TDF Specification

Issue 2.1 (June 1993)

Defence Research Agency
St Andrews Rd
Malvern
Worcestershire
WR14 3PS
United Kingdom

Notice to Readers

TDF is a portability technology and an architecture neutral format for expressing software applications which was developed by the United Kingdom's Defence Research Agency (DRA). DRA has demonstrated that the TDF technology can support ANSI C on MIPS®, Intel 386™, VAX™, SPARC™ and Motorola® 680x0.

Requests for information about TDF should be directed to:

Dr. N E Peeling
Defence Research Agency
St. Andrews Road
Malvern
Worcestershire
United Kingdom WR14 3PS

Tel. +44 684 895314
Fax +44 684 894303

Internet peeling%hermes.mod.uk@relay.mod.uk

While every attempt has been made to ensure the accuracy of all the information in this document the Defence Research Agency assumes no liability to any party for loss or damage, whether direct, indirect, incidental, or consequential, caused by errors or omissions or by statements of any kind in this document, or for the use of any product or system described herein. The reader shall bear the sole responsibility for his/her actions taken in reliance on the information in this document.

This document is for advanced information. It is not necessarily to be regarded as a final or official statement by the Defence Research Agency.

June 1993

Preface

This is Issue 2.1 of the TDF Specification.

Major changes to Issue 2.0 are described below.

The encoding is now specified in this document.

Changes to the notation

Optional arguments of constructions are written OPTION(*x*).

Those arguments which contain a variable number of elements are written LIST(*x*) or SLIST(*x*). The text has been changed so that the elements are no longer written with the *a*[*i*] notation, but are referred to as the elements of the list.

The previous specification described lists of compound values in the construction(for example see the *params_intro* argument of *make_proc* in Issue 2.0, Revision 1). In this issue an auxiliary SORT is introduced, the argument is said to be a list of elements of this SORT, and the definition of the auxiliary SORT is given separately.

Major changes

The new SORT ACCESS has been introduced to allow for extending the information that can be given about how a variable is used. The new SORT TRANSFER_MODE has been introduced to control some of the constructions which refer to memory.

ERROR_TREATMENTS have been modified and the rules for floating point errors changed.

A VARIETY representation is required to be a number of bits using twos-complement.

The ERROR_TREATMENT *wrap* replaces *ignore*, and is extended to signed VARIETIES.

All the results of a procedure have the same SHAPE.

The new UNITS, “version” and “linkinfo”, have been added.

All frequency information has been removed.

The encoding has been modified to allow more SORTS to be extended.

CONTENTS

1 Introduction

TDF is a porting technology and, as a result, it is a central part of a shrink-wrapping, distribution and installation technology. TDF has been chosen by the Open Software Foundation as the basis of its Architecture Neutral Distribution Format. It was developed by the United Kingdom's Defence Research Agency (DRA). DRA is working with Unix System Laboratories to commercialise the TDF technology. TDF is not UNIX specific, although most of the implementation has been done on UNIX.

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. TDF is intended for porting in this sense. It is designed so that a program in its TDF 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 TDF's “tokens”, and the TDF 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 TDF 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.

TDF 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 TDF constructions to be devised to represent them all. These TDF 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 TDF 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 TDF 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 TDF.

TDF is neutral with respect to machine architectures. In designing TDF, 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 TDF has constructions which are closely related to typical language features and it has an abstract model of memory. We expect 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 TDF 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.

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

2 Structure of TDF

Each piece of TDF program is classified as being of a particular SORT. Some pieces of TDF 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 TDF 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 TDF format. When TDF 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.

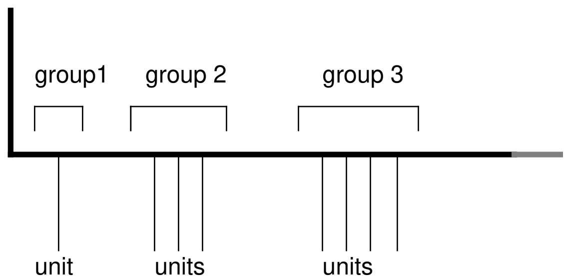
2.1 The Overall Structure

A separable piece of TDF is called a CAPSULE. A producer generates a CAPSULE; the TDF 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, without 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 TDF 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 TDF 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 TDF 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.

- *tld2* unit. Every capsule has exactly one tld2 unit. It gives information to the TDF linker about those items in the capsule which are visible externally.
- *version* unit. These units contain information about the versions of TDF used.
- *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.

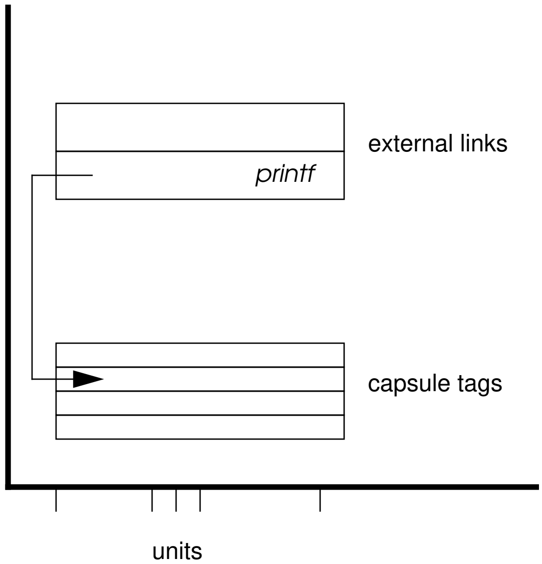
- *al_tagdef* units. These units give the definitions of alignment tags.

- *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.

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

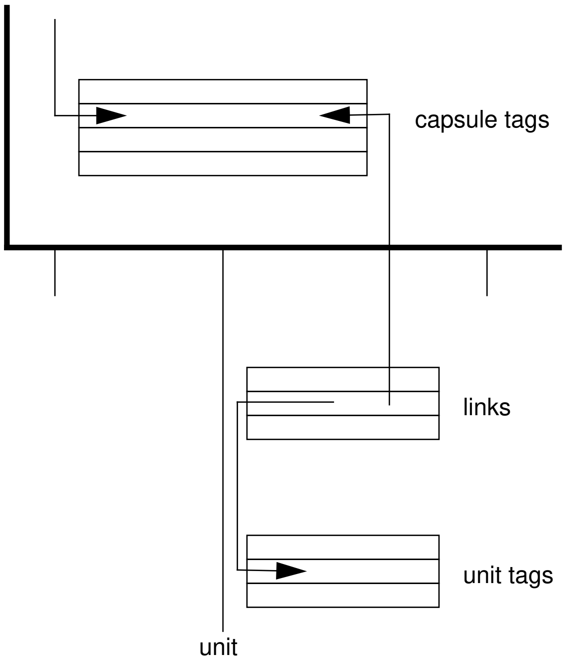
This organisation is imposed to help installers, by ensuring that the information needed to process a unit has been provided before that unit arrives. For example, the token definitions occur before any tag definition, so that, during translation, the tokens may be expanded as the tag definitions are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

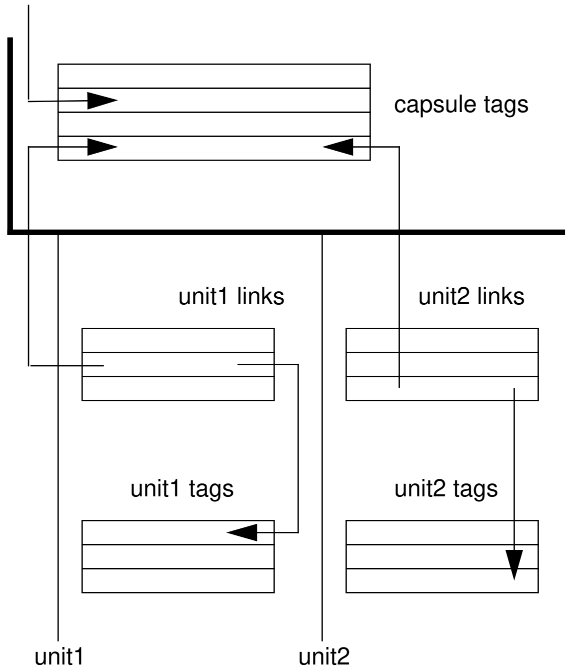
The tags used inside a unit have to be related to these capsule tags, so that they can be properly named. A similar mechanism is used, with a table of tags at the unit level, and links between these and the capsule level tags.



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

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 TDF 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 re-organises the units so that the correct order is achieved.

This can be done without looking into the interior of the units (except for the *tld2* 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 TDF; speed is required in the linking and installing processes, if necessary at the expense of time in the production of TDF.

2.2 Tokens

Tokens are used (applied) in the TDF 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 TDF 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 TDF of the definition with *par0*, *par1* and *par2* substituted by the actual parameters.

Only simple substitution is possible; though definitions can contain conditionals and uses of other tokens, they cannot be recursive and 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 TDF 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 TDF 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

TDF can be extended for two major reasons.

First, as part of the evolution of TDF, 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 TDF 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 TDF to permit special processing. TDF 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 TDF encoding.

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

1. The encoding of every SORT in TDF can be extended indefinitely (except for certain auxiliary SORTS). This mechanism should only be used for extending standard TDF to the next standard, since otherwise extensions made by different groups of people might conflict with each other. See “Extendable integer encoding” on page 62.

2. Basic TDF 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 TDF. The rules for linking this extra information are also laid down. See “make_capsule” on page 15.

   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.

   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 TDF 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 “BITSTREAM” on page 62.

3 Describing the Structure

The following examples show how TDF constructs are described in this document. The first is the construct *floating* (page 40):

fv: FLOATING_VARIETY
→ SHAPE

The construct's 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* (page 22) not only states that it has an EXP argument and an EXP result:

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

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

Some construct's arguments are optional. This is denoted as follows (from *apply_proc*, page 17):

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

var_param is an optional argument to the *apply_proc* construct shown above.

Some constructs take a varying number of arguments. *params* in the above construct is an example. These are denoted by LIST. There is a similar contruction, 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* (page 18) 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 a TDF-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, e.g. *exp_cond* on page 17). Secondly, standard tools might want to ignore embedded fragments of TDF adhering to some private standard. In these cases it is desirable for tools to be able to skip the irrelevant pieces of TDF. 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 (e.g.):

BITSTREAM FLOATING_VARIETY

This construct must have a FLOATING_VARIETY argument, but certain TDF-processing tools may benefit from being able to skip past the argument (which might itself be a very large piece of TDF) 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* etc.

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* etc.

The bit encodings are also specified in this document. The details are given in "The bit encoding of TDF" on page 61. 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 extendable or non-extendable. This is specified in the following form:

Is coding extendable? 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.

4 Installer Behaviour

4.1 Definition of terms

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

- “undefined”: means that 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 TDF construct, the installer may assume that P holds without having to check it. If, in fact, a producer has produced TDF for which P does not hold, the effect is undefined.
- “target-defined”: means that behaviour will be defined, but that 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 TDF. 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 TDF they are processing is well-formed, nor that undefined constructs are absent. If the TDF 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 optimization is permitted”.

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

5 Specification of TDF Constructs

5.1 ACCESS

Number of encoding bits: 3

Is coding extendable? Yes

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

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

> Comment: In this specification only the *visible* and *long_jump_access* properties are available. Possible extensions include information about aliasing.

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.

The notation *param_sorts*(*token_value*) is intended to mean the following. The token definition or token declaration for *token_value* gives the SORTS of its arguments in the SORTNAME component. The BITSTREAM in *token_args* consists of these SORTS in the given order. If no token declaration or definition exists in the CAPSULE, the BITSTREAM cannot be read.

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*.

5.1.4 long_jump_access

Encoding number: 6

→ 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.

5.1.5 standard_access

Encoding number: 4

→ ACCESS

An object qualified as having *standard_access* has normal (i.e. no special) access properties.

5.1.6 visible

Encoding number: 5

→ 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: al_tag

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

Comment: For example, the following equation has a circular form

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

and it or a similar equation might occur in TDF. But 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.

Simultaneous equations defining ALIGNMENTs can always be solved, since the only significant combining operation is *unite_alignments*, which is set union.

See “Circular types in languages” on page 57.

5.4 AL_TAGDEF_PROPS

Number of encoding bits: 0

Unit identification: al_tagdef

5.4.1 make_al_tagdefs

Encoding number: 0

no_labels: TDFINT
tds: SLIST(AL_TAGDEF)
→ AL_TAGDEF_PROPs

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: 3

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 "Memory Model" on page 56). 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*.

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 etc. 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 obtain_al_tag

Encoding number: 4

at: AL_TAG

→ ALIGNMENT

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

5.5.5 unite_alignments

Encoding number: 5

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*.

5.5.6 code_alignment

Encoding number: 6

→ ALIGNMENT

Delivers $\{code\}$, the ALIGNMENT of the POINTER produced by *make_local_lv*.

5.5.7 frame_alignment

Encoding number: 7

→ ALIGNMENT

Delivers the ALIGNMENT of POINTERS produced by *current_env*.

5.5.8 alloca_alignment

Encoding number: 8

→ ALIGNMENT

Delivers the ALIGNMENT of POINTERS produced from *local_alloc*.

5.5.9 var_param_alignment

Encoding number: 9

→ ALIGNMENT

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 TDF, but an installer may specify limits. See “Representing bitfields” on page 60 and “Permitted limits” on page 60.

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 TDF 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 “The TDF encoding” on page 62.

5.8 BOOL

Number of encoding bits: 3

Is coding extendable? Yes

A BOOL is a piece of TDF 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 “The TDF encoding” on page 62.

5.10 CAPSULE

Number of encoding bits: 0

A CAPSULE is an independent piece of TDF. There is only one construction, *make_capsule*.

5.10.1 make_capsule

Encoding number: 0

prop_names: SLIST(TDFIDENT)
capsule_linking: SLIST(CAPSULE_LINK)
external_linkage: SLIST(EXTERN_LINK)
groups: SLIST(GROUP)
→ CAPSULE

make_capsule brings together UNITs and linking and naming information. See “The Overall Structure” on page 3.

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 “PROPS” on page 38. 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.

5.11 CAPSULE_LINK

Number of encoding bits: 0

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

5.11.1 make_capsule_link

Encoding number: 0

sn: TDFIDENT
n: TDFINT
→ CAPSULE_LINK

n is the number of CAPSULE level linkable entities of the kind given by *sn*. *sn* corresponds to the linkable entity identification.

5.12 CASELIM

Number of encoding bits: 0

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

5.12.1 make_caselim

Encoding number: 0

branch: LABEL
lower: SIGNED_NAT
upper: SIGNED_NAT
→ CASELIM

Makes a triple of destination and limits. The *case* construction (page 18) uses a list of CASELIMS. If the control variable of the *case* lies between *lower* and *upper*, control passes to *branch*.

5.13 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.

Comment: It is expected that additional ERROR_TREATMENTS will be needed.

5.13.1 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.

5.13.2 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.

5.13.3 error_jump

Encoding number: 3

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.

Comment: This rule implies that a further construction is needed to guarantee that errors have been processed. This is not yet included.

5.13.4 wrap

Encoding number: 4

→ ERROR_TREATMENT

wrap is an ERROR_TREATMENT which will only be used in constructions delivering EXP INTEGER(v). The result will be evaluated and any bits in the result lying outside the representing VARIETY will be discarded (see “Representing integers” on page 58).

5.13.5 impossible

Encoding number: 5

→ ERROR_TREATMENT

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

5.14 EXP

Number of encoding bits: 7

Is coding extendable? Yes

EXPs are pieces of TDF 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 etc. of programming languages.

5.14.1 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.

5.14.2 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.

5.14.3 add_to_ptr

Encoding number: 3

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

arg1 and *arg2* are evaluated and added to produce the result. 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.

Comment: In the simple representation of POINTER arithmetic (see “Memory Model” on page 56) *add_to_ptr* is represented by addition. The constraint “*x* includes *y*” ensures that no padding has to be inserted in this case.

5.14.4 and

Encoding number: 4

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 *and* of the two values in the representing VARIETY. The result is delivered with the same SHAPE as the arguments.

See “Representing integers” on page 58.

5.14.5 apply_proc

Encoding number: 5

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 in-lined. 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.

The constituents of *params* will have the SHAPEs specified by the formal parameters of the procedure which is being called. The list in *params* will correspond one-to-one in the same order to the corresponding list of formal parameters in *make_proc*.

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 54.

5.14.6 assign

Encoding number: 6

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

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

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

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 "Overlapping" on page 58 and "Incomplete assignment" on page 58.

Comment: The constraint "*x* will include *alignment*(*y*)" ensures in the simple memory model (page 56) that no change is needed to the POINTER.

5.14.7 assign_with_mode

Encoding number: 7

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 (q.v.).

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

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

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 "Overlapping" on page 58 and "Incomplete assignment" on page 58.

5.14.8 case

Encoding number: 8

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. 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.

5.14.9 change_bitfield_to_int

Encoding number: 9

x: VARIETY
arg1: EXP BITFIELD *v*
→ EXP INTEGER *x*

arg1 when evaluated gives a value of SHAPE BITFIELD *v*, which is converted to INTEGER *x* and delivered.

All the values belonging to BITFIELD *v* will also belong to VARIETY *v*.

5.14.10 change_floating_variety

Encoding number: 10

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

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

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

See "Floating point errors" on page 59.

5.14.11 change_int_to_bitfield

Encoding number: 11

x: BITFIELD_VARIETY
arg1: EXP INTEGER *v*
→ EXP BITFIELD *x*

arg1 when evaluated gives a value of SHAPE INTEGER *v*, which is converted to BITFIELD *x* and delivered.

All the values belonging to VARIETY *v* will also belong to BITFIELD *x*.

5.14.12 change_variety

Encoding number: 12

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*.

5.14.13 component

Encoding number: 13

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 "Constant evaluation" on page 53).

5.14.14 concat_nof

Encoding number: 14

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*.

5.14.15 conditional

Encoding number: 15

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 "Least Upper Bound" on page 60.

5.14.16 contents

Encoding number: 16

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

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

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

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.

> Comment: The constraint "*x* will include *alignment*(s)" ensures in the simple memory model (page 56) that no change is needed to the POINTER.

5.14.17 contents_with_mode

Encoding number: 17

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

A value of SHAPE *s* 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 (q.v.).

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

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

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.

5.14.18 current_env

Encoding number: 18

→ EXP POINTER(*frame_alignment*)

A value of SHAPE POINTER(*frame_alignment*) 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 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 "Original pointers" on page 58).

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

5.14.19 div1

Encoding number: 19

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 "Division and modulus" on page 53 for the definitions of D1, D2, M1 and M2.

5.14.20 div2

Encoding number: 20

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 "Division and modulus" on page 53 for the definitions of D1, D2, M1 and M2.

5.14.21 env_offset

Encoding number: 21

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

t will be the tag of a *variable*, *identify* or procedure parameter with the *visible* property.

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

If it is 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).

The ALIGNMENT, *frame_alignment*, includes the set union of all the ALIGNMENTs which can be produced by *alignment* from any SHAPE.

See "Special alignments" on page 57.

5.14.22 fail_installer

Encoding number: 22

message: TDFSTRING(k, n)
→ EXP BOTTOM

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

5.14.23 float_int

Encoding number: 23

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 there is a floating point error it is handled by *flpt_err*. See "Floating point errors" on page 59.

5.14.24 floating_abs

Encoding number: 24

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.

If there is a floating point error it is handled by *flpt_err*. See "Floating point errors" on page 59.

5.14.25 floating_div

Encoding number: 25

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 “Floating point errors” on page 59.

5.14.26 floating_minus

Encoding number: 26

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 “Floating point errors” on page 59.

5.14.27 floating_mult

Encoding number: 27

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 “Floating point errors” on page 59.

> Comment: Note that 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*.

5.14.28 floating_negate

Encoding number: 28

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.

If there is a floating point error it is handled by *flpt_err*. See “Floating point errors” on page 59.

5.14.29 floating_plus

Encoding number: 29

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 “Floating point errors” on page 59

> Comment: Note that 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*.

5.14.30 floating_test

Encoding number: 30

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

If there is a floating point error it is handled by *flpt_err*. See “Floating point errors” on page 59.

5.14.31 goto

Encoding number: 31

dest: LABEL
→ EXP BOTTOM

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

5.14.32 goto_local_lv

Encoding number: 32

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.

5.14.33 identify

Encoding number: 33

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 “sequence” on page 32.

5.14.34 integer_test

Encoding number: 34

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*.

5.14.35 labelled

Encoding number: 35

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* 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 “Least Upper Bound” on page 60.

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*.

5.14.36 last_local

Encoding number: 36

x: EXP OFFSET(*x*, *y*)
→ 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 “Special alignments” on page 57.

5.14.37 local_alloc

Encoding number: 37

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.

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. Any use of the pointer thereafter is undefined.

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 the next use of *local_alloc* will give an equal POINTER (in the sense of *pointer_test*).

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

5.14.38 local_free

Encoding number: 38

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 *x*.

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

5.14.39 local_free_all

Encoding number: 39

→ 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.

5.14.40 long_jump

Encoding number: 41

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

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.

5.14.41 make_compound

Encoding number: 42

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

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 “offset_pad” on page 29.

The values *x*[2 * *k* - 1] will be such that the components when in place do not overlap. See “Overlapping” on page 58.

5.14.42 make_floating

Encoding number: 43

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

mantissa will be a STRING of ASCII characters, each of which is either ASCII's point symbol or is greater than or equal to ASCII's zero symbol. Those characters, *c*, which lie between 48 and 63 will represent the digit *c*-48.

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

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 etc.

5.14.43 make_int

Encoding number: 44

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

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

5.14.44 make_local_lv

Encoding number: 45

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*.

5.14.45 make_nof

Encoding number: 46

arg1: LIST(EXP)
→ EXP NOF(*n*, *s*)

Creates an array of *n* values of SHAPE *s*, containing the given values produced by evaluating the members of *arg1* in the same order as they occur in the list.

The list may be empty.

5.14.46 make_nof_int

Encoding number: 47

v: VARIETY
str: TDFSTRING(*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.

5.14.47 make_null_local_lv

Encoding number: 48

→ 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.

5.14.48 make_null_proc

Encoding number: 49

→ 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.

5.14.49 make_null_ptr

Encoding number: 50

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*).

5.14.50 make_proc

Encoding number: 51

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* (if present) 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. See “Special alignments” on page 57. 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.

When the procedure is called (by *apply_proc*) the actual parameters supplied in the call will correspond one-to-one with the formal parameters. These actual parameters will act as initialising values as if the actual and formal parameters formed a variable declaration with *body* as its body.

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 54.

5.14.51 make_top

Encoding number: 52

→ EXP TOP

make_top delivers a value of SHAPE TOP (i.e. void).

5.14.52 make_value

Encoding number: 53

s: SHAPE
→ 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. But if it is used for arithmetic or as a POINTER for taking *contents* or *add_to_ptr* etc. the effect is undefined.

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

The SHAPE *s* will not be BOTTOM.

5.14.53 minus

Encoding number: 54

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*.

5.14.54 move_some

Encoding number: 55

md: TRANSFER_MODE
arg1: EXP POINTER *x*
arg2: EXP POINTER *y*
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 (q.v.).

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

sz will be a non-negative OFFSET, see “offset_pad” on page 29.

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 “Overlapping” on page 58.

5.14.55 mult

Encoding number: 56

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*.

5.14.56 n_copies

Encoding number: 57

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.

5.14.57 negate

Encoding number: 58

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*.

5.14.58 not

Encoding number: 59

arg1: EXP INTEGER(v)
→ EXP INTEGER(V)

The argument is evaluated producing an integer value, of VARIETY, v. The result is the bitwise *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 “Representing integers” on page 58.

5.14.59 obtain_tag

Encoding number: 60

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.

5.14.60 offset_add

Encoding number: 61

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.

> Comment: The effect of the constraint “y will include z” is that, in the simple representation of pointer arithmetic (page 56), this operation can be represented by addition.

> Comment: *offset_add* can lose information,so that *offset_subtract* does not have the usual relation with it.

5.14.61 offset_div

Encoding number: 62

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. The division will be exact. That is, if *arg2* has no side effects:-

arg1= *offset_mult*(*arg2*, *offset_div*(v, *arg1*, *arg2*))

If the result cannot be expressed in the VARIETY being used to represent v the effect is undefined.

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

5.14.62 offset_div_by_int

Encoding number: 63

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 division will be exact. That is, if arg2 has no side effects:-

arg1 = *offset_mult*(*offset_div_by_int*(*arg1*, *arg2*), *arg2*)

The lower bound of v will be zero.

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

5.14.63 offset_max

Encoding number: 64

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 "Comparison of pointers and offsets" on page 57.

Comment: In the simple memory model (page 56) 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.

5.14.64 offset_mult

Encoding number: 65

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 n* times to *offset_zero*(x).

5.14.65 offset_negate

Encoding number: 66

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

The inverse of the argument is delivered.

Comment: In the simple memory model (page 56) this can be represented by negate.

5.14.66 offset_pad

Encoding number: 67

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*.

Comment: In the simple memory model (page 56) 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.

5.14.67 offset_subtract

Encoding number: 69

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 and y will include z, by the constraints on OFFSETS.

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

5.14.68 offset_test

Encoding number: 70

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*.

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

Comment: In the simple memory model (page 56) this can be represented by *integer_test*.

5.14.69 offset_zero

Encoding number: 71

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*).

5.14.70 or

Encoding number: 72

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 *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 “Representing integers” on page 58.

5.14.71 plus

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 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*.

5.14.72 pointer_test

Encoding number: 74

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*.

The effect of this construction is the same as:-

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

Comment: In the simple memory model (page 56) this construction can be represented by integer_test.

5.14.73 proc_test

Encoding number: 75

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*.

Two PROCs are equal if they are identical or if they were both made with *make_null_proc*. Otherwise they are unequal.

5.14.74 rem1

Encoding number: 76

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 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 suitable masking and rem1 by a power of two yield equally good code.

See “Division and modulus” on page 53 for the definitions of D1, D2, M1 and M2.

5.14.75 rem2

Encoding number: 77

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

See “Division and modulus” on page 53 for the definitions of D1, D2, M1 and M2.

5.14.76 repeat

Encoding number: 78

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.

5.14.77 return

Encoding number: 79

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* will have *arg1* with the same SHAPE.

5.14.78 round_with_mode

Encoding number: 80

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

arg 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 there is a floating point error it is handled by *flpt_err*. See “Floating point errors” on page 59.

5.14.79 sequence

Encoding number: 81

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 the *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 that this 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.

5.14.80 shape_offset

Encoding number: 82

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*.

5.14.81 shift_left

Encoding number: 83

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*.

If *b* is greater than or equal to the minimum number of bits needed to represent *v* in twos-complement, then the effect is undefined.

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

5.14.82 shift_right

Encoding number: 84

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.

5.14.83 subtract_ptrs

Encoding number: 85

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*.

5.14.84 variable

Encoding number: 86

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 “sequence” on page 32.

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

5.14.85 xor

Encoding number: 87

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 *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 “Representing integers” on page 58.

5.15 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.

5.15.1 string_extern

Encoding number: 1

byte_align
s: TDFIDENT(*n*)
→ EXTERNAL

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

5.15.2 unique_extern

Encoding number: 2

byte_align

u: UNIQUE

→ EXTERNAL

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

5.16 EXTERN_LINK

Number of encoding bits: 0

An auxiliary SORT providing a list of LINKEXTERN.

5.16.1 make_extern_link

Encoding number: 0

el: SLIST(LINKEXTERN)

→ 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.

5.17 FLOATING_VARIETY

Number of encoding bits: 2

Is coding extendable? Yes

These describe kinds of floating point number.

5.17.1 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.

5.17.2 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.

5.17.3 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 2.

> Comment: It is expected that in future specifications *base* will be allowed to take more values.

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.

A TDF 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.

5.18 GROUP

Number of encoding bits: 0

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

5.18.1 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*.

5.19 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.

5.19.1 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.

5.19.2 make_label

Encoding number: 1

labelno: TDFINT
→ LABEL

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

5.20 LINK

Number of encoding bits: 0

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

5.20.1 make_link

Encoding number: 0

unit_name: TDFINT
capsule_name: TDFINT
→ 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 TDF builder in the course of resolving sharing and name clashes when constructing a composite CAPSULE.

5.21 LINKEXTERN

Number of encoding bits: 0

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.

5.21.1 make_linkextern

Encoding number: 0

internal: TDFINT
ext: EXTERNAL
→ 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.

5.22 LINKS

Number of encoding bits: 0

5.22.1 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.

5.23 NAT

Number of encoding bits: 3

Is coding extendable? Yes

These are non-negative integers of unlimited size.

5.23.1 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.

5.23.2 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.

5.23.3 computed_nat

Encoding number: 3

arg: EXP INTEGER(*var_limits*(0,*h*))
→ NAT

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

5.23.4 make_nat

Encoding number: 4

n: TDFINT
→ NAT

n is a non-negative integer of unbounded magnitude.

5.24 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 (e.g. in IEEE floating point).

5.24.1 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.

5.24.2 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.

5.24.3 equal

Encoding number: 3

→ NTEST

Signifies "equal" test.

5.24.4 greater_than

Encoding number: 4

→ NTEST

Signifies "greater than" test.

5.24.5 greater_than_or_equal

Encoding number: 5

→ NTEST

Signifies "greater than or equal" test.

5.24.6 less_than

Encoding number: 6

→ NTEST

Signifies "less than" test.

5.24.7 less_than_or_equal

Encoding number: 7

→ NTEST

Signifies "less than or equal" test.

5.24.8 not_equal

Encoding number: 8

→ NTEST

Signifies "not equal" test.

5.24.9 not_greater_than

Encoding number: 9

→ NTEST

Signifies "not greater than" test.

5.24.10 not_greater_than_or_equal

Encoding number: 10

→ NTEST

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

5.24.11 not_less_than

Encoding number: 11

→ NTEST

Signifies "not less than" test.

5.24.12 not_less_than_or_equal

Encoding number: 12

→ NTEST

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

5.24.13 less_than_or_greater_than

Encoding number: 13

→ NTEST

Signifies "less than or greater than" test.

5.24.14 not_less_than_and_not_greater_than

Encoding number: 14

→ NTEST

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

5.24.15 comparable

Encoding number: 15

→ NTEST

Signifies "comparable" test.

5.24.16 not_comparable

Encoding number: 16

→ NTEST

Signifies "not comparable" test.

5.25 PROPS

A PROPS is an assemblage of program information. This standard offers five ways of constructing a PROPS — i.e. it defines five 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 TDF program;
- definitions of the pieces of TDF program for which TOKENs stand.

The standard can be extended by the definition of new kinds of PROPS information and new PROPS constructs for expressing them; and 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 a particular GROUP have the same SORT, given by the following mapping for the SORTS defined in this specification. The mapping is extendable.

- *tokdec* corresponds to TOKDEC_PROPS
- *tokdef* corresponds to TOKDEF_PROPS
- *al_tagdef* corresponds to AL_TAGDEF_-PROPS
- *tagdec* corresponds to TAGDEC_PROPS
- *tagdef* corresponds to TAGDEF_PROPS

In addition there is a *tld2* UNIT, see “The TDF encoding” on page 62.

5.26 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.

5.26.1 rounding_mode_apply_token

Encoding number: 1

token_value: TOKEN
token_args: BITSTREAM
param_sorts(token_value)
→ ROUNDING_MODE

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.

5.26.2 rounding_mode_cond

Encoding number: 2

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

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 round_as_state

Encoding number: 3

→ ROUNDING_MODE

Round as specified by the current state of the machine.

5.26.4 to_nearest

Encoding number: 4

→ ROUNDING_MODE

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

5.26.5 toward_larger

Encoding number: 5

→ ROUNDING_MODE

Signifies rounding toward next largest.

5.26.6 toward_smaller

Encoding number: 6

→ ROUNDING_MODE

Signifies rounding toward next smallest.

5.26.7 toward_zero

Encoding number: 7

→ ROUNDING_MODE

Signifies rounding toward zero.

5.27 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.

5.27.1 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.

5.27.2 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.

5.27.3 bitfield

Encoding number: 3

bf_var: BITFIELD_VARIETY
→ SHAPE

A BITFIELD is used to represent a pattern of bits which may be packed. Installers shall represent these bits compactly; there shall be no intervening bits.

A BITFIELD_VARIETY specifies the number of bits and whether they are considered to be signed. See “Representing bitfields” on page 60.

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 “Permitted limits” on page 60.

5.27.4 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.

5.27.5 compound

Encoding number: 5

sz: EXP OFFSET(x, { })
→ SHAPE

The SHAPE constructor COMPOUND describes cartesian products and unions.

sz will evaluate to a constant, non-negative OFFSET (see “offset_pad” on page 29). The resulting SHAPE describes a value whose size is given by *sz*.

5.27.6 floating

Encoding number: 6

fv: FLOATING_VARIETY
→ SHAPE

Most of the floating point arithmetic operations, *floating_plus*, *floating_minus* etc., 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 “Representing floating point” on page 59.

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 “Permitted limits” on page 60.

5.27.7 integer

Encoding number: 7

var: VARIETY
→ SHAPE

The different kinds of INTEGER are distinguished by having different VARIETIEs. A fundamental VARIETY (not a TOKEN or conditional) is represented by two SIGNED_NATs, 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 TDF does likewise. Because TDF is completely architecture neutral and makes no assumptions about word length, this means that the VARIETIES of the two arguments must be identical. An example illustrates this. A piece of TDF 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 VARIETIEs 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 “Permitted limits” on page 60.

For the representation of integers see “Representing integers” on page 58.

5.27.8 nof

Encoding number: 9

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*. *n* may be zero.

5.27.9 offset

Encoding number: 10

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 runtime 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.

The set *arg1* will include the set *arg2*.

See "Memory Model" on page 56.

5.27.10 pointer

Encoding number: 11

arg: ALIGNMENT
→ SHAPE

A POINTER is a value which points to space allocated in a computer's memory. The POINTER constructor takes an ALIGNMENT argument. See "Memory Model" on page 56

5.27.11 proc

Encoding number: 12

→ SHAPE

PROC is the SHAPE which describes pieces of program.

5.27.12 top

Encoding number: 13

→ 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.

5.28 SIGNED_NAT

Number of encoding bits: 3

Is coding extendable? Yes

These are positive or negative integers of unbounded size.

5.28.1 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.

5.28.2 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.

5.28.3 computed_signed_nat

Encoding number: 3

arg: EXP INTEGER(*v*)
→ SIGNED_NAT

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

5.28.4 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 iff *neg* is true.

5.28.5 snat_from_nat

Encoding number: 5

neg: BOOL
n: NAT
→ SIGNED_NAT

The result is negated iff *neg* is true.

5.29 SORTNAME

Number of encoding bits: 5

Is coding extendable? Yes

These are the names of the SORTs which can be parameters of TOKEN definitions.

5.29.1 access

Encoding number: 18

→ SORTNAME

5.29.2 al_tag

Encoding number: 1

→ SORTNAME

5.29.3 alignment_sort

Encoding number: 2

→ SORTNAME

5.29.4 bitfield_variety

Encoding number: 3

→ SORTNAME

5.29.5 bool

Encoding number: 4

→ SORTNAME

5.29.6 error_treatment

Encoding number: 5

→ SORTNAME

5.29.7 exp

Encoding number: 6

→ SORTNAME

The SORT of EXP.

5.29.8 floating_variety

Encoding number: 7

→ SORTNAME

5.29.9 foreign_sort

Encoding number: 8

foreign_name: TDFSTRING
→ SORTNAME

This SORT enables unanticipated kinds of information to be placed in TDF.

5.29.10 label

Encoding number: 9

→ SORTNAME

5.29.11 nat

Encoding number: 10

→ SORTNAME

5.29.12 ntest

Encoding number: 11

→ SORTNAME

5.29.13 rounding_mode

Encoding number: 12

→ SORTNAME

5.29.14 shape

Encoding number: 13

→ SORTNAME

5.29.15 signed_nat

Encoding number: 14

→ SORTNAME

5.29.16 tag

Encoding number: 15

→ SORTNAME

The SORT of TAG.

5.29.17 transfer_mode

Encoding number: 19

→ SORTNAME

5.29.18 token

Encoding number: 16

params: LIST(SORTNAME)
result: SORTNAME
→ SORTNAME

The SORTNAME of a TOKEN. Note that it can have tokens as parameters.

5.29.19 variety

Encoding number: 17

→ SORTNAME

5.30 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.

5.30.1 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.

5.30.2 make_tag

Encoding number: 1

tagno: TDFINT

→ TAG *x*

make_tag produces a TAG identified by *tagno*.

5.31 TAGACC

Number of encoding bits: 0

Constructs a pair of a TAG and an OPTION(ACCESS) for use in *make_proc*.

5.31.1 make_tagacc

Encoding number: 0

tg: TAG POINTER
var_param_alignment
acc: OPTION(ACCESS)

→ TAGACC

Constructs the pair for *make_proc*.

5.32 TAGDEC

Number of encoding bits: 2

Is coding extendable? Yes

A TAGDEC declares a TAG for incorporation into a TAGDEC_PROPS.

5.32.1 make_id_tagdec

Encoding number: 1

t_intro: TDFINT
acc: OPTION(ACCESS)
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.

5.32.2 make_var_tagdec

Encoding number: 2

t_intro: TDFINT
acc: OPTION(ACCESS)
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.

5.32.3 common_tagdec

Encoding number: 3

t_intro: TDFINT
acc: OPTION(ACCESS)
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.

5.33 TAGDEC_PROPS

Number of encoding bits: 0

Unit identification: tagdec

5.33.1 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.

5.34 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. Every TAG defined in a TAGDEF will be declared in a TAGDEC.

5.34.1 make_id_tagdef

Encoding number: 1

t: TDFINT
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.

t will be declared in the CAPSULE using *make_id_tagdec*.

There will not be more than one TAGDEF defining *t* in a CAPSULE.

5.34.2 make_var_tagdef

Encoding number: 2

t: TDFINT
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.

e will be a constant which can be evaluated at load_time.

t will be declared in the CAPSULE using *make_var_tagdec*.

There will not be more than one TAGDEF defining *t* in a CAPSULE.

5.34.3 common_tagdef

Encoding number: 3

t: TDFINT
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.

e will be a constant evaluable at load_time.

t will be declared in the CAPSULE using *common_tagdec*. Let the maximum SHAPE of these (see “common_tagdec” on page 44) 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.

5.35 TAGDEF_PROPS

Number of encoding bits: 0

Unit identification: tagdef

5.35.1 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.

5.36 TAGSHACC

Number of encoding bits: 0

5.36.1 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*.

5.37 TDFBOOL

A TDFBOOL is the TDF encoding of a boolean.

5.38 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 BITSTREAM.

5.39 TDFINT

A TDFINT is the TDF encoding of an unbounded unsigned integer constant.

5.40 TDFSTRING

A TDFSTRING(k, n) encodes a sequence of n unsigned integers of size k bits.

5.41 TOKDEC

Number of encoding bits: 1

Is coding extendable? Yes

A TOKDEC declares a TOKEN for incorporation into a UNIT.

5.41.1 make_tokdec

Encoding number: 1

tok: TDFINT
s: SORTNAME
→ TOKDEC

The sort of the token *tok* is declared to be *s*.

5.42 TOKDEC_PROPS

Number of encoding bits: 0

Unit identification: tokdec

5.42.1 make_tokdecs

Encoding number: 0

tds: SLIST(TOKDEC)
→ TOKDEC_PROPs

tds is a list of TOKDECs which gives the sorts associated with TOKENs.

5.43 TOKDEF

Number of encoding bits: 1

Is coding extendable? Yes

A TOKDEF gives the definition of a TOKEN for incorporation into a TOKDEF_PROPS.

5.43.1 make_tokdef

Encoding number: 1

tok: TDFINT
def: BITSTREAM TOKEN_DEFN
→ TOKDEF

A TOKDEF is constructed which defines the TOKEN *tok* to stand for the fragment of TDF, *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 “token_definition” on page 47.

At the application of this TOKEN actual pieces of TDF having SORT *sn*[*i*] are supplied to correspond to the *tk*[*i*]. The application denotes the piece of TDF obtained by substituting these actual parameters for the corresponding TOKENs within *body*.

The body need not be re-evaluated if there are no parameters.

TOKEN definitions will not be recursive, that is, they will not be applied in *body*, or in any *token* application invoked in applying the *body*.

5.44 TOKDEF_PROPS

Number of encoding bits: 0

Unit identification: tokdef

5.44.1 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.

5.45 TOKEN

Number of encoding bits: 2

Is coding extendable? Yes

Linkable entity identification: token

These are used to stand for functions evaluated at installation time. They are represented by TDFINTs.

5.45.1 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.

5.45.2 make_tok

Encoding number: 2

tokno: TDFINT
→ TOKEN

make_tok constructs a TOKEN identified by *tokno*.

5.45.3 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.

5.46 TOKEN_DEFN

Number of encoding bits: 1

Is coding extendable? Yes

An auxiliary SORT used in *make_tokdef* and *use_tokdef*.

5.46.1 token_definition

Encoding number: 1

result_sort: SORTNAME
tok_params: LIST(TOKFORMALS)
body: *result_sort*
→ TOKEN_DEFN

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*.

5.47 TOKFORMALS

Number of encoding bits: 0

5.47.1 make_tokformals

Encoding number: 0

sn: SORTNAME
tk: TDFINT
→ TOKFORMALS

An auxiliary construction to make up the elements of the lists in *token_defn*.

5.48 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*.

5.48.1 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.

The notation *param_sorts*(*token_value*) is intended to mean the following. The token definition or token declaration for *token_value* gives the SORTS of its arguments in the SORTNAME component. The BITSTREAM in *token_args* consists of these SORTS in the given order. If no token declaration or definition exists in the CAPSULE, the BITSTREAM cannot be read.

5.48.2 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.

5.48.3 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.

5.48.4 overlap

Encoding number: 5

→ 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.

If *overlap* is used to qualify a *contents_with_mode* or an *assign_with_mode* for which *arg2* is neither *contents* nor *contents_with_mode*, then the qualification is as if *standard_transfer_mode* was used.

See “Overlapping” on page 58.

5.48.5 standard_transfer_mode

Encoding number: 4

→ TRANSFER_MODE

This TRANSFER_MODE implies no special properties.

5.48.6 volatile

Encoding number: 6

→ TRANSFER_MODE

If volatile is used to qualify a construction it shall not be optimised away.

Comment: This is intended to implement ANSI C’s volatile construction.

5.49 UNIQUE

Number of encoding bits: 0

These are used to provide world-wide unique names for TOKENs and TAGs.

This implies a registry for allocating UNIQUE values.

5.49.1 make_unique

Encoding number: 0

text: SLIST(TDFIDENT)
→ UNIQUE

Two UNIQUE values are equal iff they were constructed with equal arguments.

5.50 UNIT

Number of encoding bits: 0

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.

5.50.1 make_unit

Encoding number: 0

local_vars: SLIST(TDFINT)
lks: SLIST(LINKS)
properties: BYTESTREAM
→ 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 "make_capsule" on page 15.

The length of *lks* will be either 0 or equal to the length of *capsule_linking* in *make_capsule*.

5.51 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 TDF, but an installer may specify limits. See "Permitted limits" on page 60.

For the representation of integers see "Representing integers" on page 58.

5.51.1 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.

5.51.2 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.

5.51.3 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).

5.52 VERSION_PROPS

Number of encoding bits: 0

Unit identification: versions

This UNIT gives information about version numbers.

Comment: Initially this UNIT contains only the version of TDF being used

5.52.1 make_versions

Encoding number: 0

version_info: SLIST(VERSION)
→ VERSION_PROPS

Contains version information.

5.53 VERSION

Number of encoding bits: 1

Is coding extendable? Yes

5.53.1 make_version

Encoding number: 1

major_version: TDFINT
minor_version: TDFINT
→ VERSION

The major and minor version numbers of the TDF used. An increase in minor version number means an extension of facilities, an increase in major version number means an incompatible change. TDF with the same major number but a lower minor number than the installer shall install correctly.

For TDF conforming to this specification the major number will be 2 and the minor number will be 1.

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

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, i.e. static) to identify the definition for subsequent linking. ghkjdfl gdfhg fkdhhjg hdfkjg hdfjk ghfsdkj ghjdfk gd ghfjkd ghfdkj ghjkdf hgkjfds hgjkfdhg jkdf ghfk dhgfkdj fhgjdk hfjkds fhjdks fhjksd fhfsda

Comment: 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.

7 Notes

7.1 Binding

The following constructions introduce TAGs:- *identify*, *variable*, *make_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* 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* construction occurs in the body of another *make_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 *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 construction 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 and LABELs 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

TDF 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 a TDF 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 TDF 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.

But this raises the question, who chooses the fixed representation and the unique TOKENs and their specification? Clearly TDF provides a mechanism for performing the standardisation without itself defining a standard.

Here TDF 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*, or defined with *make_var_tagdef* or *common_tagdef* and neither *contents* nor *assign* will be applied to it nor to any POINTER derived from it.

It may not contain any of the following constructions:- *apply_proc*, *assign_to_volatile*, *contents_of_volatile*, *current_env*, *goto_local_lv*, *make_local_lv*, *move_some*, *repeat*, *round_as_state*.

If it contains *labelled* there will only be jumps to the LABELS from within *starter*, not from within any of the *places*.

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:

$p \text{ D1 } q = n$

where

$p = n*q + (p \text{ M1 } q)$

$\text{sign}(p \text{ M1 } q) = \text{sign}(q)$

$0 <= |p \text{ M1 } q| < |q|$

Class 2:

$p \text{ D2 } q = n$

where

$p = n*q + (p \text{ M2 } q)$

$\text{sign}(p \text{ M2 } q) = \text{sign}(p)$

$0 <= |p \text{ M2 } q| < |q|$

7.5 Equality of EXPs

A definition of equality of EXPs would be a considerable part of a formal specification of TDF, 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*, 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

TDF 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. But 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). TDF allows it by means of the POINTER({*code*}). TDF 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: 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. TDF 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 TDF. 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 TDF.

So TDF 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.

7.9 Procedures

The *params* of an *apply_proc* and the *params_intro* of the *make_proc* which created the procedure being applied will correspond one-to-one in respect of SHAPE.

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*.

All uses of *return* in a procedure will return values of the same SHAPE, and this will be the *result_shape* specified in all uses of *apply_proc* calling the procedure.

Any SHAPE is permitted as the *result_shape* in an *apply_proc*.

7.10 Frames

TDF 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. TDF 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 TDF 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.

The ALIGNMENT of the POINTER delivered by *current_env* is *frame_alignment*. This shall include the set union of all the ALIGNMENTs which can be produced by *alignment* from any SHAPE. Note that this does not say that *frame_alignment* is that set union. 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*) 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*.

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.

> Comment: 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 VARIETYs, all BITFIELD_VARIETIEs, all FLOATING_VARIETYs, *proc*, *code*, *pointer* and *offset*. There are also three special ALIGNMENTS, *frame_alignment*, *alloca_alignment* and *var_param_alignment*.

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 TDF. 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 starting POINTER and that of the end POINTER. The ALIGNMENT set of the first must include the ALIGNMENT set of the second.

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(>= , *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 clear a machine with bit addressing is hypothesized. 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 SHAPE *s*.

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)
$\rightarrow$ 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)
$\rightarrow$ 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 by the note in “offset_test” on page 30.

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 TDF 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 four special ALIGNMENTS. One of these is *code_alignment*, the ALIGNMENT of the POINTER delivered by *make_local_lv*.

The other three special ALIGNMENTS are *frame_alignment*, *alloca_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. But 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 “Equality of ALIGNMENTS” on page 54).

Also notice that POINTER(*frame_alignment*) and OFFSET(*frame_alignment*, x) etc. 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(*frame_alignment*))={*pointer*}

7.13.5 Atomic assignment

At least one VARIETY shall exist such that *assign* and *assign_to_volatile* 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.

> Comment: Note that 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*, *case*, *goto*, *goto_local_lv*, *long_jump*, *return*, the *test* constructions and all instructions containing the *error_jump* ERROR_TREATMENT.

The order of evaluation of the components of other constructions is as follows. The components may be evaluated in any order and with their components - down to the TDF 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 are to be taken (“error_jump” on page 16) 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. Hence 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 indicating 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 TDF world (such as *calloc*) it is part of their interface with TDF 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 “current_-env” on page 20).

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 independant 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 SHAPE of the value being transferred is n INTEGER, a FLOATING, a BITFIELD, a POINTER, an OFFSET or a PROC, then the destination shall be unchanged.

If the SHAPE of the value being transferred is COMPOUND or NOF, then the contents of the destination are undefined.

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^n-1 for some *n*. *n* is called the number of bits in the representation.

If the lower bound of the VARIETY is negative the representing range shall be from -2^n to 2^n-1 for some *n*. *n*+1 is called the number of bits in the representation.

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 32 bits, signed or unsigned.

> Comment: 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*, properly inlude those of a VARIETY, *w*, the representing VARIETY for *v* shall include or be equal to the representing VARIETY for *w*.

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 TDF 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.

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_VARIETIES shall be implemented by a representation which has at least the properties specified.

Installers may limit the size of FLOATNG_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.

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 1023.

Operations are performed and overflows detected in the representing FLOATING_VARIETY.

7.21 Floating point errors

The only permitted ERROR_TREATMENTS for operations delivering FLOATING_VARIETIES are *impossible* and *error_jump*.

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 TDF (at present).

If an *error_jump* is taken as the result of a floating point error the operations to test what kind of error it was are outside the TDF 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*, *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 Representing bitfields

BITFIELD_VARIETIES 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_VARIETIES. Such a limit shall be not less than 32 bits, signed or unsigned.

> Comment: It is intended that there should be no upper limit allowed at some future date.

7.24 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.

7.25 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.

7.26 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 TDF specification takes the attitude that the use of read-only areas is a property of how TDF is used - a part of the installation process - and there should not be TDF constructions to say that some values in a CAPSULE are read-only. Such constructions could not be sufficiently general.

8 The bit encoding of TDF

This is a description of the encoding used for TDF. 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 TDF is encoded using the previously defined concepts.

8.1 The Basic Encoding

TDF 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.

TDF 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 (i.e. recursively).

8.2 Fundamental encodings

This section describes the encoding of TDFINT, TDFBOOL, TDFSTRING, TDFIDENT, BITSTREAM, 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. Thirdly 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 TDF.

The encoding consists of, first a TDFINT giving the number of bits, second a TDFINT giving the number of integers, which may be zero. Thirdly it contains *n* *k*-bit integers. After each integer, 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 TDF 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 BITSTREAM 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 a TDF construction without reading through it. BYTESTREAM provides a means of doing this while remaining byte aligned, so facilitating copying the TDF. 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 TDF encoding

The descriptions of SORTS and constructors contain encoding information which is interpreted as follows to define the TDF encoding.

1. A TDF 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*) — e.g. SLIST(UNIT) — the encoding consists of a TDFINT, *n*, followed by *n* encodings of *x*.

5. For the classes which are named LIST(*x*) — e.g. LIST(EXP) — the encoding consists of a 1-bit integer which will be 0, follwed 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, i.e. 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 *tld2* UNIT is encoded specially. It is always the first UNIT in a Capsule. If *ntk* is the number of CAPSULE token external links and *ntg* is the number of CAPSULE tag external links, then it consists of *ntk* TDFINTS, followed by *ntg* TDFINTS.

   The *ntk* integers describe the *ntk* external tokens, the first integer describing the first external token. Each such integer will be a number between 0 and 15, interpreted as below.

   The *nt*g integers describe the *nt*g external tags, the first integer describing the first external tag. Each such integer will be a number between 0 and 15, interpreted as below.

   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: 1 means “defined in this capsule and can only be uniquely defined”, 0 means “not defined in this capsule”.

   Bit 3: 1 means “defined in this capsule and can be multiply defined”, 0 means “not defined in this capsule”.

INDEX

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

X