prcprops:	OPTION(PROCPROPS)
p:	EXP PROC
callee_params:	CALLEES
	→ EXP BOTTOM

p is called in the sense of **apply_general_proc** with the caller parameters of the immediately enclosing procedure and CALLEES given by *callee_params* and PROCPROPS *prcprops*.

The result of the call is delivered as the result of the immediately enclosing procedure in the sense of **return**. The SHAPE of the result of *p* will be identical to the SHAPE specified as the result of immediately enclosing procedure.

No pointers to any callee parameters, variables, identifications or local allocations defined in immediately enclosing procedure will be accessed either in the body of *p* or after the return.

The presence or absence of each of the PROCPROPS **check_stack** and **untidy**, in *prcprops*, will be reflected in the PROCPROPS of the immediately enclosing procedure.

5.16.114 untidy_return

Encoding number: 113

arg1:	EXP*x*
	→ EXP BOTTOM

arg1 is evaluated to produce a value, *v*. The evaluation of the immediately enclosing procedure ceases and *v* is delivered as the result of the procedure, in such a manner as that pointers to any callee parameters or local allocations are valid in the calling procedure.

untidy_return can only occur in a procedure defined by **make_general_proc** with PROCPROPS including *untidy*.

5.16.115 variable

Encoding number: 114

opt_access:	OPTION(ACCESS)
name_intro:	TAG POINTER(*alignment(x)*)
init:	EXP *x*
body:	EXP *y*
	→ EXP *y*

init is evaluated to produce a value, *v*. Space is allocated to hold a value of SHAPE *x* and this is initialised with *v*. Then *body* is evaluated. During this evaluation, an original POINTER pointing to the allocated space is bound to *name_intro*. This means that inside *body* an evaluation of **obtain_tag** (*name_intro*) will produce a POINTER to this space. The lifetime of *name_intro* is the evaluation of *body*.

The value delivered by **variable** is that produced by *body*.

If *opt_access* contains *visible*, it means that the contents of the space may be altered while the procedure containing this declaration is not the current procedure. Hence if there are any copies of this value they will need to be refreshed from the variable when the procedure is returned to. The easiest implementation when *opt_access* is *visible* may be to keep the value in memory, but this is not a necessary requirement.

The TAG given for *name_intro* will not be reused within the current UNIT. No rules for the hiding of one TAG by another are given: this will not happen.

The order in which the constituents of *init* and *body* are evaluated shall be indistinguishable in all observable effects (apart from time) from completely evaluating *init* before starting *body*. See the note about order in Section 5.16.107 on page 76.

When compiling languages which permit uninitialised variable declarations, **make_value** may be used to provide an initialisation.

5.16.116 xor

Encoding number: 115

arg1:	EXP INTEGER(*v*)
arg2:	EXP INTEGER(*v*)
	→ EXP INTEGER(*v*)

The arguments are evaluated producing integer values of the same VARIETY, *v*. The result is the bitwise logical XOR of these two integers in the representing VARIETY. The result is delivered as the result of the construct, with the same SHAPE as the arguments.

See also Section 7.18 on page 151.

5.17 EXTERNAL

Number of encoding bits: 2

Is coding extendable? Yes

An EXTERNAL defines the classes of external name available for connecting the internal names inside a CAPSULE to the world outside the CAPSULE.

5.17.1 string_extern

Encoding number: 1

byte_align

s: TDFIDENT(*n*)
→ EXTERNAL

string_extern produces an EXTERNAL identified by the TDFIDENT *s*.

5.17.2 unique_extern

Encoding number: 2

byte_align

u: UNIQUE
→ EXTERNAL

unique_extern produces an EXTERNAL identified by the UNIQUE *u*.

5.17.3 chain_extern

Encoding number: 3

byte_align

s: TDFIDENT
prev: TDFINT
→ EXTERNAL

chain_extern produces an EXTERNAL identified by the TDFIDENT *s*.

prev is a linkable entity of the same kind as *s* identified by a **make_link_extern**.

The action of linking a set of CAPSULEs $c_i = 1..n$ containing:

make_link_extern$(e_i$, *chain_extern(S,p))*

make_link_extern$(P_i$, *string_extern*$(p_i))$

is to produce a joint capsule with a:

make_link_extern$(e_1$, *chain_extern*$(S, p_n))$

make_link_extern$(p_n$, *string_extern*$(p_n))$

where P_i is locally linked to $e_i + 1$ for $i = 1..n - 1$, eliminating the other EXTERNALs identifying S and P_i .

For many purposes, the strings P_i would be identical, for example, for chaining together dynamic initialisations in C++ (see Section 7.29 on page 156). Clearly, the intended use of EXTERNALs like these must be commutative and associative with respect to linking order.

5.18 EXTERN_LINK

Number of encoding bits: 0

Is coding extendable? No

This is an auxiliary SORT providing a list of LINKEXTERN.

5.18.1 make_extern_link

Encoding number: 0

el: SLIST(LINKEXTERN)
$\rightarrow$ EXTERN_LINK

make_capsule requires a SLIST (EXTERN_LINK) to express the links between the linkable entities and the named (by EXTERNALS) values outside the CAPSULE.

5.19 FLOATING_VARIETY

Number of encoding bits: 3

Is coding extendable? Yes

These describe kinds of floating point number.

5.19.1 flvar_apply_token

Encoding number: 1

token_value:	TOKEN
token_args:	BITSTREAM*param_sorts(token_value)*
	→ FLOATING_VARIETY

The token is applied to the arguments to give a FLOATING_VARIETY.

If there is a definition for *token_value* in the CAPSULE, then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.19.2 flvar_cond

Encoding number: 2

control:	EXP INTEGER(*v*)
e1:	BITSTREAM FLOATING_VARIETY
e2:	BITSTREAM FLOATING_VARIETY
	→ FLOATING_VARIETY

The *control* is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

5.19.3 flvar_parms

Encoding number: 3

base:	NAT
mantissa_digits:	NAT
minimum_exponent:	NAT
maximum_exponent:	NAT
	→ FLOATING_VARIETY

base is the base with respect to which the remaining numbers refer. *base* will be a power of 2.

mantissa_digits is the required number of base digits, *q*, such that any number with *q* digits can be rounded to a floating point number of the variety and back again without any change to the *q* digits.

minimum_exponent is the negative of the required minimum integer such that *base* raised to that power can be represented as a non-zero floating point number in the FLOATING_VARIETY.

maximum_exponent is the required maximum integer such that *base* raised to that power can be represented in the FLOATING_VARIETY.

An XANDF translator is required to make available a representing FLOATING_VARIETY such that, if only values within the given requirements are produced, no overflow error will occur. Where several such representative FLOATING_VARIETIES exist, the translator will choose one to minimise space requirements or maximise the speed of operations.

All numbers of the form $M^* \, base^{N+1-q}$ are required to be represented exactly, where M and N are integers such that:

$base^{q-1} \leq M < base^q$
$-minimum\_exponent \leq N \leq maximum\_exponent$

Zero will also be represented exactly in any FLOATING_VARIETY.

5.19.4 complex_parms

Encoding number: 4

base: NAT
mantissa_digits: NAT
minimum_exponent: NAT
maximum_exponent: NAT
→ FLOATING_VARIETY

A FLOATING_VARIETY described by **complex_parms** holds a complex number which is likely to be represented by its real and imaginary parts, each of which is as if defined by **flvar_parms** with the same arguments.

5.19.5 float_of_complex

Encoding number: 5

csh: SHAPE
→ FLOATING_VARIETY

csh will be a complex SHAPE.

This delivers the FLOATING_VARIETY required for the real (or imaginary) part of a complex SHAPE *csh*.

5.19.6 complex_of_float

Encoding number: 6

fsh: SHAPE
→ FLOATING_VARIETY

fsh will be a floating SHAPE.

This delivers FLOATING_VARIETY required for a complex number whose real (and imaginary) parts have SHAPE *fsh*.

5.20 GROUP

Number of encoding bits: 0

Is coding extendable? No

A GROUP is a list of UNITS with the same unit identification.

5.20.1 make_group

Encoding number: 0

us: SLIST(UNIT)
→ GROUP

make_capsule contains a list of GROUPS. Each member of this list has a different unit identification deduced from the *prop_name* argument of **make_capsule**.

5.21 LABEL

Number of encoding bits: 1

Is coding extendable? Yes

A LABEL marks an EXP in certain constructions, and is used in jump-like constructions to change the control to the labelled construction.

5.21.1 label_apply_token

Encoding number: 2

token_value: TOKEN
token_args: BITSTREAM*param_sorts(token_value)*
→ LABEL*x*

The token is applied to the arguments to give a LABEL.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.21.2 make_label

Encoding number: 1

labelno: TDFINT
→ LABEL

Labels are represented in XANDF by integers, but they are not linkable. Hence the definition and all uses of a LABEL occur in the same UNIT.

5.22 LINK

Number of encoding bits: 0

Is coding extendable? No

A LINK expresses the connection between two variables of the same SORT.

5.22.1 make_link

Encoding number: 0

unit_name: TDFINT
capsule_name: TDFINT
$\rightarrow$ LINK

A LINK defines a linkable entity declared inside a UNIT as *unit_name*, to correspond to a CAPSULE linkable entity having the same linkable entity identification. The CAPSULE linkable entity is *capsule_name*.

A LINK is normally constructed by the XANDF builder in the course of resolving sharing and name clashes when constructing a composite CAPSULE.

5.23 LINKEXTERN

Number of encoding bits: 0

Is coding extendable? No

A value of SORT LINKEXTERN expresses the connection between the name by which an object is known inside a CAPSULE and a name by which it is known outside.

5.23.1 make_linkextern

Encoding number: 0

internal: TDFINT
ext: EXTERNAL
$\rightarrow$ LINKEXTERN

make_linkextern produces a LINKEXTERN connecting an object identified within a CAPSULE by a TAG, TOKEN, AL_TAG or any linkable entity constructed from *internal*, with an EXTERNAL, *ext*. The EXTERNAL is an identifier which linkers and similar programs can use.

5.24 LINKS

Number of encoding bits: 0

Is coding extendable? No

5.24.1 make_links

Encoding number: 0

ls: SLIST(LINK)
→ LINKS

make_unit uses a SLIST (LINKS) to define which linkable entities within a UNIT correspond to the CAPSULE linkable entities. Each LINK in a LINKS has the same linkable entity identification.

5.25 NAT

Number of encoding bits: 3

Is coding extendable? Yes

These are non-negative integers of unlimited size.

5.25.1 nat_apply_token

Encoding number: 1

token_value:	TOKEN
token_args:	BITSTREAM*param_sorts(token_value)*
	→ NAT

The token is applied to the arguments to give a NAT.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.25.2 nat_cond

Encoding number: 2

control:	EXP INTEGER(*v*)
e1:	BITSTREAM NAT
e2:	BITSTREAM NAT
	→ NAT

The *control* is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

5.25.3 computed_nat

Encoding number: 3

arg:	EXP INTEGER(*v*)
	→ NAT

arg will be an install-time non-negative constant. The result is that constant.

5.25.4 error_val

Encoding number: 4

err:	ERROR_CODE
	→ NAT

Gives the NAT corresponding to the ERROR_CODE *err*. Each distinct ERROR_CODE will give a different NAT.

5.25.5 make_nat

Encoding number: 5

n: TDFINT
→ NAT

n is a non-negative integer of unbounded magnitude.

5.26 NTEST

Number of encoding bits: 4

Is coding extendable? Yes

These describe the comparisons which are possible in the various test constructions. Note that **greater_than** is not necessarily the same as **not_less_than_or_equal,** since the result need not be defined (for example, in IEEE floating point).

5.26.1 ntest_apply_token

Encoding number: 1

token_value:	TOKEN
token_args:	BITSTREAM*param_sorts(token_value)*
	→ NTEST

The token is applied to the arguments to give a NTEST.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.26.2 ntest_cond

Encoding number: 2

control:	EXP INTEGER(*v*)
e1:	BITSTREAM NTEST
e2:	BITSTREAM NTEST
	→ NTEST

The *control* is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

5.26.3 equal

Encoding number: 3

→ NTEST

Signifies *equal* test.

5.26.4 greater_than

Encoding number: 4

→ NTEST

Signifies *greater than* test.

5.26.5 greater_than_or_equal

Encoding number: 5

→ NTEST

Signifies *greater than or equal* test.

5.26.6 less_than

Encoding number: 6

→ NTEST

Signifies *less than* test.

5.26.7 less_than_or_equal

Encoding number: 7

→ NTEST

Signifies *less than or equal* test.

5.26.8 not_equal

Encoding number: 8

→ NTEST

Signifies *not equal* test.

5.26.9 not_greater_than

Encoding number: 9

→ NTEST

Signifies *not greater than* test.

5.26.10 not_greater_than_or_equal

Encoding number: 10

→ NTEST

Signifies *not (greater than or equal)* test.

5.26.11 not_less_than

Encoding number: 11

→ NTEST

Signifies *not less than* test.

5.26.12 not_less_than_or_equal

Encoding number: 12

→ NTEST

Signifies *not (less than or equal)* test.

5.26.13 less_than_or_greater_than

Encoding number: 13

→ NTEST

Signifies *less than or greater than* test.

5.26.14 not_less_than_and_not_greater_than

Encoding number: 14

→ NTEST

Signifies *not less than and not greater than* test.

5.26.15 comparable

Encoding number: 15

→ NTEST

Signifies *comparable* test.

With all operands SHAPES except FLOATING, this comparison is always true.

5.26.16 not_comparable

Encoding number: 16

→ NTEST

Signifies *not comparable* test.

With all operands SHAPES except FLOATING, this comparison is always false.

5.27 OTAGEXP

Number of encoding bits: 0

Is coding extendable? No

This is an auxiliary SORT used in **apply_general_proc**.

5.27.1 make_otagexp

Encoding number: 0

tgopt: OPTION(TAG*x*)
e: EXP*x*
→ OTAGEXP

e is evaluated and its value is the actual caller parameter. If *tgopt* is present, the TAG will be bound to the final value of caller parameter in the *postlude* part of the **apply_general_proc**.

5.28 PROCPROPS

Number of encoding bits: 4

Is coding extendable? Yes

PROCPROPS is a set of properties ascribed to procedure definitions and calls.

5.28.1 procprops_apply_token

Encoding number: 1

token_value:	TOKEN
token_args:	BITSTREAM*param_sorts(token_value)*
	→ PROCPROPS

The token is applied to the arguments to give a PROCPROPS.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters in the order specified.

5.28.2 procprops_cond

Encoding number: 2

control:	EXP INTEGER(v)
e1:	BITSTREAM PROCPROPS
e2:	BITSTREAM PROCPROPS
	→ PROCPROPS

The *control* is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

5.28.3 add_procprops

Encoding number: 3

arg1:	PROCPROPS
arg2:	PROCPROPS
	→ PROCPROPS

This delivers the join of *arg1* and *arg2*.

5.28.4 check_stack

Encoding number: 4

→ PROCPROPS

The procedure *body* is required to check for stack overflow.

5.28.5 inline

Encoding number: 5

$\rightarrow$ PROCPROPS

The procedure *body* is a good candidate for inlining at its application.

5.28.6 no_long_jump_dest

Encoding number: 6

$\rightarrow$ PROCPROPS

The procedure *body* will contain no label which is the destination of a **long_jump**.

5.28.7 untidy

Encoding number: 7

$\rightarrow$ PROCPROPS

The procedure *body* may be exited using an **untidy_return**.

5.28.8 var_callees

Encoding number: 8

$\rightarrow$ PROCPROPS

Applications of the procedure may have different numbers of actual callee parameters.

5.28.9 var_callers

Encoding number: 9

$\rightarrow$ PROCPROPS

Applications of the procedure may have different numbers of actual caller parameters.

5.29 PROPS

A PROPS is an assemblage of program information. This standard offers various ways of constructing a PROPS — that is, it defines kinds of information which it is useful to express. These are:

- definitions of AL_TAGs standing for ALIGNMENTs
- declarations of TAGs standing for EXPs
- definitions of the EXPs for which TAGs stand
- declarations of TOKENs standing for pieces of program
- definitions of the pieces of XANDF program for which TOKENs stand
- linkage and naming information
- version information.

PROPS giving diagnostic information form part of an optional extension to XANDF which is described in Section 9.4.2 on page 173.

The standard can be extended by the definition of new kinds of PROPS information and new PROPS constructs for expressing them. Also, private standards can define new kinds of information and corresponding constructs without disruption to adherents to the present standard.

Each GROUP of UNITS is identified by a unit identification — a TDFIDENT. All the UNITS in that GROUP are of the same kind.

In addition there is a *tld* UNIT (see Section 8.3 on page 161).

5.30 ROUNDING_MODE

Number of encoding bits: 3

Is coding extendable? Yes

ROUNDING_MODE specifies the way rounding is to be performed in floating point arithmetic.

5.30.1 rounding_mode_apply_token

Encoding number: 1

token_value: TOKEN
token_args: BITSTREAM*param_sorts(token_value)*
→ ROUNDINGMODE

The token is applied to the arguments to give a ROUNDING_MODE.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.30.2 rounding_mode_cond

Encoding number: 2

control: EXP INTEGER(*v*)
e1: BITSTREAM ROUNDING_MODE
e2: BITSTREAM ROUNDING_MODE
→ ROUNDINGMODE

The *control* is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

5.30.3 round_as_state

Encoding number: 3

→ ROUNDINGMODE

Round as specified by the current state of the machine.

5.30.4 to_nearest

Encoding number: 4

→ ROUNDINGMODE

Signifies rounding to nearest. The effect when the number lies half-way is not specified.

5.30.5 toward_larger

Encoding number: 5

→ ROUNDINGMODE

Signifies rounding toward next largest.

5.30.6 toward_smaller

Encoding number: 6

→ ROUNDINGMODE

Signifies rounding toward next smallest.

5.30.7 toward_zero

Encoding number: 7

→ ROUNDINGMODE

Signifies rounding toward zero.

5.31 SHAPE

Number of encoding bits: 4

Is coding extendable? Yes

SHAPEs express symbolic size and representation information about run time values.

SHAPEs are constructed from primitive SHAPEs which describe values such as procedures and integers, and recursively from compound construction in terms of other SHAPEs.

5.31.1 shape_apply_token

Encoding number: 1

token_value:	TOKEN
token_args:	BITSTREAM*param_sorts(token_value)*
	→ SHAPE

The token is applied to the arguments to give a SHAPE.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.31.2 shape_cond

Encoding number: 2

control:	EXP INTEGER(*v*)
e1:	BITSTREAM SHAPE
e2:	BITSTREAM SHAPE
	→ SHAPE

The *control* is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

5.31.3 bitfield

Encoding number: 3

bf_var:	BITFIELD_VARIETY
	→ SHAPE

A BITFIELD is used to represent a pattern of bits which will be packed, provided that the *variety_enclosed* constraints are not violated (see Section 7.24 on page 154).

A BITFIELD_VARIETY specifies the number of bits and whether they are considered to be signed.

There are very few operations on BITFIELDs, which have to be converted to INTEGERs before arithmetic can be performed on them.

An installer may place a limit on the number of bits it implements (see Section 7.25 on page 155).

5.31.4 bottom

Encoding number: 4

→ SHAPE

BOTTOM is the SHAPE which describes a piece of program which does not evaluate to any result. Examples include **goto** and **return.**

If BOTTOM is a parameter to any other SHAPE constructor, the result is BOTTOM.

5.31.5 compound

Encoding number: 5

sz: EXP OFFSET(*x*,*y*)
→ SHAPE

The SHAPE constructor COMPOUND describes cartesian products and unions.

The alignments *x* and *y* will be **alignment** (*sx*) and **alignment** (*sy*) for some SHAPEs *sx* and *sy*.

sz will evaluate to a constant, non-negative OFFSET (see Section 5.16.87 on page 69). The resulting SHAPE describes a value whose size is given by *sz*.

5.31.6 floating

Encoding number: 6

fv: FLOATING_VARIETY
→ SHAPE

Most of the floating point arithmetic operations, **floating_plus**, **floating_minus**, and so on, are defined to work in the same way on different kinds of floating point number. If these operations have more than one argument, the arguments have to be of the same kind, and the result is of the same kind.

See also Section 7.20 on page 152.

An installer may limit the FLOATING_VARIETIES it can represent. A statement of any such limits shall be part of the specification of an installer. See Section 7.20 on page 152.

5.31.7 integer

Encoding number: 7

var: VARIETY
→ SHAPE

The different kinds of INTEGER are distinguished by having different VARIETYs. A fundamental VARIETY (not a TOKEN or conditional) is represented by two SIGNED_NATs, which are respectively the lower and upper bounds (inclusive) of the set of values belonging to the VARIETY.

Most architectures require that dyadic integer arithmetic operations take arguments of the same size, and so XANDF does likewise. Because XANDF is completely architecture neutral and makes no assumptions about word length, this means that the VARIETYs of the two arguments must be identical. An example illustrates this. A piece of XANDF which attempted to add two values whose SHAPEs were:

INTEGER(0, 60000)

and:

INTEGER(0, 30000)

would be undefined. The reason is that without knowledge of the target architecture's word length, it is impossible to guarantee that the two values are going to be represented in the same number of bytes. On a 16-bit machine they probably would, but not on a 15-bit machine. The only way to ensure that two INTEGERs are going to be represented in the same way in all machines is to stipulate that their VARIETYs are exactly the same.

When any construct delivering an INTEGER of a given VARIETY produces a result which is not representable in the space which an installer has chosen to represent that VARIETY, an integer overflow occurs. Whether it occurs in a particular case depends on the target, because the installers' decisions on representation are inherently target-defined.

A particular installer may limit the ranges of integers that it implements. See Section 7.18 on page 151.

5.31.8 nof

Encoding number: 8

n: NAT
s: SHAPE
→ SHAPE

The NOF constructor describes the SHAPE of a value consisting of an array of *n* values of the same SHAPE, *s*.

5.31.9 offset

Encoding number: 9

arg1: ALIGNMENT
arg2: ALIGNMENT
→ SHAPE

The SHAPE constructor OFFSET describes values which represent the differences between POINTERs, that is they measure offsets in memory. It should be emphasised that these are in general run-time values.

An OFFSET measures the displacement from the value indicated by a POINTER (*arg1*) to the value indicated by a POINTER (*arg2*). Such an offset is only defined if the POINTERs are derived from the same original POINTER.

An OFFSET may also measure the displacement from a POINTER to the start of a BITFIELD_VARIETY, or from the start of one BITFIELD_VARIETY to the start of another. Hence, unlike the argument of pointer , *arg1* or *arg2* may consist entirely of BITFIELD_VARIETYs.

The set *arg1* will include the set *arg2*.

See also Section 7.13 on page 147.

5.31.10 pointer

Encoding number: 10

arg: ALIGNMENT
$\rightarrow$ SHAPE

A POINTER is a value which points to space allocated in a computer's memory. The POINTER constructor takes an ALIGNMENT argument. This argument will not consist entirely of BITFIELD_VARIETYs. See Section 7.13 on page 147.

5.31.11 proc

Encoding number: 11

$\rightarrow$ SHAPE

PROC is the SHAPE which describes pieces of program.

5.31.12 top

Encoding number: 12

$\rightarrow$ SHAPE

TOP is the SHAPE which describes pieces of program which return no useful value. **assign** is an example: it performs an assignment, but does not deliver any useful value.

5.32 SIGNED_NAT

Number of encoding bits: 3

Is coding extendable? Yes

These are positive or negative integers of unbounded size.

5.32.1 signed_nat_apply_token

Encoding number: 1

token_value: TOKEN
token_args: BITSTREAM*param_sorts(token_value)*
→ SIGNED_NAT

The token is applied to the arguments to give a SIGNED_NAT.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.32.2 signed_nat_cond

Encoding number: 2

control: EXP INTEGER(*v*)
e1: BITSTREAM SIGNED_NAT
e2: BITSTREAM SIGNED_NAT
→ SIGNED_NAT

The *control* is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

5.32.3 computed_signed_nat

Encoding number: 3

arg: EXP INTEGER(*v*)
→ SIGNED_NAT

arg will be an install-time constant. The result is that constant.

5.32.4 make_signed_nat

Encoding number: 4

neg: TDFBOOL
n: TDFINT
→ SIGNED_NAT

n is a non-negative integer of unbounded magnitude. The result is negative if *neg* is true.

5.32.5 snat_from_nat

Encoding number: 5

neg: BOOL
n: NAT
→ SIGNED_NAT

The result is negated if *neg* is true.

5.33 SORTNAME

Encoding number: 5

Is coding extendable? Yes

These are the names of the SORTs which can be parameters of TOKEN definitions.

5.33.1 access

Encoding number: 1

→ SORTNAME

5.33.2 al_tag

Encoding number: 2

→ SORTNAME

5.33.3 alignment_sort

Encoding number: 3

→ SORTNAME

5.33.4 bitfield_variety

Encoding number: 4

→ SORTNAME

5.33.5 bool

Encoding number: 5

→ SORTNAME

5.33.6 error_treatment

Encoding number: 6

→ SORTNAME

5.33.7 exp

Encoding number: 7

→ SORTNAME

The SORT of EXP.

5.33.8 floating_variety

Encoding number: 8

→ SORTNAME

5.33.9 foreign_sort

Encoding number: 9

foreign_name: STRING(*k*,*n*)
→ SORTNAME

This SORT enables unanticipated kinds of information to be placed in XANDF.

5.33.10 label

Encoding number: 10

→ SORTNAME

5.33.11 nat

Encoding number: 11

→ SORTNAME

5.33.12 ntest

Encoding number: 12

→ SORTNAME

5.33.13 procprops

Encoding number: 13

→ SORTNAME

5.33.14 rounding_mode

Encoding number: 14

$\rightarrow$ SORTNAME

5.33.15 shape

Encoding number: 15

$\rightarrow$ SORTNAME

5.33.16 signed_nat

Encoding number: 16

$\rightarrow$ SORTNAME

5.33.17 string

Encoding number: 17

$\rightarrow$ SORTNAME

5.33.18 tag

Encoding number: 18

$\rightarrow$ SORTNAME

The SORT of TAG.

5.33.19 transfer_mode

Encoding number: 19

$\rightarrow$ SORTNAME

5.33.20 token

Encoding number: 20

result: SORTNAME
params: LIST(SORTNAME)
$\rightarrow$ SORTNAME

This is the SORTNAME of a TOKEN. Note that it can have tokens as parameters, but not as result.

5.33.21 variety

Encoding number: 21

$\rightarrow$ SORTNAME

5.34 STRING

Number of encoding bits: 3

Is coding extendable? Yes

5.34.1 string_apply_token

Encoding number: 1

token_value: TOKEN
token_args: BITSTREAM*param_sorts(token_value)*
→ STRING(*k*,*n*)

The token is applied to the arguments to give a STRING.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.34.2 string_cond

Encoding number: 2

control: EXP INTEGER(*v*)
e1: BITSTREAM STRING
e2: BITSTREAM STRING
→ STRING(*k*,*n*)

The *control* is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

5.34.3 concat_string

Encoding number: 3

arg1: STRING(*k*,*n*)
arg2: STRING(*k*,*m*)
→ STRING(*k*,*n*+*m*)

This gives a STRING which is the concatenation of *arg1* with *arg2*.

5.34.4 make_string

Encoding number: 4

arg: TDFSTRING(*k*,*n*)
→ STRING(*k*,*n*)

This delivers the STRING identical to the *arg*.

5.35 TAG

Number of encoding bits: 1

Is coding extendable? Yes

Linkable entity identification: tag

These are used to name values and variables in the run time program.

5.35.1 tag_apply_token

Encoding number: 2

token_value:	TOKEN
token_args:	BITSTREAM*param_sorts(token_value)*
	→ TAG*x*

The token is applied to the arguments to give a TAG.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.35.2 make_tag

Encoding number: 1

tagno:	TDFINT
	→ TAG*x*

make_tag produces a TAG identified by *tagno*.

5.36 TAGACC

Number of encoding bits: 0

Is coding extendable? No

This constructs a pair of a TAG and an OPTION(ACCESS) for use in **make_proc**.

5.36.1 make_tagacc

Encoding number: 0

tg:	TAG POINTER*var_param_alignment*
acc:	OPTION(ACCESS)
	→ TAGACC

This constructs the pair for **make_proc**.

5.37 TAGDEC

Number of encoding bits: 2

Is coding extendable? Yes

A TAGDEC declares a TAG for incorporation into a TAGDEC_PROPS.

5.37.1 make_id_tagdec

Encoding number: 1

t_intro:	TDFINT
acc:	OPTION(ACCESS)
signature:	OPTION(STRING)
x:	SHAPE
	→ TAGDEC

A TAGDEC announcing that the TAG *t_intro* identifies an EXP of SHAPE *x* is constructed.

acc specifies the ACCESS properties of the TAG.

If there is a **make_id_tagdec** for a TAG then all other **make_id_tagdec** for the same TAG will specify the same SHAPE and there will be no **make_var_tagdec** or **common_tagdec** for the TAG.

If two **make_id_tagdecs** specify the same tag and both have signatures present, the strings will be identical. Possible uses of this signature argument are outlined in Section 7.28 on page 156.

5.37.2 make_var_tagdec

Encoding number: 2

t_intro:	TDFINT
acc:	OPTION(ACCESS)
signature:	OPTION(STRING)
x:	SHAPE
	→ TAGDEC

A TAGDEC announcing that the TAG *t_intro* identifies an EXP of SHAPE POINTER (**alignment**(*x*)) is constructed.

acc specifies the ACCESS properties of the TAG.

If there is a **make_var_tagdec** for a TAG then all other **make_var_tagdec** for the same TAG will specify the same SHAPE and there will be no **make_id_tagdec** or **common_tagdec** for the TAG.

If two **make_var_tagdecs** specify the same tag and both have signatures present, the strings will be identical. Possible uses of this signature argument are outlined in Section 7.28 on page 156.

5.37.3 common_tagdec

Encoding number: 3

t_intro:	TDFINT
acc:	OPTION(ACCESS)
signature:	OPTION(STRING)
x:	SHAPE
	→ TAGDEC

A TAGDEC announcing that the TAG *t_intro* identifies an EXP of SHAPE POINTER *(alignment(x))* is constructed.

acc specifies the ACCESS properties of the TAG.

If there is a **common_tagdec** for a TAG then there will be no **make_id_tagdec** or **make_var_tagdec** for that TAG. If there is more than one **common_tagdec** for a TAG, the one having the maximum SHAPE shall be taken to apply for the CAPSULE. Each pair of such SHAPES will have a maximum. The maximum of two SHAPES, *a* and *b,* is defined as follows:

1. If the *a* is equal to *b*, the maximum is *a*.
2. If *a* and *b* are COMPOUND (*x*) and COMPOUND (*y*) respectively and *a* is an initial segment of *b,* then *b* is the maximum. Similarly if *b* is an initial segment of *a*, then *a* is the maximum.
3. If *a* and *b* are NOF (*n*, *x*) and NOF (*m*, *x*) respectively and *n* is less than or equal to *m,* then *b* is the maximum. Similarly if *m* is less than or equal to *n*, then *a* is the maximum.
4. Otherwise *a* and *b* have no maximum.

If two **common_tagdecs** specify the same tag and both have signatures present, the strings will be identical. Possible uses of this signature argument are outlined in Section 7.28 on page 156.

5.38 TAGDEC_PROPS

Number of encoding bits: 0

Is coding extendable? No

Unit identification: tagdec

5.38.1 make_tagdecs

Encoding number: 0

no_labels:	TDFINT
tds:	SLIST(TAGDEC)
	→ TAGDEC_PROPS

no_labels is the number of local LABELs used in *tds*. *tds* is a list of TAGDECs which declare the SHAPEs associated with TAGs.

5.39 TAGDEF

Number of encoding bits: 2

Is coding extendable? Yes

A value of SORT TAGDEF gives the definition of a TAG for incorporation into a TAGDEF_PROPS.

5.39.1 make_id_tagdef

Encoding number: 1

t: TDFINT
signature: OPTION(STRING)
e: EXP*x*
→ TAGDEF

make_id_tagdef produces a TAGDEF defining the TAG *x* constructed from the TDFINT, *t*. This TAG is defined to stand for the value delivered by *e*.

e will be a constant which can be evaluated at load-time or *e* will be some *initial_value* (E) (see Section 5.16.48 on page 56).

t will be declared in the CAPSULE using **make_id_tagdec**. If both the **make_id_tagdec** and **make_id_tagdef** have signatures present, the strings will be identical.

If *x* is PROC and the TAG represented by *t* is named externally via a CAPSULE_LINK, e will be some **make_proc** or **make_general_proc**.

There will not be more than one TAGDEF defining *t* in a CAPSULE.

5.39.2 make_var_tagdef

Encoding number: 2

t: TDFINT
opt_access: OPTION(ACCESS)
signature: OPTION(STRING)
e: EXP*x*
→ TAGDEF

make_var_tagdef produces a TAGDEF defining the TAG POINTER (*x*) constructed from the TDFINT, *t*. This TAG stands for a variable which is initialised with the value delivered by *e*. The TAG is bound to an original pointer which has the evaluation of the program as its lifetime.

If *opt_access* contains *visible*, the meaning is that the variable may be used by agents external to the capsule, and so it must not be optimised away. If it contains *constant*, the initialising value will remain in it throughout the program.

e will be a constant which can be evaluated at load-time or *e* will be some *initial_value* (E) (see Section 5.16.48 on page 56). *t* will be declared in the CAPSULE using **make_var_tagdec**. If both the **make_var_tagdec** and **make_var_tagdef** have signatures present, the strings will be identical.

There will not be more than one TAGDEF defining *t* in a CAPSULE.

5.39.3 common_tagdef

Encoding number: 3

t:	TDFINT
opt_access:	OPTION(ACCESS)
signature:	OPTION(STRING)
e:	EXP*x*
	→ TAGDEF

common_tagdef produces a TAGDEF defining the TAG POINTER (*x*) constructed from the TDFINT, *t*. This TAG stands for a variable which is initialised with the value delivered by *e*. The TAG is bound to an original pointer which has the evaluation of the program as its lifetime.

If *opt_access* contains *visible*, the meaning is that the variable may be used by agents external to the capsule, and so it must not be optimised away. If it contains *constant*, the initialising value will remain in it throughout the program.

e will be a constant evaluable at load-time or *e* will be some *initial_value* (E) (see Section 5.16.48 on page 56).

t will be declared in the CAPSULE using **common_tagdec**. If both the **common_tagdec** and **common_tagdef** have signatures present, the strings will be identical. Let the maximum SHAPE of these (see Section 5.37.3 on page 116) be *s*.

There may be any number of **common_tagdef** definitions for *t* in a CAPSULE. Of the *e* parameters of these, one will be a maximum. This maximum definition is chosen as the definition of *t*. Its value of *e* will have SHAPE *s*.

The maximum of two **common_tagdef** EXPs, *a* and *b*, is defined as follows:

1. If *a* has the form *make_value* (*s*), *b* is the maximum.
2. If *b* has the form *make_value* (*s*), *a* is the maximum.
3. If *a* and *b* have SHAPE COMPOUND (*x*) and COMPOUND (*y*) respectively and the value produced by *a* is an initial segment of the value produced by *b*, then *b* is the maximum. Similarly if *b* is an initial segment of *a*, then *a* is the maximum.
4. If *a* and *b* have SHAPE NOF (*n*, *x*) and NOF (*m*, *x*) respectively and the value produced by *a* is an initial segment of the value produced by *b*, then *b* is the maximum. Similarly if *b* is an initial segment of *a*, then *a* is the maximum.
5. If the value produced by *a* is equal to the value produced by *b*, the maximum is *a*.
6. Otherwise *a* and *b* have no maximum.

5.40 TAGDEF_PROPS

Number of encoding bits: 0

Is coding extendable? No

Unit identification: tagdef

5.40.1 make_tagdefs

Encoding number: 0

no_labels:	TDFINT
tds:	SLIST(TAGDEF)
	→ TAGDEF_PROPS

no_labels is the number of local LABELs used in *tds*. *tds* is a list of TAGDEFs which give the EXPs which are the definitions of values associated with TAGs.

5.41 TAGSHACC

Number of encoding bits: 0

Is coding extendable? No

5.41.1 make_tagshacc

Encoding number: 0

sha:	SHAPE
opt_access:	OPTION(ACCESS)
tg_intro:	TAG
	→ TAGSHACC

This is an auxiliary construction to make the elements of *params_intro* in **make_proc**.

5.42 TDFBOOL

A TDFBOOL is the XANDF encoding of a boolean.

5.43 TDFIDENT

A TDFIDENT (*k*, *n*) encodes a sequence of *n* unsigned integers of size *k* bits. *k* will be a multiple of 8.

This construction will not be used inside a BIT STREAM.

5.44 TDFINT

A TDFINT is the XANDF encoding of an unbounded unsigned integer constant.

5.45 TDFSTRING

A TDFSTRING (*k*, *n*) encodes a sequence of *n* unsigned integers of size *k* bits.

5.46 TOKDEC

Number of encoding bits: 1

Is coding extendable? Yes

A TOKDEC declares a TOKEN for incorporation into a UNIT.

5.46.1 make_tokdec

Encoding number: 1

tok:	TDFINT
signature:	OPTION(STRING)
s:	SORTNAME
	→ TOKDEC

The sort of the token *tok* is declared to be *s*. Note that *s* will always be a token SORT, with a list of parameter SORTS (possible empty) and a result SORT.

If two *make_tokdecs* specify the same token and both have signatures present, the strings will be identical. Possible uses of this signature argument are outlined in Section 7.28 on page 156.

5.47 TOKDEC_PROPS

Number of encoding bits: 0

Is coding extendable? No

Unit identification: tokdec

5.47.1 make_tokdecs

Encoding number: 0

tds:	SLIST(TOKDEC)
	→ TOKDEC_PROPS

tds is a list of TOKDECs which gives the sorts associated with TOKENs.

5.48 TOKDEF

Number of encoding bits: 1

Is coding extendable? Yes

A TOKDEF gives the definition of a TOKEN for incorporation into a TOKDEF_PROPS.

5.48.1 make_tokdef

Encoding number: 1

tok:	TDFINT
signature:	OPTION(STRING)
def:	BITSTREAM TOKEN_DEFN
	→ TOKDEF

A TOKDEF is constructed which defines the TOKEN *tok* to stand for the fragment of XANDF, *body*, which may be of any SORT with a SORTNAME, except for *token*. The SORT of the result, *result_sort*, is given by the first component of the BITSTREAM. See also Section 5.51.1 on page 125.

tok may have been introduced by a **make_tokdec.** If both the **make_tokdec** and **make_tokdef** have signatures present, the strings will be identical.

At the application of this TOKEN, actual pieces of XANDF having SORT *sn* [*i*] are supplied to correspond to the *tk* [*i*]. The application denotes the piece of XANDF obtained by substituting these actual parameters for the corresponding TOKENs within *body*.

5.49 TOKDEF_PROPS

Number of encoding bits: 0

Is coding extendable? No

Unit identification: tokdef

5.49.1 make_tokdefs

Encoding number: 0

no_labels:	TDFINT
tds:	SLIST(TOKDEF)
	→ TOKDEF_PROPS

no_labels is the number of local LABELs used in *tds*. *tds* is a list of TOKDEFs which gives the definitions associated with TOKENs.

5.50 TOKEN

Number of encoding bits: 2

Is coding extendable? Yes

Unit identification: token

These are used to stand for functions evaluated at installation time. They are represented by TDFINTs.

5.50.1 token_apply_token

Encoding number: 1

token_value: TOKEN
token_args: BITSTREAM*param_sorts(token_value)*
→ TOKEN

The token is applied to the arguments to give a TOKEN.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.50.2 make_tok

Encoding number: 2

tokno: TDFINT
→ TOKEN

make_tok constructs a TOKEN identified by *tokno*.

5.50.3 use_tokdef

Encoding number: 3

tdef: BITSTREAM TOKEN_DEFN
→ TOKEN

tdef is used to supply the definition, as in **make_tokdef**. Note that TOKENS are only used in **x_apply_token** constructions.

5.51 TOKEN_DEFN

Encoding number: 1

Is linking extendable? Yes

This is an auxiliary SORT used in **make_tokdef** and **use_tokdef**.

5.51.1 token_definition

Encoding number: 1

result_sort: SORTNAME
tok_params: LIST(TOKFORMALS)
body: *result_sort*
$\rightarrow$ TOKEN_DEFN

This makes a token definition. *result_sort* is the SORT of *body*. *tok_params* is a list of formal TOKENS and their SORTS. *body* is the definition, which can use the formal TOKENS defined in *tok_params*.

The effect of applying the definition of a TOKEN is as if the following sequence was obeyed:

1. First, the actual parameters (if any) are expanded to produce expressions of the appropriate SORTS. During this expansion, all token applications in the actual parameters are expanded.

2. Second, the definition is copied, making fresh TAGS and LABELS where these are introduced in **identify**, **variable**, **labelled**, **conditional**, **make_proc, make_general_proc** and **repeat** constructions. Any other TAGS or LABELS used in *body* will be provided by the context (see below) of the TOKEN_DEFN or by the expansions of the actual parameters.

3. Third, the actual parameter expressions are substituted for the formal parameter tokens in *tok_params* to give the final result.

The context of a TOKEN_DEFN is the set of names (TOKENS, TAGS, LABELS, AL_TAGS, and so on) in scope at the site of the TOKEN_DEFN.

Thus, in a **make_tokdef**, the context consists of the set of TOKENS defined in its tokdef UNIT, together with the set of linkable entities defined by the **make_links** of that UNIT. Note that this does not include LABELS and the only TAGS included are ‘‘global’’ ones.

In a **use_tokdef**, the context may be wider, since the site of the TOKEN_DEFN need not be in a tokdef UNIT; it may be an actual parameter of a token application. If this happens to be within an EXP, there may be TAGS or LABELS locally within scope; these will be in the context of the TOKEN_DEFN, together with the global names of the enclosing UNIT as before.

Early drafts of this specification limited token definitions to be non-recursive. There is no intrinsic reason for the limitation on recursive TOKENs. Since the UNIT structure implies different namespaces, there is very little implementation advantage to be gained from retaining the limitation.

5.52 TOKFORMALS

Number of encoding bits: 0

Is coding extendable? No

5.52.1 make_tokformals

Encoding number: 0

sn: SORTNAME
tk: TDFINT
→ TOKFORMALS

This is an auxiliary construction to make up the elements of the lists in **token_definition**.

5.53 TRANSFER_MODE

Number of encoding bits: 3

Is coding extendable? Yes

A TRANSFER_MODE controls the operation of **assign_with_mode**, **contents_with_mode** and **move_some**.

A TRANSFER_MODE acts like a set of the values *overlap*, *trap_on_nil*, *complete* and *volatile*. The TRANSFER_MODE **standard_transfer_mode** acts like the empty set. **add_modes** acts like set union.

5.53.1 transfer_mode_apply_token

Encoding number: 1

token_value: TOKEN
token_args: BITSTREAM*param_sorts(token_value)*
→ TRANSFER_MODE

The token is applied to the arguments encoded in the BITSTREAM *token_args* to give a TRANSFER_MODE.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.53.2 transfer_mode_cond

Encoding number: 2

control: EXP INTEGER(*v*)
e1: BITSTREAM TRANSFER_MODE
e2: BITSTREAM TRANSFER_MODE
→ TRANSFER_MODE

control is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

5.53.3 add_modes

Encoding number: 3

md1: TRANSFER_MODE
md2: TRANSFER_MODE
→ TRANSFER_MODE

A construction qualified by **add_modes** has both TRANSFER_MODES *md1* and *md2*. If *md1* is **standard_transfer_mode** then the result is *md2* and symmetrically. This operation is associative and commutative.

5.53.4 overlap

Encoding number: 4

→ TRANSFER_MODE

If **overlap** is used to qualify a **move_some** or an **assign_with_mode** for which *arg2* is a **contents** or **contents_with_mode,** then the source and destination might overlap. The transfer shall be made as if the data were copied from the source to an independent place and thence to the destination.

See also Section 7.16 on page 151.

5.53.5 standard_transfer_mode

Encoding number: 5

→ TRANSFER_MODE

This TRANSFER_MODE implies no special properties.

5.53.6 trap_on_nil

Encoding number: 6

→ TRANSFER_MODE

If **trap_on_nil** is used to qualify a **contents_with_mode** operation with a nil pointer argument, or an **assign_with_mode** whose *arg1* is a nil pointer, or a **move_some** with either argument a nil pointer, the XANDF exception *nil_access* is raised.

5.53.7 volatile

Encoding number: 7

→ TRANSFER_MODE

If **volatile** is used to qualify a construction, it shall not be optimised away.

This is intended to implement ANSI C's **volatile** construction. In this use, any **volatile** identifier should be declared as a TAG with *used_as_volatile* ACCESS.

5.53.8 complete

Encoding number: 8

→ TRANSFER_MODE

A transfer qualified with **complete** shall leave the destination unchanged if the evaluation of the value transferred is left with a **jump**.

5.54 UNIQUE

Encoding number: 0

Is coding extendable? No

These are used to provide world-wide unique names for TOKENs and TAGs.

This implies a registry for allocating UNIQUE values.

5.54.1 make_unique

Encoding number: 0

text: SLIST(TDFIDENT)
$\rightarrow$ UNIQUE

Two UNIQUE values are equal if they were constructed with equal arguments.

5.55 UNIT

Number of encoding bits: 0

Is coding extendable? No

A UNIT gathers together a PROPs and LINKs which relate the names by which objects are known inside the PROPs and names by which they are to be known across the whole of the enclosing CAPSULE.

5.55.1 make_unit

Encoding number: 0

local_vars:	SLIST(TDFINT)
lks:	SLIST(LINKS)
properties:	BYTESTREAM PROPS
	→ UNIT

local_vars gives the number of linkable entities of each kind. These numbers correspond (in the same order) to the variable sorts in **capsule_linking** in **make_capsule.** The linkable entities will be represented by TDFINTs in the range 0 to the corresponding $nl - 1$.

lks gives the LINKs for each kind of entity in the same order as in *local_vars*.

The properties will be a PROPS of a form dictated by the unit identification (see Section 5.11.1 on page 33).

The length of *lks* will be either 0 or equal to the length of *capsule_linking* in **make_capsule**.

5.56 VARIETY

Number of encoding bits: 2

Is coding extendable? Yes

These describe the different kinds of integer which can occur at run time. The fundamental construction consists of a SIGNED_NAT for the lower bound of the range of possible values, and a SIGNED_NAT for the upper bound (inclusive at both ends).

There is no limitation on the magnitude of these bounds in XANDF, but an installer may specify limits. See Section 7.18 on page 151.

5.56.1 var_apply_token

Encoding number: 1

token_value:	TOKEN
token_args:	BITSTREAM*param_sorts(token_value)*
	→ VARIETY

The token is applied to the arguments to give a VARIETY.

If there is a definition for *token_value* in the CAPSULE then *token_args* is a BITSTREAM encoding of the SORTs of its parameters, in the order specified.

5.56.2 var_cond

Encoding number: 2

control:	EXP INTEGER(*v*)
e1:	BITSTREAM VARIETY
e2:	BITSTREAM VARIETY
	→ VARIETY

The *control* is evaluated. It will be a constant at install time under the constant evaluation rules. If it is non-zero, *e1* is installed at this point and *e2* is ignored and never processed. If *control* is zero then *e2* is installed at this point and *e1* is ignored and never processed.

5.56.3 var_limits

Encoding number: 3

lower_bound:	SIGNED_NAT
upper_bound:	SIGNED_NAT
	→ VARIETY

lower_bound is the lower limit (inclusive) of the range of values which shall be representable in the resulting VARIETY, and *upper_bound* is the upper limit (inclusive).

5.56.4 var_width

Encoding number: 4

signed_width: BOOL
width: NAT
→ VARIETY

If *signed_width* is true then this construction is equivalent to **var_limits** ($-2^{width-1}$, $2^{width-1}-1$). If *signed_width* is false then this construction is **var_limits** (0, $2^{width}-1$).

5.57 VERSION_PROPS

Number of encoding bits: 0

Unit identification: versions

This UNIT gives information about version numbers and user information.

5.57.1 make_versions

Encoding number: 0

version_info: SLIST(VERSION)
→ VERSION_PROPS

This contains version information.

5.58 VERSION

Number of encoding bits: 1

Is coding extendable? Yes

5.58.1 make_version

Encoding number: 1

major_version:	TDFINT
minor_version:	TDFINT
	→ VERSION

These are the major and minor version numbers of the XANDF used. An increase in minor version number means an extension of facilities. An increase in major version number means an incompatible change. XANDF with the same major number but a lower minor number than the installer shall install correctly.

For XANDF conforming to this specification, the major number will be 4 and the minor number will be 0.

Every CAPSULE will contain at least one *make_version* construct.

5.58.2 user_info

Encoding number: 2

information:	STRING(*k*,*n*)
	→ VERSION

This is (usually character) information included in the XANDF for labelling purposes.

Chapter 6

Supplementary UNIT

6.1 LINKINFO_PROPS

Number of encoding bits: 0

Unit identification: linkinfo

This is an additional UNIT which gives extra information about linking.

6.1.1 make_linkinfos

Encoding number: 0

no_labels:	TDFINT
tds:	SLIST(LINKINFO)
	→ LINKINFO_PROPS

This makes the UNIT.

6.2 LINKINFO

Number of encoding bits: 2

Is coding extendable? Yes

6.2.1 static_name_def

Encoding number: 1

assexp:	EXP
id:	TDFSTRING
	→ LINKINFO

assexp will be an **obtain_tag** construction which refers to a TAG which is defined with **make_id_tagdef**, **make_var_tagdef** or **common_tagdef.** This TAG will not be linked to an EXTERNAL.

The name *id* shall be used (but not exported, that is, static) to identify the definition for subsequent linking.

Note: This construction is likely to be needed for profiling, so that useful names appear for statically defined objects. It may also be needed when C++ is translated into C, in order to identify global initialisers.

6.2.2 make_comment

Encoding number: 2

n: TDFSTRING
→ LINKINFO

n shall be incorporated into the object file as a comment, if this facility exists. Otherwise the construct is ignored.

6.2.3 make_weak_defn

Encoding number: 3

namer: EXP
val: EXP
→ LINKINFO

namer and *val* will be **obtain_tag** constructions which refer to TAGs which are defined with **make_id_tagdef**, **make_var_tagdef** or **common_tagdef.** They shall be made synonymous.

6.2.4 make_weak_symbol

Encoding number: 4

id: TDFSTRING
val: EXP
→ LINKINFO

val will be an **obtain_tag** construction which refers to a TAG which is defined with **make_id_tagdef**, **make_var_tagdef** or **common_tagdef**.

This TAG shall be made weak (in the same sense as in the SVID ABI Symbol Table), and *id* shall be synonymous with it.

Chapter 7

Notes

7.1 Binding

The following constructions introduce TAGs:

identify
variable
make_proc
make_general_proc
make_id_tagdec
make_var_tagdec
common_tagdec

During the evaluation of **identify** and **variable** a value, *v*, is produced which is bound to the TAG during the evaluation of an EXP or EXPs. The TAG is ''in scope'' for these EXPs. This means that in the EXP, a use of the TAG is permissible and will refer to the declaration.

The **make_proc** and **make_general_proc** construction introduces TAGs which are bound to the actual parameters on each call of the procedure. These TAGs are in scope for the body of the procedure.

If a **make_proc** or **make_general_proc** construction occurs in the body of another **make_proc** or **make_general_proc,** the TAGS of the inner procedure are not in scope in the outer procedure, nor are the TAGS of the outer in scope in the inner.

The **apply_general_proc** construction permits the introduction of TAGs whose scope is the *postlude* argument. These are bound to the values of caller parameters after the evaluation of the body of the procedure.

The **make_id_tagdec**, **make_var_tagdec** and **common_tagdec** constructions introduce TAGs which are in scope throughout all the *tagdef* UNITs. These TAGs may have values defined for them in the *tagdef* UNITs, or values may be supplied by linking.

The following constructions introduce LABELs:

conditional
repeat
labelled

The constructions themselves define EXPs for which these LABELs are ''in scope''. This means that in the EXPs, a use of the LABEL is permissible and will refer to the introducing construction.

TAGs, LABELs and TOKENs (as TOKEN parameters) introduced in the body of a TOKEN definition are systematically renamed in their scope each time the TOKEN definition is applied. The scope will be completely included by the TOKEN definition.

Each of the values introduced in a UNIT will be named by a different TAG, and the labelling constructions will use different labels, so no visibility rules are needed. The set of TAGs and LABELs used in a simple UNIT are considered separately from those in another simple UNIT, so no question of visibility arises. The compound and link UNITs provide a method of relating the items in one simple UNIT to those in another, but this is through the intermediary of another set of TAGs and TOKENs at the CAPSULE level.

7.2 Character Codes

XANDF does not have a concept of characters. It transmits integers of various sizes. So, if a producer wishes to communicate characters to an installer, it will usually have to do so by encoding them in some way as integers.

An ANSI C producer sending an XANDF program to a set of normal C environments may well choose to encode its characters using the ASCII codes; an EBCDIC based producer transmitting to a known set of EBCDIC environments might use the code directly; and a wide character producer might likewise choose a specific encoding. For some programs, this way of proceeding is necessary, because the codes are used both to represent characters and for arithmetic, so the particular encoding is enforced. In these cases it will not be possible to translate the characters to another encoding because the character codes will be used in the XANDF as ordinary integers, which must not be translated.

Some producers may wish to transmit true characters, in the sense that something is needed to represent particular printing shapes and nothing else. These representations will have to be transformed into the correct character encoding on the target machine.

Probably the best way to do this is to use TOKENs. A fixed representation for the printing marks could be chosen in terms of integers and TOKENs introduced to represent the translation from these integers to local character codes, and from strings of integers to strings of local character codes. These definitions could be bound on the target machine, and the installer should be capable of translating these constructions into efficient machine code. To make this a standard, unique TOKENs should be used.

However, this raises the question of who chooses the fixed representation and the unique TOKENs and their specification? Clearly, XANDF provides a mechanism for performing the standardisation without itself defining a standard.

Here, XANDF gives rise to the need for extra standards, especially in the specification of globally named unique TOKENs.

7.3 Constant Evaluation

Some constructions require an EXP argument which is constant at install time. For an EXP to satisfy this condition it must be constructed according to the following rules after substitution of token definitions and selection of *exp_cond* branches.

If it contains **obtain_tag** then the tag will be introduced within the EXP, or defined with **make_id_tagdef**.

It may not contain any of the following constructions:

apply_proc
apply_general_proc
assign_with_mode
contents_with_mode
continue
current_env
error_jump
goto_local_lv
make_local_lv
move_some
repeat
round_as_state

Any use of **contents** or **assign** will be applied only to POINTERS derived from **variable** constructions.

If it contains **labelled**, there will only be jumps to the LABELS from within *starter*, not from within any of the *places*.

Any use of **obtain_tag** defined with **make_id_tagdef** will occur after the end of the **make_id_tagdef**.

Note specifically that a constant EXP may contain **env_offset**.

7.4 Division and Modulus

Two classes of division(D) and remainder(M) construct are defined. The two classes have the same definition if both operands have the same sign. Neither is defined if the second argument is zero.

Class 1:

p D1 $q = n$

where:

$p = n * q + (p$ M1 $q)$

sign(p M1 q) = sign(q)

$0 <= | p$ M1 $q | < | q |$

Class 2:

p D2 $q = n$

where:

$p = n * q + (p$ M2 $q)$

sign(p M2 q) = sign(p)

$0 <= | p$ M2 $q | < | q |$

7.5 Equality of EXPs

A definition of equality of EXPs would be a considerable part of a formal specification of XANDF, and is not given here.

7.6 Equality of SHAPEs

Equality of SHAPEs is defined recursively.

Two SHAPEs are equal if they are both BOTTOM, or both TOP, or both PROC.

Two SHAPEs are equal if they are both integer or both **floating**, or both **bitfield**, and the corresponding parameters are equal.

Two SHAPEs are equal if they are both NOF, the numbers of items are equal and the SHAPE parameters are equal.

Two OFFSETs or two POINTERs are equal if their ALIGNMENT parameters are pairwise equal.

Two COMPOUNDs are equal if their OFFSET EXPS are equal.

No other pairs of SHAPEs are equal.

7.7 Equality of ALIGNMENTS

Two ALIGNMENTS are equal if and only if they are equal sets.

7.8 Exceptions and Jumps

XANDF allows simply for labels and jumps within a procedure, by means of the **conditional**, **labelled** and **repeat** constructions, and the **goto**, **case** and various **test** constructions. However, there are two more complex *jumping* situations.

First there is the jump, known to stay within a procedure, but to a computed destination. Many languages have discouraged this kind of construction, but it is still available in COBOL (implicitly), and it can be used to provide other facilities (see below). XANDF allows it by means of the POINTER({ *code* }). XANDF is arranged so that this can usually be implemented as the address of the label. The **goto_local_lv** construction just jumps to the label.

The other kind of construction needed is the jump out of a procedure to a label which is still active, restoring the environment of the destination procedure: this is the *long jump*. Related to this is the notion of *exception*. Unfortunately long jumps and exceptions do not co-exist well. Exceptions are commonly organised so that any necessary destruction operations are performed as the stack of frames is traversed; long jumps commonly go directly to the destination. XANDF must provide some facility which can express both of these concepts. Furthermore, exceptions come in several different versions, according to how the exception handlers are discriminated and whether exception handling is initiated if there is no handler which will catch the exception.

Fortunately the normal implementations of these concepts provide a suggestion as to how they can be introduced into XANDF. The local label value provides the destination address, the environment (produced by *current_env*) provides the stack frame for the destination, and the stack re-setting needed by the local label jumps themselves provides the necessary stack information. If more information is needed, such as which exception handlers are active, this can be created by producing the appropriate XANDF.

So, XANDF takes the **long_jump** as the basic construction, and its parameters are a local label value and an environment. Everything else can be built in terms of these.

The XANDF arithmetic constructions allow specifying a LABEL as destination if the result of the operation is exceptional. This is sufficient for the kind of explicit exception handling found in C++ and, in principle, could also be used to implement the kind of ‘‘automatic’’ exception detection and handling found in Ada, for example.

However, many architectures have facilities for automatically trapping on exceptions, without explicit testing. To take advantage of this, there is a **trap** ERROR_TREATMENT with associated ERROR_CODEs. The action taken on an exception with trap ERROR_TREATMENT will be to ‘‘throw’’ the ERROR_CODE. Since each language has its own idea of how to interpret the ERROR_CODE and handle exceptions, the onus is on the producer writer to describe how to throw an ERROR_CODE.

The producer writer must give a definition of a TOKEN ˜Throw : NAT → EXP where the NAT will be **error_val** for some ERROR_CODE. This token definition must be consistent with the interpretation of the ERROR_CODE parameter and the method of handling exceptions. Usually this will consist of decoding the ERROR_CODE and doing a **long_jump** on some globals set up by the procedure in which the exception handling takes place.

The translator writer will provide a parameterless EXP TOKEN, ˜Set_signal_handler. This TOKEN will use ˜Throw and must be applied before any possible exceptions. This implies that the definition of both ˜Throw and ˜Set_signal_handler must be bound before translation of any CAPSULE which uses them, presumably by linking with some XANDF libraries.

These tokens are specified in more detail in Section 9.4.1 on page 172.

7.9 Procedures

The *var_param* of an **apply_proc** and the *var_intro* of the corresponding **make_proc** will either both be present or both absent. If they are present, the body of the **make_proc** can access the actual parameter by using OFFSET arithmetic relative to the POINTER TAG. This provides a method of supplying a variable number of parameters, by composing them into a compound value which is supplied as the *var_param*.

However, this has proved to be unsatisfactory for the implementation of variable number of parameters in C — one cannot choose the POINTER alignment of the TAG independently from the actual parameters in non-prototype calls.

The definition of caller parameters in general procedures addresses this difficulty by describing the layout of caller parameters qualified by PROCPROPS **var_callers**. This allows both the call and the body to have the same view of the OFFSETs within a parameter set, regardless of whether or not the particular parameter has been named. Similar consideration applies to accessing within the callee parameters.

All uses of **return**, **untidy_return** and **tail_call** in a procedure will return values of the same SHAPE, and this will be the *result_shape* specified in all uses of **apply_proc** or **apply_general_proc** calling the procedure .

The use of **untidy_return** gives a generalisation of **local_alloc.** It extends the validity of pointers allocated by **local_alloc** within the immediately enclosing procedure into the calling procedure. The original space of these pointers may be invalidated by **local_free** just as if it had been generated by **local_alloc** in the calling procedure.

The PROCPROPS **check_stack** may be used to check that limit set by **set_stack_limit** is not exceeded by the allocation of the static locals of a procedure body to be obeyed. If it is exceeded, then the producer-defined TOKEN ~Throw: (NAT → EXP) will be invoked as ~Throw*(error_val(stack_overflow))*. Note that this will not include any space generated by **local_alloc**; an explicit test is required to do check these.

Any SHAPE is permitted as the *result_shape* in an **apply_proc** or **apply_general_proc**.

7.10 Frames

XANDF states that while a particular procedure activation is current, it is possible to create a POINTER, by using **current_env**, which gives access to all the declared variables and identifications of the activation which are alive and which have been marked as visible. The construction **env_offset** gives the OFFSET of one of these relative to such a POINTER. These constructions may serve for several purposes.

One significant purpose is to implement such languages as Pascal, which have procedures declared inside other procedures. One way of implementing this is by means of a display, that is, a tuple of frame pointers of active procedures.

Another purpose is to find active variables satisfying some criterion in all the procedure activations. This is commonly required for garbage collection. XANDF does not force the installer to implement a frame pointer register, since some machines do not work best in this way. Instead, a frame pointer is created only if required by **current_env.** The implication of this is that this sort of garbage collection needs the collaboration of the producer to create XANDF which makes the correct calls on **current_env** and **env_offset** and place suitable values in known positions.

Programs compiled especially to provide good diagnostic information can also use these operations.

In general, any program which wishes to manipulate the frames of procedures other than the current one can use **current_env** and **env_offset** to do so.

A frame consists of three components: the caller parameters, callee parameters, and locals of the procedure involved. Since each component may have different internal methods of access within the frame, each has a different special frame alignment associated with pointers within them. These are **callers_alignment**, **callees_alignment** and **locals_alignment**. The POINTER produced by **current_env** will be some union of these special alignments, depending on how the procedure was defined.

Each of these frame alignments are considered to contain any ALIGNMENT produced by **alignment** from any SHAPE. Note that this does not say that they are the set union of all such ALIGNMENTS. This is because the interpretation of **pointer** and **offset** operations (notably **add_to_pointer**) may be different, depending on the implementation of the frames; they may involve extra indirections.

Accordingly, because of the constraints on **add_to_ptr**, an OFFSET produced by **env_offset** can only be added to a POINTER produced by **current_env.** It is a further constraint that such an OFFSET will only be added to a POINTER produced from **current_env** used on the procedure which declared the TAG.

7.11 Lifetimes

TAGs are bound to values during the evaluation of EXPs, which are specified by the construction which introduces the TAG. The evaluation of these EXPs is called the *lifetime* of the activation of the TAG.

Note that *lifetime* is a different concept from that of *scope*. For example, if the EXP contains the application of a procedure, the evaluation of the body of the procedure is within the lifetime of the TAG, but the TAG will not be in scope.

A similar concept applies to LABELs.

7.12 Alloca

The constructions involving *alloca* (**last_local**, **local_alloc**, **local_free**, **local_free_all**) as well as the **untidy_return** construction imply a stack-like implementation which is related to procedure calls. They may be implemented using the same stack as the procedure frames, if there is such a stack, or it may be more convenient to implement them separately. However, note that if the *alloca* mechanism is implemented as a stack, this may be an upward or a downward growing stack.

The state of this notional stack is referred to here as the *alloca* state. The construction **local_alloc** creates a new space on the *alloca* stack, the size of this space being given by an OFFSET. In the special case that this OFFSET is zero, **local_alloc** in effect gives the current *alloca* state (normally a POINTER to the top of the stack).

A use of **local_free_all** returns the *alloca* state to what it was on entry to the current procedure.

The construction **last_local** gives a POINTER to the top item on the stack, but it is necessary to give the size of this (as an OFFSET) because this cannot be deduced if the stack is upward growing. This top item will be the whole of an item previously allocated with **local_alloc**.

The construction **local_free** returns the state of the *alloca* machine to what it was when its parameter POINTER was allocated. The OFFSET parameter will be the same value as that with which the POINTER was allocated.

The ALIGNMENT of the POINTER delivered by **local_alloc** is **alloca_alignment**. This shall include the set union of all the ALIGNMENTs which can be produced by alignment from any SHAPE.

The use of **alloca_alignment** arises so that the *alloca* stack can hold any kind of value. The sizes of spaces allocated must be rounded up to the appropriate ALIGNMENT. Since this includes all value ALIGNMENTS, a value of any ALIGNMENT can be assigned into this space. Note that there is no necessary relation with **frame_alignment**, though they must both contain all the ALIGNMENTs which can be produced by **alignment** from any SHAPE

Stack pushing is **local_alloc.** Stack popping can be performed by use of **last_local** and **local_free**. Remembering the state of the *alloca* stack and returning to it can be performed by using **local_alloc** with a zero OFFSET and **local_free**.

Note that stack pushing can also be achieved by the use of a procedure call with **untidy_return**.

A transfer of control to a local label by means of **goto**, **goto_local_lv,** any **test** construction or any **error_jump** will not change the *alloca* stack.

Note: If an installer implements **identify** and **variable** by creating space on a stack when they come into existence, rather than doing the allocation for **identify** and **variable** at the start of a procedure activation, then it may have to consider making the *alloca* stack into a second stack.

7.13 Memory Model

The layout of data in memory is entirely determined by the calculation of OFFSETs relative to POINTERs. That is, it is determined by OFFSET arithmetic and the **add_to_ptr** construction.

A POINTER is parameterised by the ALIGNMENT of the data indicated. An ALIGNMENT is a set of all the different kinds of basic value which can be indicated by a POINTER. That is, it is a set chosen from all VARIETIES, all FLOATING_VARIETIES, all BITFIELD_VARIETIES, *proc*, *code*, *pointer* and *offset*. There are also three special ALIGNMENTS, **frame_alignment**, **alloca_alignment** and **var_param_alignment**.

The parameter of a POINTER will not consist entirely of BITFIELD_VARIETIES.

The implication of this is that the ALIGNMENT of all procedures is the same, the ALIGNMENT of all POINTERs is the same, and the ALIGNMENT of all OFFSETS is the same.

At present this corresponds to the state of affairs for all machines. But it is certainly possible that, for example, 64-bit pointers might be aligned on 64-bit boundaries while 32-bit pointers are aligned on 32-bit boundaries. In this case it will become necessary to add different kinds of pointer to XANDF. This will not present a problem, because, to use such pointers, similar changes will have to be made in languages to distinguish the kinds of pointer if they are to be mixed.

The difference between two POINTERs is measured by an OFFSET. Hence an OFFSET is parameterised by two ALIGNMENTs, that of the start POINTER and that of the end POINTER. The ALIGNMENT set of the first must include the ALIGNMENT set of the second.

The parameters of an OFFSET may consist entirely of BITFIELD_VARIETYs.

The operations on OFFSETs are subject to various constraints on ALIGNMENTs. It is important not to read into offset arithmetic what is not there. Accordingly some rules of the algebra of OFFSETs are given below:

- **offset_add** is associative.
- **offset_mult** corresponds to repeated offset_addition.
- **offset_max** is commutative, associative and idempotent.
- **offset_add** distributes over **offset_max** where they form legal expressions.
- **offset_test***(prob, ≥ , a, b)* continues if **offset_max***(a,b) = a*.

7.13.1 Simple Model

An example of the representation of OFFSET arithmetic is given below. This is not a definition, but only an example. In order to make this example clear, a machine with bit-addressing is hypothesised. This machine is referred to as the *simple model*.

In this machine, ALIGNMENTs will be represented by the number by which the bit address of data must be divisible. For example, 8-bit bytes might have an ALIGNMENT of 8, longs of 32 and doubles of 64. OFFSETs will be represented by the displacement in bits from a POINTER. POINTERs will be represented by the bit address of the data. Only one memory space will exist. Then, in this example a possible conforming implementation would be as follows:

add_to_ptr is addition.

offset_add is addition.

offset_div and **offset_div_by_int** are exact division.

offset_max is maximum.

offset_mult is multiply.

offset_negate is negate.

offset_pad*(a,x)* is *((x + a* **-** *1) / a) * a*.

offset_subtract is subtract.

offset_test is **integer_test**.

offset_zero is 0.

shape_offset*(s)* is the minimum number of bits needed to be moved to move a value of SHAPEs .

Note that these operations only exist where the constraints on the parameters are satisfied. Elsewhere, the operations are undefined.

All the computations in this representation are obvious, but there is one point to make concerning **offset_max,** which has the following arguments and result:

arg1: EXP OFFSET (*x*,*y*)
arg2: EXP OFFSET (*z*,*y*)
→ EXP OFFSET(*unite_alignments(x,z)y*

The SHAPES could have been chosen to be:

arg1: EXP OFFSET (*x*,*y*)
arg2: EXP OFFSET (*z*,*t*)
→ EXP OFFSET(*unite_alignments(x,z),intersect_alignments(y,t)*)

where *unite_alignments* is **set union** and *intersect_alignments* is **set intersection**. This would have expressed the most general reality. The representation of *unite_alignments* (*x*, *z*) is the maximum of the representations of *x* and *z* in the simple model. Unfortunately the representation of *intersect_alignments* (*y*, *t*) is not the minimum of the representations of *y* and *t*. In other words, the simple model representation is not a homomorphism if *intersect_alignments* is used. Because the choice of representation in the installer is an important consideration, the actual definition was chosen instead. It seems unlikely that this will affect practical programs significantly.

7.13.2 Comparison of Pointers and Offsets

Two POINTERS to the same ALIGNMENT, *a*, are equal if and only if the result of **subtract_ptrs** applied to them is equal to **offset_zero** (*a*).

The comparison of OFFSETS is reduced to the definition of **offset_max** and the equality of OFFSETS in accordance with the definition in Section 5.16.89 on page 70.

7.13.3 Circular Types in Languages

It is assumed that circular types in programming languages will always involve the SHAPES PROC or POINTER (*x*) on the circular path in their representation. Since the ALIGNMENT of POINTER is { *pointer* } and does not involve the ALIGNMENT of the thing pointed at, circular SHAPES are not needed. The circularity is always broken in ALIGNMENT (or PROC).

7.13.4 Special Alignments

There are seven special ALIGNMENTS. One of these is **code_alignment** — the ALIGNMENT of the POINTER delivered by **make_local_lv**.

The ALIGNMENT of a parameter of SHAPE *s* is given by **parameter_alignment**(*s*) which will always contain **alignment**(*s*).

The other five special ALIGNMENTS are **alloca_alignment**, **callees_alignment**, **callers_alignment**, **locals_alignment** and **var_param_alignment**. Each of these contains the **set union** of all the ALIGNMENTs which can be produced by **alignment** from any SHAPE. However, they need not be equal to that **set union**, nor need there be any relation between them.

In particular they are not equal (in the sense of Section 7.7 on page 141).

Each of these five special ALIGNMENTs refer to alignments of various components of a frame.

Notice that pointers and offsets such as POINTER(*callees_alignment(true))* and OFFSET*(callees_alignment(true) ,x)*, and so on, can have some special representation, and that **add_to_ptr** and **offset_add** can operate correctly on these representations. However, it is necessary that:

alignment(POINTER(A)) = { pointer }

for any ALIGNMENT *A*.

7.13.5 Atomic Assignment

At least one VARIETY shall exist such that **assign** and **assign_with_mode** are atomic operations. This VARIETY shall be specified as part of the installer specification. It shall be capable of representing the numbers 0 to 127.

Note: It is not necessary for this to be the same VARIETY on each machine. Normal practice will be to use a TOKEN for this VARIETY and choose the definition of the TOKEN on the target machine.

7.14 Order of Evaluation

The order of evaluation is specified in certain constructions in terms of equivalent effect with a canonical order of evaluation. These constructions are **conditional**, **identify**, **labelled**, **repeat**, **sequence** and **variable**. Let these be called the *order-specifying* constructions.

The constructions which change control also specify a canonical order. These are **apply_proc**, **apply_general_proc**, **case**, **goto**, **goto_local_lv**, **long_jump**, **return**, **untidy_return**, **return_to_label**, **tail_call**, the **test** constructions, and all instructions containing the **error_jump** and **trap** ERROR_TREATMENT.

The order of evaluation of the components of other constructions is that the components may be evaluated in any order and with their components — down to the XANDF leaf level — interleaved in any order. The constituents of the order specifying constructions may also be interleaved in any order, but the order of the operations within an order specifying operation shall be equivalent in effect to a canonical order.

Note that the rule specifying when *error-jumps* or *traps* are to be taken (see Section 5.15.4 on page 38) relaxes the strict rule that everything has to be as if completed by the end of certain constructions. Without this rule, pipelines would have to stop at such points, in order to be sure of processing any errors. Since this is not normally needed, it would be an expensive requirement, so hence the existence of this rule. However, a construction will be required to force errors to be processed in the cases where this is important.

7.15 Original Pointers

Certain constructions are specified as producing original pointers. They allocate space to hold values and produce pointers to indicate that new space. All other pointer values are derived pointers, which are produced from original pointers by a sequence of **add_to_ptr** operations. Counting original pointers as being derived from themselves, every pointer is derived from just one original pointer.

A null pointer is counted as an original pointer.

If procedures are called which come from outside the XANDF world (such as *calloc*) it is part of their interface with XANDF to state if they produce original pointers, and what is the lifetime of the pointer.

As a special case, original pointers can be produced by using **current_env** and **env_offset**(see Section 5.16.25 on page 48).

Note that:

add_to_ptr(*p,offset_add(q, r)*)

is equivalent to:

add_to_ptr(*add_to_ptr(p,q),r*)

In the case that *p* is the result of **current_env** and *q* is the result of **env_offset**, **add_to_ptr** (*p*,*q*) is defined to be an original pointer. For any such expression, *q* will be produced by **env_offset** applied to a TAG introduced in the procedure in which **current_env** was used to make *p*.

7.16 Overlapping

In the case of **move_some,** or **assign** or **assign_with_mode** in which *arg2* is a **contents** or **contents_with_mode**, it is possible that the source and destination of the transfer might overlap.

In this case, if the operation is **move_some** or **assign_with_mode** and the TRANSFER_MODE contains overlap, then the transfer shall be performed correctly, that is, as if the data were copied from the source to an independent place and then to the destination.

In all cases, if the source and destination do not overlap, the transfer shall be performed correctly.

Otherwise the effect is undefined.

7.17 Incomplete Assignment

If the *arg2* component of an **assign** or **assign_with_mode** operation is left by means of a *jump*, the question arises as to what value is in the destination of the transfer.

If the TRANSFER_MODE **complete** is used, the destination shall be left unchanged if the *arg2* component is left by means of a *jump*. If **complete** is not used and *arg2* is left by a *jump*, the destination may be affected in any way.

7.18 Representing Integers

Integer VARIETIES shall be represented by a range of integers which includes those specified by the given bounds. This representation shall be twos-complement.

If the lower bound of the VARIETY is non-negative, the representing range shall be from 0 to $2^{8n} - 1$ for some n. n is called the number of bytes in the representation. The number of bits in the representation is $8n$.

If the lower bound of the VARIETY is negative, the representing range shall be from -2^{8n-1} to $2^{8n-1} - 1$ for some n. n is called the number of bytes in the representation. The number of bits in the representation is $8n$.

Installers may limit the size of VARIETY that they implement. A statement of such limits shall be part of the specification of the installer. In no case may such limits be less than 64 bits, signed or unsigned.

Note: It is intended that there should be no upper limit allowed at some future date.

Operations are performed in the representing VARIETY. If the result of an operation does not lie within the bounds of the stated VARIETY, but does lie in the representation, the value produced in that representation shall be as if the VARIETY had the lower and upper bounds of the representation. The implication of this is usually that a number in a VARIETY is represented by that same number in the representation.

If the bounds of a VARIETY, *v*, include those of a VARIETY, *w*, the representing VARIETY for *v* shall include or be equal to the representing VARIETY for *w*.

The representations of two VARIETIES of the form **var_limits** $(0, 2^n - 1)$ and **var_limits** $(-2^{n-1}, 2^{n-1} - 1)$ shall have the same number of bits and the mapping of their ALIGNMENTS into the target alignment shall be the same.

7.19 Overflow and Integers

It is necessary first to define what overflow means for integer operations and second to specify what happens when it occurs. The intention of XANDF is to permit the simplest possible implementation of common constructions on all common machines while allowing precise effects to be achieved, if necessary at extra cost.

Integer varieties may be represented in the computer by a range of integers which includes the bounds given for the variety. An arithmetic operation may therefore yield a result which is within the stated variety, or outside the stated variety but inside the range of representing values, or outside that range. Most machines provide instructions to detect the latter case; testing for the second case is possible but a little more costly.

In the first two cases, the result is defined to be the value in the representation. Overflow occurs only in the third case:

- If the ERROR_TREATMENT is **impossible**, overflow will not occur. If it should happen to do so, the effect of the operation is undefined.
- If the ERROR_TREATMENT is **error_jump**, a LABEL is provided to jump to if overflow occurs.
- If the ERROR_TREATMENT is **trap**(*overflow*), a producer_defined TOKEN:

  ~Throw: NAT → EXP

  must be provided. On an overflow, the installer will arrange that ~Throw*(error_val(overflow))* is evaluated.

The **wrap** ERROR_TREATMENT is provided so that a useful defined result may be produced in certain cases where it is usually easily available on most machines. This result is available on the assumption that machines use binary arithmetic for integers. This is certainly so at present, and there is no close prospect of other bases being used.

If a precise result is required, further arithmetic and testing may be needed which the installer may be able to optimise away if the word lengths happen to suit the problem. In extreme cases it may be necessary to use a larger variety.

7.20 Representing Floating Point

FLOATING_VARIETYs shall be implemented by a representation which has at least the properties specified.

Installers may limit the size of FLOATING_VARIETY which they implement. A statement of such limits shall be part of the specification of an installer.

The limit may also permit or exclude infinities.

Any installer shall implement at least one FLOATING_VARIETY with the following properties (compare with *IEEE doubles*):

1. *mantissa_digits* shall not be less than 53.
2. *minimum_exponent* shall not be less than 1023.
3. *maximum_exponent* shall not be less than 1022.

Operations are performed and overflows detected in the representing FLOATING_VARIETY.

Note: There shall be at least two FLOATING_VARIETIES, one occupying the same number of bytes and having the same alignment as a VARIETY representation, and one occupying twice as many bytes.

7.21 Floating Point Errors

The only permitted ERROR_TREATMENTs for operations delivering FLOATING_VARIETYs are **impossible, error_jump** and **trap***(overflow)*.

The kinds of floating point error which can occur depend on the machine architecture (especially whether it has IEEE floating point) and on the definitions in the ABI being obeyed.

Possible floating point errors depend on the state of the machine and may include *overflow*, *divide_by_zero*, *underflow*, *invalid_operation* and *inexact*. The setting of this state is performed outside XANDF (at present).

If an **error_jump** or **trap** is taken as the result of a floating point error, the operations to test what kind of error it was are outside the definition (at present).

7.22 Rounding and Floating Point

Each machine has a rounding state which shall be one of **to_nearest**, **toward_larger**, **toward_smaller** or **toward_zero**. For each operation delivering a FLOATING_VARIETY, except for **make_floating**, any rounding necessary shall be performed according to the rounding state.

7.23 Floating Point Accuracy

While it is understood that most implementations will use IEEE floating arithmetic operations, there are machines which use other formats and operations. It is intended that they should not be excluded from having XANDF implementations.

For XANDF to have reasonably consistent semantics across many platforms, one must have some minimum requirements on the accuracies of the results of the floating point operations defined in XANDF. The provisional requirements sketched below would certainly be satisfied by an IEEE implementation.

Let $\oplus$ be some primitive dyadic arithmetic operator and $\oplus'$ be its XANDF floating-point implementation. Let F be some FLOATING_VARIETY and F' be a representational variety of F.

Condition 1:
If a, b and $a \oplus b$ can all be represented exactly in F, then they will also be represented exactly in F'. Extending the ' notation in the obvious manner:

$$(a \oplus b)' = (a' \oplus' b')$$

This equality will also hold using the XANDF equality test, that is:

$$(a \oplus b)' =' (a' \oplus' b')$$

Condition 2:
The operator $\oplus'$ is monotonic in the sense apposite to the operator $\oplus$. For example, consider the operator +; if x is any number and a and b are as above:

$$(x > b) \quad ((a' +' x') \geq (a + b)')$$

and:

$$(x < b) \quad ((a' +' x') \leq (a + b)')$$

and so on, reflecting the weakening of the ordering after the operation from > to ≥ and < to ≤. Once again, the inequalities will hold for their XANDF equivalents, for example, ≥' and >'.

Similar conditions can be expressed for the monadic operations.

For the floating-point test operation, there are obvious analogues to both conditions. The weakening of the ordering in the monotonicity condition, however, may lead to surprising results, arising mainly from the uncertainty of the result of equality between floating numbers which cannot be represented exactly in F.

7.24 Representing Bitfields

BITFIELD_VARIETYs specify a number of bits and shall be represented by exactly that number of bits in twos-complement notation. Producers may expect them to be packed as closely as possible.

Installers may limit the number of bits permitted in BITFIELD_VARIETYs. Such a limit shall be not less than 32 bits, signed or unsigned.

Note: It is intended that there should be no upper limit allowed at some future date.

Some offsets of which the second parameter contains a BITFIELD alignment are subject to a constraint defined below. This constraint is referred to as *variety_enclosed*.

Note: The intent of this constraint is to force BITFIELDS to be implemented (in memory) as being included in some properly aligned VARIETY value.

The constraint applies to:

x : *offset* (*p*, *b*)

and to:

sh = *bitfield* (*bfvar_bits* (*s*,*n*))

where *alignment* (*sh*) is included in *b*.

The constraint is as follows:

There will exist a VARIETY, *v*, and *r* : *offset* (*p*, *q*) where *v* is in *q*.

offset_pad(*b, r*) ≤ *x*

and:

offset_pad(*b, r+sz (v)*) ≥ **offset_pad**(*b, x+sz (sh)*)

where the comparisons are in the sense of **offset_test**, + is **offset_add** and *sz* is **shape_offset**.

7.25 Permitted limits

An installer may specify limits on the sizes of some of the data SHAPES which it implements. In each case there is a minimum set of limits such that all installers shall implement at least the specified SHAPES. Part of the description of an installer shall be the limits it imposes. Installers are encouraged not to impose limits if possible, though it is not expected that this will be feasible for floating point numbers.

7.26 Least Upper Bound

The LUB of two SHAPEs, *a* and *b* is defined as follows:

- If *a* and *b* are equal shapes, then *a*.
- If *a* is BOTTOM then *b*.
- If *b* is BOTTOM then *a*.
- Otherwise TOP.

7.27 Read-only Areas

Consider three scenarios in increasingly static order:

- Dynamic loading.

  A new module is loaded, initialising procedures are obeyed and the results of these are then marked as read-only.

- Normal loading.

  An *ld* program is obeyed which produces various (possibly circular) structures which are put into an area which will be read-only when the program is obeyed.

- Using ROM.

  Data structures are created (again possibly circular) and burnt into ROM for use by a separate program.

In each case, program is obeyed to create a structure, which is then frozen. The special case when the data is, say, just a string is not sufficiently general.

This XANDF specification takes the attitude that the use of read-only areas is a property of how XANDF is used — a part of the installation process — and there should not be XANDF constructions to say that some values in a CAPSULE are read-only. Such constructions could not be sufficiently general.

7.28 Tag and Token Signatures

In an XANDF program there will usually be references to TAGs which are not defined in XANDF; their definitions are intended to be supplied by a host system in system specific libraries.

These TAGs will be declared (but not defined) in an XANDF CAPSULE and will be specified by external linkages of the CAPSULE with EXTERNALs containing either TDFIDENTS or UNIQUEs. In early development drafts of this specification, the external names required by system linking could only be derived from those EXTERNALs.

Version 4.0 gives an alternative method of constructing extra-XANDF names. Each global TAG declaration can now contain a STRING signature field which may be used to derive the external name required by the system.

This addition is principally motivated by the various name mangling schemes of C++. The STRING signature can be constructed by concatenations and token expansions. Suitable usages of TOKENs can ensure that the particular form of name-mangling can be deferred to installation time and hence allow, at least conceptually, linking with different C++ libraries.

As well as TAG declarations, TAG definitions are allowed to have signatures. The restriction that the signature (if present) of a TAG definition being identical to its corresponding definition could allow type checking across separately compiled CAPSULEs.

Similar considerations apply to TOKENs; although token names are totally internal to XANDF, it would allow checking that a token declared in one CAPSULE has the same type as its definition in another.

7.29 Dynamic Initialisation

The dynamic initialisation of global variables is required for languages like C++. Previous to version 4.0, the only initialisations permissible were those at load-time; in particular, no procedure calls were allowed in forming the initialising value. Version 4.0 introduces the constructor, **initial_value**, to remedy this situation.

Several different implementation strategies could be considered for this. Basically, one must ensure that all the **initial_value** expressions are transformed into assignments to globals in some procedure. One might expect that there would be one such procedure invented for each CAPSULE and that somehow this procedure is called before the main program.

This raises problems on how we can name this procedure so that it can be identified as being a special initialising procedure. Some UNIX linkers reserve a name like **_init** specially so that all instances of it from different modules can be called before the main procedure. This is a possible solution; however it would mean that the XANDF linker would have to be similarly modified to construct the calling sequences for the *init* procedures in the different CAPSULEs.

The use of the **chain_extern** construct provides a means of linking similar constructions in different CAPSULEs in a more generally applicable manner. Applying it to the initialisation procedure problem, each initialisation procedure is given the same name **˜init**, say, and will end with a (tail) call to an external procedure also given a distinguished name **˜prev**, say. By doing a linking with **chain_extern** on **˜init** and the link of **˜prev**, a set of CAPSULEs can be XANDF linked so that a **˜prev** of one CAPSULE will become the **˜init** of another, leaving the combined CAPSULE with one **˜init** which calls all of the others. Suitable linking could also ensure that the final **˜prev** is the main procedure of the program and thus a call of the final **˜init** will run the program.

This, of course, raises the question of how one uses a system linker to perform the same kind of operation for **.o** files produced by separately translated CAPSULEs. Some system linkers provide some help in this, but most would require a pre-pass on the **.o** files to resolve this chaining linkage.

Chapter 8

The Bit Encoding of XANDF

This is a description of the encoding used for XANDF.

Section 8.1 defines the basic level of encoding, in which integers consisting of a specified number of bits are appended to the sequence of bytes. Section 8.2 defines the second level of encoding, in which fundamental kinds of value are encoded in terms of integers of specified numbers of bits. Section 8.3 defines the third level, in which XANDF is encoded using the previously defined concepts.

8.1 The Basic Encoding

XANDF consists of a sequence of 8-bit bytes used to encode integers of a varying number of bits, from 1 to 32. These integers will be called basic integers.

XANDF is encoded into bytes in increasing byte index, and within the byte the most significant end is filled before the least significant. Let the bits within a byte be numbered from 0 to 7, 0 denoting the least significant bit and 7 the most significant. Suppose that the bytes up to *n–1* have been filled and that the next free bit in byte *n* is bit *k*. Then bits *k+1 to 7* are full and bits *0 to k* remain to be used. Now an integer of *d* bits is to be appended.

- If *d* is less than or equal to *k*, the *d* bits will occupy bits *k–d+1* to *k* of byte *n*, and the next free bit will be at bit *k–d*. Bit 0 of the integer will be at bit *k–d+1* of the byte, and bit *d–1* of the integer will be at bit *k*.
- If *d* is equal to *k+1*, the *d* bits will occupy bits *0* to *k* of byte *n* and the next free bit will be bit *7* of byte *n+1*. Bit *d–1* of the integer will be at bit *k* of the byte.
- If *d* is greater than *k+1*, the most significant *k+1* bits of the integer will be in byte *n*, with bit *d–1* at bit *k* of the byte. The remaining *d–k–1* least significant bits are then encoded into the bytes, starting at byte *n+1*, bit *7*, using the same algorithm (that is, recursively).

8.2 Fundamental Encodings

This section describes the encoding of TDFINT, TDFBOOL, TDFSTRING, TDFIDENT, BIT STREAM, BYTESTREAM, BYTE_ALIGN and extendable integers.

8.2.1 TDFINT

TDFINT encodes non-negative integers of unbounded size. The encoding uses octal digits encoded in 4-bit basic integers. The most significant octal digit is encoded first, the least significant last. For all digits except the last the 4-bit integer is the value of the octal digit. For the last digit the 4-bit integer is the value of the octal digit plus 8.

8.2.2 TDFBOOL

TDFBOOL encodes a boolean, true or false. The encoding uses a 1-bit basic integer, with 1 encoding true and 0 encoding false.

8.2.3 TDFSTRING

TDFSTRING encodes a sequence containing *n* non-negative integers, each of *k* bits. The encoding consists of first a TDFINT giving the number of bits; second a TDFINT giving the number of integers, which may be zero; and third it contains *n* *k*-bit basic integers, giving the sequence of integers required, the first integer being first in this sequence.

8.2.4 TDFIDENT

TDFIDENT also encodes a sequence containing *n* non-negative integers. These integers will all consist of the same number of bits, which will be a multiple of 8. It is a property of the encoding of the other constructions that TDFIDENTS will start on either bit 7 or bit 3 of a byte and end on bit 7 or bit 3 of a byte. It thus has some alignment properties which are useful to permit fast copying of sections of XANDF.

The encoding consists of first a TDFINT giving the number of bits; second a TDFINT giving the number of integers, which may be zero, then if the next free bit is not bit 7 of some byte, it is moved on to bit 7 of the next byte; and third it contains *n* *k*-bit integers. If the next free bit is not bit 7 of some byte, it is moved on to bit 7 of the next byte.

8.2.5 BITSTREAM

It can be useful to be able to skip a construction without reading through it. BITSTREAM provides a means of doing this.

A BITSTREAM encoding of X consists of a TDFINT giving the number of bits of encoding which are occupied by the X. Hence to skip over a BITSTREAM while decoding, one should read the TDFINT and then advance the bit index by that number of bits. To read the contents of a BIT STREAM encoding of X, one should read and ignore a TDFINT and then decode an X. There will be no spare bits at the end of the X, so reading can continue directly.

8.2.6 BYTESTREAM

It can be useful to be able to skip an XANDF construction without reading through it. BYTESTREAM provides a means of doing this while remaining byte aligned, so facilitating copying the XANDF. A BYTESTREAM will always start when the bit position is 3 or 7.

A BYTESTREAM encoding of X starts with a TDFINT giving a number, *n*. After this, if the current bit position is not bit 7 of some byte, it is moved to bit 7 of the next byte. The next *n* bytes are an encoding of X. There may be some spare bits left over at the end of X.

Hence to skip over a BYTESTREAM while decoding, one should read a TDFINT, *n*, move to the next byte alignment (if the bit position is not 7) and advance the bit index over *n* bytes. To read a BYTESTREAM encoding of X one should read a TDFINT, *n*, and move to the next byte, *b* (if the bit position is not 7), and then decode an X. Finally the bit position should be moved to *n* bytes after *b*.

8.2.7 BYTE_ALIGN

byte_align leaves the bit position alone if it is 7, and otherwise moves to bit 7 of the next byte.

8.2.8 Extendable Integer Encoding

A *d*-bit extendable integer encoding enables an integer greater than zero to be encoded given *d*, a number of bits.

If the integer is between 1 and $2^d - 1$ inclusive, a *d*-bit basic integer is encoded.

If the integer, *i*, is greater than or equal to 2^d a *d*-bit basic integer encoding of zero is inserted and then $i - 2^d + 1$ is encoded as a *d*-bit extendable encoding.

8.3 The XANDF Encoding

The descriptions of SORTS and constructors contain encoding information which is interpreted as follows to define the XANDF encoding:

1. An XANDF CAPSULE is an encoding of the SORT CAPSULE.
2. For each SORT, a number of encoding bits, *b*, is specified. If this is zero, there will only be one construction for the class, and its encoding will consist of the encodings of its components, in the given order.
3. If the number of encoding bits, *b,* is not zero, the SORT is described as extendable or as not extendable. For each construction there is an encoding number given. If the SORT is extendable, this number is output as an extendable integer. If the SORT is described as not extendable, the number is output as a basic integer. This is followed by the encodings of the components of the construction in the order given in the description of the construct.
4. For the classes which are named SLIST (*x*) — for example, SLIST (UNIT) — the encoding consists of a TDFINT, *n,* followed by *n* encodings of *x*.
5. For the classes which are named LIST (*x*) — for example, LIST (EXP) — the encoding consists of a 1-bit integer which will be 0, followed by an SLIST (*x*). The 1-bit integer is to allow for extensions to other representations of LISTS.
6. For the classes which are named OPTION (*x*), the encoding consists of a 1-bit basic integer. If this is zero, the option is absent and there is no more encoding. If the integer is 1, the option is present and an encoding of *x* follows.
7. BITSTREAMS occur in only two kinds of place. One is the constructions with the form *x_cond*, which are the install-time conditionals. For each of these the class encoded in the BITSTREAM is the same as the class which is the result of the *x_cond* construction. The other kind of place is as the *token_args* component of a construction with the form *x_apply_token*. This component always gives the parameters of the TOKEN. It can only be decoded if there is a token definition or a token declaration for the particular token being applied, that is, for the *token_value* component of the construction. In this case, the SORTS and hence the classes of the actual token arguments are given by the declaration or definition, and encodings of these classes are placed in sequence after the number of bits. If the declaration or definition are not available, the BITSTREAM can only be skipped.
8. BYTESTREAM *X* occurs in only one place, the encoding of the SORT UNIT. The SORT *X* is determined by the UNIT identification which is given for each of the relevant SORTS.
9. The *tld* UNIT is encoded specially. It is always the first UNIT in a Capsule and is used to pass information to the XANDF linker.

The first entry in a *tld* UNIT is a TDFINT giving the format of the remainder of the UNIT. Currently, the linker supports formats 0 and 1, but others may be added to give greater functionality while retaining compatibility. With format 0, the remainder of UNIT is identical to a now obsolete *tld2* UNIT. With format 1, the remainder of the UNIT is as follows:

> If *n* is the number of EXTERN_LINKs in the *external_linkage* argument of **make_capsule,** the remainder consists of *n* sequences. These sequences are in the order given by *external_linkage*. Each element of a sequence consist of one TDFINT per linkable entity in the corresponding element of the **make_extern_link** in the same order. These integers will describe proper ties of the corresponding external links.
>
> (In format 0, there are only two sequences, the first describing the token external links and the second describing the tag external links.)
>
> Bit 0: 1 means used in this capsule, 0 means not used in this capsule.
>
> Bit 1: 1 means declared in this capsule, 0 means not declared in this capsule.
>
> Bit 2: 0 means not defined in this capsule, 1 means defined in this capsule.
>
> Bit 3: 0 means is uniquely defined, 1 means may be defined multiple times.

8.4 File Formats

There may be various kinds of files which contain XANDF bitstream information. Each will start with a 4 byte magic-number identifying the kind of file, followed by 2 TDFINTs giving the major and minor version numbers of the XANDF involved.

A CAPSULE file will have a magic-number TDFC. The encoding of the CAPSULE will be byte-aligned following the version numbers.

An XANDF library file will have a magic-number TDFL. These files are constructed by the XANDF linker.

An XANDF archive file will have a magic-number TDFA.

Other file formats introduced should follow a similar pattern.

The XANDF linker will refuse to link XANDF files with different major version numbers. The resulting minor version number is the maximum of component minor version numbers.

Chapter 9

Token Register

9.1 Introduction

9.1.1 Background

XANDF tokens offer a general encapsulation and expansion mechanism which allows any implementation detail to be delayed to the most appropriate stage of program translation. This provides a means for encapsulating any target dependencies in a neutral form, with specific implementations defined through standard XANDF features. This raises a natural opportunity for well-understood sets of XANDF tokens to be included along with XANDF itself as interface between XANDF tools.

9.1.2 Token Register Objectives

As XANDF tokens may be used to represent any piece of XANDF, they may be used to supplement any XANDF interface between software tools. However, that raises the issue of control authority for such an interface. In many cases, the interfaces may be considered to ‘‘belong’’ to a particular tool. In other cases, the names and specifications of tokens need to be recorded for common use.

This token register is used to record the names and specifications of tokens which may need to be assumed by more than one software tool. It also defines a naming scheme which should be used consistently to avoid ambiguity between tokens.

Five classes of tokens are identified:

1. target dependency tokens, which are concerned with describing target architecture or translator detail
2. basic mapping tokens, which relate general language features to architecture detail
3. XANDF interface tokens, which may be required to complete the specification of some XANDF constructs
4. language programming interfaces (LPI) which may be specific to a particular producer
5. application programming interfaces (API).

These classes are discussed separately, in Section 9.2 through Section 9.6 below.

9.1.3 Naming Scheme

A flat namespace will suffice for XANDF token names if producer writers adopt the simple constraints described here. XANDF has separate provision for a hierarchic unique naming scheme, but that was intended for a specific purpose which has not yet been realised.

External names for program or application specific tokens should be confined to ‘‘simple names’’, which we define to mean that they consist only of letters, digits and underscore (the characters allowed in C identifiers). Normally there will be very few such external names, as tokens internal to a single capsule do not require to be named. All other token names will consist of some controlled prefix followed by a simple name, with the prefix identifying the control authority.

For API tokens, the prefix will consist of a sequence of simple names, each followed by a dot, where the first simple name is the name of the API as listed or referred to in Section 9.6 on page 176.

The prefix for producer-specific and target-dependency tokens will begin and end with characters that distinguish them from the above cases. However, common XANDF tools such as DISP, TNC and PL-TDF assume that token names contain only letters, digits, underscore, dot, and/or tilde (˜).

The following prefixes are currently reserved:

- ˜

  XANDF interface tokens (as specified in the **XANDF Interface Tokens**, Section 9.4 on page 172) and LPI tokens specific to a C producer (see Section 9.5.1 on page 174)

- .˜

  Registered target dependency tokens (as specified in the **Target Dependency Tokens** Section 9.2) and basic mapping tokens (as specified in **Basic Mapping Tokens** Section 9.3 on page 169)

- .ET˜

  LPI tokens specific to a Fortran producer (see example in Section 9.5.2 on page 175).

9.2 Target Dependency Tokens

Target dependency tokens provide a common interface to simple constructs where the required detail for any specific architecture can be expressed within XANDF, but the detail will be architecture specific. Every installer should have associated with it a capsule containing the installer specific definitions of all the tokens specified within this section.

Some of these tokens provide information about the integer and floating point variety representations supported by an installer, in a form that may be used by XANDF analysis tools for architecture specific analysis, or by library generation tools when generating an architecture specific version of a library. Other target dependency tokens provide commonly required conversion routines.

It is recommended that these tokens should not be used directly within application programs. They are designed for use within LPI definitions, which can provide a more appropriate interface for applications.

9.2.1 Integer Variety Representations

Since XANDF specifies integer representations to be twos-complement, the number of bits required to store an integer variety representation fully specifies that representation. The minimum or maximum signed or unsigned integer that can be represented within any variety representation can easily be determined from the number of bits.

.~r ep_var_width

w: NAT
→ NAT

If w lies within the range of VARIETY sizes supported by the associated installer, *rep_var_width(w)* will be the number of bits required to store values of VARIETY *var_width(b,w)* for any BOOL *b*.

If *w* is outside the range of VARIETY sizes supported by the associated installer, *rep_var_width(w)* will be 0.

.~rep_atomic_width

→ NAT

.~rep_atomic_width will be the number of bits required to store values of some VARIETY, *v*, such that **assign** and **assign_with_mode** are atomic operations if the value assigned has SHAPE integer (*v*). The XANDF specification guarantees existence of such a number.

9.2.2 Floating Variety Representations

Floating point representations are much more diverse than integers, but we may assume that each installer will support a finite set of distinct representations. For convenience in distinguishing between these representations within architecture specific XANDF, the set of distinct representations supported by any specific installer are stated to be ordered into a sequence of non-decreasing memory size. An analysis tool can easily count through this sequence to determine the properties of all supported representations, starting at 1 and using **.~rep_fv_width** to test for the sequence end.

.~rep_fv

n: NAT
→ FLOATING_VARIETY

.~rep_fv(n) will be the FLOATING_VARIETY whose representation is the *n*th of the sequence of supported floating point representations. *n* will lie within this range.

.~rep_fv_width

n: NAT
→ NAT

If *n* lies within the sequence range of supported floating point representations, *.~rep_fv_width(n)* will be the number of bits required to store values of FLOATING_VARIETY *.~rep_fv(n)*.

If *n* is outside the sequence range of supported floating point representations, *.~rep_fv_width(n)* will be 0.

.~rep_fv_radix

n: NAT
→ NAT

.~rep_fv_radix(n) will be the radix used in the representation of values of FLOATING_VARIETY *.~rep_fv(n)*

n will lie within the sequence range of supported floating point representations.

.~rep_fv_mantissa

n: NAT
→ NAT

.~rep_fv_mantissa(n) will be the number of base *.~rep_fv_radix(n)* digits in the mantissa representation of values of FLOATING_VARIETY *.~rep_fv(n)*.

n will lie within the sequence range of supported floating point representations.

.~rep_fv_min_exp

n: NAT
→ NAT

.~rep_fv_min_exp(n) will be the maximum integer *m* such that $(.\tilde{}rep\_fv\_radix(n))^{-m}$ is exactly representable (though not necessarily normalised) by the FLOATING_VARIETY *.~rep_fv(n)*.

n will lie within the sequence range of supported floating point representations.

.~rep_fv_max_exp

n: NAT
→ NAT

.~rep_fv_max_exp(n) will be the maximum integer *m* such that $(.\tilde{}rep\_fv\_radix(n))^{-m}$ is exactly representable by the FLOATING_VARIETY *.~rep_fv(n)*.

n will lie within the sequence range of supported floating point representations.

.~rep_fv_epsilon

n: NAT
→ EXP FLOATING .~rep_fv(*n*)

.~rep_fv_epsilon(n) will be the smallest strictly positive real *x* such that (1.0 + *x*) is exactly representable by the FLOATING_VARIETY *.~rep_fv(n)*.

n will lie within the sequence range of supported floating point representations.

.~rep_fv_min_val

n: NAT
→ EXP FLOATING .~rep_fv(*n*)

.~rep_fv_min_val(n) will be the smallest strictly positive real number that is exactly representable (though not necessarily normalised)) by the FLOATING_VARIETY *.~rep_fv(n)*.

n will lie within the sequence range of supported floating point representations.

.~rep_fv_max_val

n: NAT
→ EXP FLOATING .~rep_fv(*n*)

.~rep_fv_max_val(n) will be the largest real number that is exactly representable by the FLOATING_VARIETY *.~rep_fv(n)*.

n will lie within the sequence range of supported floating point representations.

9.2.3 Non-numeric Representations

.~ptr_width

→ NAT

.~ptr_width will be the minimum *.~rep_var_width(w)* for any *w* such that any pointer to any alignment may be converted to an integer of VARIETY *var_width(b,w)*, for some BOOL *b*, and back again without loss of information, using the conversions *.~ptr_to_int* and *.~int_to_ptr* (see Section 9.2.4 on page 168).

.~best_div

→ NAT

.~best_div is 1 or 2 to indicate preference for class 1 or class 2 division and modulus (as defined in Section 7.4 on page 140). This token would be used in situations where either class is valid but must be used consistently.

.~little_endian

BOOL

.~little_endian is a property of the relationship between different variety representations and arrays. If an array of a smaller variety can be mapped onto a larger variety, and *.~little_endian* is true, then smaller indices of the smaller variety array map onto smaller ranges of the larger variety. If *.~little_endian* is false, no such assertion can be made.

9.2.4 Common Conversion Routines

This subsection contains a set of conversion routines between values of different shapes, that are not required to have any specific meaning apart from reversibility. If the storage space requirements for the two shapes are identical, the conversion can usually be achieved without change of representation. When that is the case, and if the two shapes can be stored at a common alignment, the conversion can simply be achieved by assignment via a common union, which will ensure the required alignment consistency.

.~ptr_to_ptr

a1: ALIGNMENT
a2: ALIGNMENT
p: EXP POINTER(*a1*)
→ EXP POINTER(*a2*)

.~ptr_to_ptr converts pointers from one pointer shape to another.

If *p* is any pointer with alignment a1, then *.~ptr_to_ptr(a2 ,a1 ,.~ptr_to_ptr(a1 ,a2 , p))* shall result in the same pointer *p,* provided that the number of bits required to store a pointer with alignment *a2* is not less than that required to store a pointer with alignment *a1*.

.~ptr_to_int

a: ALIGNMENT
v: VARIETY
p: EXP POINTER(*a*)
→ EXP INTEGER(*v*)

.~ptr_to_int converts a pointer to an integer. The result is undefined if the VARIETY *v* is insufficient to distinguish between all possible distinct pointers *p* of alignment *a*.

.~int_to_ptr

v: VARIETY
a: ALIGNMENT
i: EXP INTEGER(*v*)
→ EXP POINTER(*a*)

.~int_to_ptr converts an integer to a pointer. The result is undefined unless the integer *i* was obtained without modification from some pointer using *.~ptr_to_int* with the same variety and alignment arguments.

If *p* is any pointer with alignment *a,* and *v* is **var_width***(b ,.~ptr_width)* for some BOOL *b*, then
.~int_to_ptr(v ,a ,.~ptr_to_int(a , v , p))
shall result in the same pointer *p*.

.~f_to_ptr

a:	ALIGNMENT
fn:	EXP PROC
	→ EXP POINTER(*a*)

.~f_to_ptr converts a procedure to a pointer. The result is undefined except as required for consistency with *.~ptr_to_f*.

.~ptr_to_f

a:	ALIGNMENT
p:	EXP POINTER(*a*)
	→ EXP PROC

.~ptr_to_f converts a pointer to a procedure. The result is undefined unless the pointer *p* was obtained without modification from some procedure *f* using *.~f_to_ptr(a , f)*. The same procedure *f* is delivered.

9.3 Basic Mapping Tokens

Basic mapping tokens provide-target specific detail for specific language features that are defined to be target-dependent. This detail need not be fixed for a particular target architecture, but needs to provide compatibility with any external library with which an application program is to be linked.

Tokens specific to the C and Fortran language families are included. Like the target dependency tokens, it is again recommended that these tokens should not be used directly within application programs. They are designed for use within LPI definitions, which can provide a more appropriate interface for applications.

Every operating system variant of an installer should have associated with it a capsule containing the definitions of all the tokens specified within this section.

9.3.1 C Mapping Tokens

.~char_width

→ NAT

.~char_width is the number of bits required to store values of the representation VARIETY that corresponds to the C type **char**.

.~short_width

→ NAT

.~short_width is the number of bits required to store values of the representation VARIETY that corresponds to the C type **short int**.

.~int_width

→ NAT

.~int_width is the number of bits required to store values of the representation VARIETY that corresponds to the C type
B int .

.~long_width

→ NAT

.~long_width is the number of bits required to store values of the representation VARIETY that corresponds to the C type **long int**.

.~size_t_width

→ NAT

.~size_t_width is the number of bits required to store values of the representation VARIETY that corresponds to the C type **size_t**. It will be the same as one of *.~short_width*, *.~int_width* or *.~long_width*.

.~fl_rep

→ NAT

.~fl_rep is the sequence number (see Section 9.2.2 on page 165) of the floating point representation to be used for values of C type **float**.

.~dbl_rep

→ NAT

.~dbl_rep is the sequence number of the floating point representation to be used for values of C type **double**.

.~ldbl_rep

→ NAT

.~ldbl_rep is the sequence number of the floating point representation to be used for values of C type **long double**.

.~pv_align

→ ALIGNMENT

.~*pv_align* is the common alignment for all pointers that can be represented by the C generic pointer type **void***. For architecture independence, this would have to be a union of several alignments, but for many installers it can be simplified to

alignment(integer(var_width(false,.~char_width))).

.~min_struct_rep

→ NAT

.~*min_struct_rep* is the number of bits required to store values of the smallest C integral type which share the same alignment properties as a structured value whose members are all of that same integral type. It will be the same as one of .~*char_width*, .~*short_width*, .~*int_width* or .~*long_width*.

.~char_is_signed

→ BOOL

.~*char_is_signed* is true if the C type **char** is treated as signed, or false if it is unsigned.

.~bitfield_is_signed

→ BOOL

.~*bitfield_is_signed* is true if bitfield members of structures in C are treated as signed, or false if unsigned.

9.3.2 Fortran Mapping Tokens

.~F_char_width

→ NAT

.~*F_char_width* is the number of bits required to store values of the representation VARIETY that corresponds to the Fortran77 type **CHARACTER**.

In most cases, .~*F_char_width* is the same as .~*char_width*.

.~F_int_width

→ NAT

.~*F_int_width* is the number of bits required to store values of the representation VARIETY that corresponds to the Fortran77 type **INTEGER**.

In most cases, .~*F_int_width* is the same as .~*int_width*.

.~F_fl_rep

→ NAT

.~F_fl_rep is the sequence number (see Section 9.2.2 on page 165) of the floating point representation to be used for values of Fortran77 type **REAL**, with the constraint that:

.~rep_fv_width(.~F_fl_rep) = .~F_int_width.

If this constraint cannot be met, *.~F_fl_rep* will be 0.

.~F_dbl_rep

→ NAT

.~F_dbl_rep is the sequence number of the floating point representation to be used for values of Fortran77 type **DOUBLE PRECISION**, with the constraint that
*.~rep_fv_width(.~F_dbl_rep) = 2 * .~F_int_width.*

If this constraint cannot be met, *.~F_dbl_rep* will be 0.

9.4 XANDF Interface Tokens

A very few specifically named tokens are referred to within this XANDF specification which are required to complete the ability to use certain XANDF constructs. Responsibility for providing appropriate definitions for these tokens is indicated with the specifications below.

The interface to an optional Diagnostic Extension is also introduced.

9.4.1 Exception Handling

~Throw

n: NAT
→ EXP BOTTOM

The EXP *e* defined as the body of this token will be evaluated on occurrence of any error whose ERROR_TREATMENT is *trap*. The type of error can be determined within *e* from the NAT *n*, which will be **error_val***(ec)* for some ERROR_CODE *ec*. The token definition body *e* will typically consist of a *long_jump* to some previously set exception handler.

Exception handling using *trap* and ~Throw will usually be determined by producers for languages that specify their own exception handling semantics. Responsibility for the ~Throw token definition will therefore normally rest with producers, by including this token within the producer specific LPI.

˜Set_signal_handler

→ EXP OFFSET(*locals_alignment*, *locals_alignment)*

˜Set_signal_handler must be applied before any use of the ERROR_TREATMENT trap, to indicate the need for exception trapping. Responsibility for the ˜Set_signal_handler token definition will rest with installers. Responsibility for applying it will normally rest with producers.

The resulting offset value will contain the amount of space beyond any stack limit, which must be reserved for use when handling a *stack_overflow* trap raised by exceeding that limit.

˜Sync_handler

→ EXP TOP

˜Sync_handler delays subsequent processing until any pending exceptions have been raised, which is necessary to synchronise exception handler modification. It must be applied immediately prior to any action that modifies the effect of ˜Throw, such as assignment to a variable holding an exception handler as *long_jump* destination. Responsibility for the ˜Sync_handler token definition will rest with installers. Responsibility for applying it will normally rest with producers.

9.4.2 Diagnostic Extension

An optional extension to XANDF can be used to provide program diagnostic information that can be transformed by installers to the form required by popular platform-specific debuggers.

This extension is not part of this XANDF specification. However, the following description identifies the 4 tokens involved and outlines their form.

˜exp_to_source
˜diag_id_source
˜diag_type_scope
˜diag_tag_scope

bdy: EXP
. . .: . .
→ EXP

Each of these 4 tokens has several arguments of which the first, *bdy*, is an EXP. In each case, the default definition body, when no diagnostic information is required, is simply *bdy*.

Note that this description is quite sufficient to enable installers to ignore any diagnostic information that may be included in produced XANDF.

9.5 Language Programming Interfaces

A Language Programming Interface (LPI) is here defined to mean a set of tokens, usually specific to a particular producer, which will encapsulate language features at a higher level than basic XANDF constructs, more convenient for the producer to produce.

Responsibility for the specification of individual LPIs lies with the appropriate producer itself. Before an application can be installed on some target platform, the appropriate LPI token definitions must have been built for that platform. In this sense, the LPI can be considered as a primitive API, which is discussed in Section 9.6 on page 176.

The process by which the LPI token definition library or capsule is generated for any specific platform will vary according to the LPI, and responsibility for defining that process will also lie with the appropriate producer. Some LPIs can be fully defined by architecture neutral XANDF, using the tokens specified in Section 9.2 on page 164 and Section 9.3 on page 169 to encapsulate any target dependencies. When that is the case, the generation process can be fully automated. For other LPIs the process may be much less automated. In some cases where the source language implies a complex run-time system, this might even require a small amount of new code to be written for each platform.

Generally, the individual LPI tokens do not need to be specified in the token registry, provided they follow a registered naming scheme to ensure uniqueness (see Section 9.1 on page 163). In exceptional circumstances it may be necessary for some XANDF tool to recognise individual LPI tokens explicitly by name. This will be the case when experimenting with potential extensions to XANDF, for example in the field of parallelism. In other cases, an XANDF installer or other tool may recognise an LPI token by name rather than its definition by choice, for some unspecified advantage. It is a pragmatic choice in such cases whether to include such token specifications in the token registry. For widely used producers, one can assume availability of the LPI token specifications, or standard definitions, separately from the token register, but one should expect any such tokens to be specified within the register for all cases where significant advantage could be taken by an installer only if it recognises the token by name.

9.5.1 C Producer LPI

A C producer LPI can be defined by an architecture neutral token definition capsule provided with the producer. In this case, target specific detail is included only by use of the target dependency tokens and C mapping tokens specified in Section 9.2 on page 164 and Section 9.3.1 on page 169, respectively. Target-specific versions of this capsule are obtained by transformation, using an XANDF simplification tool to bind in the definitions of the target dependency and C mapping tokens that are provided with the target installer. No special treatment is required for any of the C LPI tokens, though translation time can be slightly improved in a few cases if the names are recognised and standard token definition exercised explicitly within some installers.

Such a C LPI would not need to include standard library features, for which the C language requires header files. The standard C library is one example of an API, discussed in Section 9.6 on page 176.

9.5.2 Fortran LPI

The following tokens are suggested, though subject to detailed specification of argument and result SORTs, as an example in case any installers may be able to produce better code than could be achieved by normal token expansion. In particular, some installers may be able to inline standard function calls.

.Et˜EXP: Exponential (e ** x) of any floating variety, including complex.

.Et˜LOG: (Natural) logarithm of any floating variety, including complex.

.Et˜LOG_10: Base 10 logarithm of any floating variety, including complex.

.Et˜LOG_2: Base 2 logarithm of any floating variety, including complex.

.Et˜SIN: Sine of any floating variety, including complex.

.Et˜COS: Cosine of any floating variety, including complex.

.Et˜TAN: Tangent of any floating variety, including complex.

.Et˜ASIN: Inverse sine of any floating variety, including complex.

.Et˜ACOS: Inverse cosine of any floating variety, including complex.

.Et˜ATAN: Inverse (one argument) tangent of any floating variety, including complex.

.Et˜ATAN2: Inverse (two arguments) tangent of any floating variety, excluding complex.

.Et˜SINH: Hyperbolic sine of any floating variety, including complex.

.Et˜COSH: Hyperbolic cosine of any floating variety, including complex.

.Et˜TANH: Hyperbolic tangent of any floating variety, including complex.

.Et˜ASINH: Inverse hyperbolic sine of any floating variety, including complex.

.Et˜ACOSH: Inverse hyperbolic cosine of any floating variety, including complex.

.Et˜ATANH: Inverse hyperbolic tangent of any floating variety, including complex.

.Et˜MOD: Floating point remainder of any floating variety, excluding complex.

9.6 Application Programming Interfaces

Application Programming Interfaces (APIs) are typically specified with a C mapping, which defines the required contents for C header files which a portable C program must include by name to gain access to target-specific implementations of an API library. The XANDF approach to API specification includes using a ‘‘#pragma’’ token syntax within architecture neutral C header files, such that all implementation dependencies are encapsulated by API specific tokens. These API tokens are the XANDF representation of the API. Both the API library and API token definitions are required before an XANDF program using the API can be installed on any particular platform.

Platform-specific definitions for API tokens are produced automatically, with few exceptions, for any platform with a conformant implementation of the API. This is achieved by a token library building process which analyses the architecture neutral header files for the API concerned, together with the platform specific header files that provide normal (non-XANDF) C access to the API. The few exceptions occur where the platform-specific header files have been written to make use of specific C compiler built-in features, typically recognised by identifiers with a prefix such as ‘‘_builtin_’’. Such cases are very likely to require explicit recognition of the corresponding token name in XANDF installers.

Generally, API token names and specifications are not detailed in this token register. The token specifications are clearly dependent on the associated API specifications. Authority for controlling the actual API token names, and the relationship between API tokens and the various API standardisation authorities, remain separate subjects of discussion.

Names and specifications are given or implied below for those API tokens which frequently require built-in support from installers, and for other cases where an installer may be able to produce better code than could be achieved by normal token expansion, for example by inlining standard function calls.

9.6.1 ANSI C Standard Functions

The set of tokens implied below all have the form:

ansi.<header>.<function>

. . . : . . .
→ EXP

Tokens are defined for all cases where **<header>** is *ctype* or *string* or *math* or *stdlibn*, and **<function>** is the name of a function specified in the ANSI C standard library, declared within the corresponding header **<header.h>**.

These tokens have arguments all of SORT EXP, whose number and shape, and token result shape, all correspond to the implementation shape of the named ANSI C standard library function parameters and result. For the few cases where the function is specified not to return (for example, *ansi.stdlib.abort*), the result shape may be either TOP or BOTTOM.

9.6.2 Common Exceptional Cases

ansi.setjmp.setjmp

jb: EXP
→ EXP

ansi.setjmp.setjmp is a token which has the semantics and argument and result implementation shapes corresponding to the ANSI C macro **setjmp** declared within **<setjmp.h>**.

ansi.setjmp.longjmp

jb: EXP
v: EXP
→ EXP

ansi.setjmp.longjmp is a token which has the semantics and argument implementation shapes corresponding to the ANSI C macro **longjmp** declared within **<setjmp.h>**. The result shape may be either TOP or BOTTOM.

˜alloca

i: EXP
→ EXP

˜alloca is a token which has the semantics and argument and result implementation shapes corresponding to the BSD specified function **alloca**.

Glossary

ABI
Application Binary Interface

ANDF
Architecture Neutral Distribution Format

ANSI
American National Standards Institute

API
Application Programming Interface.

argument
Information which is passed to a function or operation and which specifies the details of the processing to be performed.

atom
An atom is a unique ID corresponding to a string name. Atoms are used to identify properties, types and selections.

compiling
production of ANDF from some source language.

DRA
The United Kingdom Defence Research Agency.

EXP
short-form for ‘‘expression’’.

installing
linking and mapping of ANDF onto a concrete machine using processor-specific libraries implementing the API calls and data formats.

LPI
Language Programming Interface

POSIX
IEEE Portable Operating System Interface

producing
production of ANDF from some source language (same meaning as ‘‘compiling’’).

translating
making a program for some specific platform from ANDF

XANDF
X/Open specification for the Architecture Neutral Distribution Format

Index