303121

UNLIMITED

2

AD-A240 823

Report No. 91014

Report No. 91014

ROYAL SIGNALS AND RADAR ESTABLISHMENT, MALVERN

DTIC ELECTE SEP 19 1991 S D D

TDF: SPECIFICATION OF SUBSET TO SUPPORT ANSI C, C++, FORTRAN 77, COBOL & PASCAL

Authors: J M Foster, M Brandreth, P W Core, I F Currie & N E Peeling

91-10970

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

91 9 18 082

May 1991

UNLIMITED

DEFENCE RESEARCH AGENCY, ELECTRONICS DIVISION
(RSRE, MALVERN)

Report 91014

Title: TDF: Specification of Subset to Support ANSI C, C++, FORTRAN 77, COBOL and Pascal

Authors: J M Foster

M Brandreth
P W Core
I F Currie
N E Peeling

Date: May 1991

Summary

TDF is an intermediate format for distributing software applications developed by the United Kingdom's Defence Research Agency, Electronics Division at RSRE, Malvern. Report no. 91005 gave an account of the whole of TDF. The present report updates the account of the subset of TDF which supports ANSI C, C++, FORTRAN 77, COBOL and Pascal, described in 91005 as TDF Level 0.

The **Introduction** gives an overview of the TDF concept and sets the scene for the **Definition**. This specifies each of the constructs which make up the subset of TDF described here. A **Glossary** gives a quick explanation of some key TDF terms.

2.21 TOKDEF

2.21.1 make_tokdef

2.22 TAGDEF

2.22.1 make_tagdef

2.23 TOKLINK

2.23.1 make_toklink

2.24 TAGLINK

2.24.1 make_taglink

2.25 TOKEN

2.25.1 apply_token

3 Glossary

1 Introduction

1.1 TDF: Scenario of Use

TDF is an intermediate format for distributing software applications developed at the United Kingdom's Defence Research Agency, Electronics Division at RSRE, Malvern.

TDF can be produced from a very wide range of programming languages and installed on a very wide range of architectures. The languages which TDF has been designed to cater for include ANSI C, C++, FORTRAN 77, COBOL, Pascal, Ada, Modula2, Common Lisp and Standard ML. This document describes a subset of TDF for which prototype software exists which demonstrates its suitability for ANSI C. The subset described in this document was also designed to support C++, FORTRAN 77, COBOL and Pascal.

TDF is defined in the form of a data-structure which is an abstract syntax for programs. It contains sufficient information to allow efficient machine code to be generated from it for a wide variety of computer architectures. A TDF data-structure representing program is encoded into a linear stream of bits and resides in a file. The encoding of this stream of bits is both space efficient and extensible so as to allow upwards compatibility for any future enhancements or amendments to the TDF definition.

TDF can be used for distributing "shrink-wrapped" software. To do this, a software vendor writes an application in a familiar programming language and then produces from it a single version of the application in TDF. The software that produces the TDF is called a TDF **producer**. The largest single component of the producer is likely to be the program that converts a program written in a high-level language, such as ANSI C, into TDF. We refer to this as the **compiler** component of the producer. Once encoded, the TDF is then shipped to any of a number of target computers owned by a software purchaser. The software that converts the encoded TDF into an executable program on a target is referred to as a TDF **installer**. The largest single part of the installer will be the program that generates machine code from arbitrary TDF programs. We refer to this as a TDF **translator**.

1.2 TDF: Level of Definition

TDF constructs are generalisations of the constructs found in different programming languages. They have been designed to satisfy the following requirements:

- All the information that a programming language can represent which helps a code generator produce efficient code should be representable in TDF. This means that programs distributed in TDF can be as efficient as if they were compiled with the best compiler on any target.

• Commonly provided hardware features should be easy to use - for instance, the single instruction "array and bound check" provided by many machines.

• As many optimisations as possible should be expressible as TDF to TDF transformations, allowing these optimisations to be written portably. They might be universal (i.e. beneficial for all languages and all target machines), in which case they could be included in a general-purpose TDF to TDF optimiser; they might be language specific, in which case they could be included in any of the compiler components for that language; or they might be specific to a class of architectures, in which case they could be included in translators for that class of target.

To satisfy these requirements, TDF has been designed as a wide-spectrum interface which at its highest level generalises high-level programming languages, whilst at its lowest level generalising assembler codes.

1.3 Values within a TDF System

Programming languages have always had the notion of static and dynamic values. Static values were those known at compile-time whilst dynamic values were calculated at run-time. The situation in TDF is similar. We will use the term "static" to describe values known at translate-time and "dynamic" to describe values which are calculated at run-time. (Note that in ANSI C the term "static" has a different meaning.)

1.3.1 Dynamic Values

We will start by considering run-time values. In programming languages, run-time values tend to be classified by a type system. Types are used for three different purposes in programming languages. Firstly, they help the programmer to model data in as natural a way as possible by providing a system of convenient data-structures - records, arrays etc. Secondly, they allow many structural programming errors to be detected at compile-time. Lastly, they provide information to a compiler which helps it to generate efficient machine-code.

The TDF analogues of types are SHAPEs. They serve only the last of the three purposes described above - providing the information which translators need in order to achieve efficient memory management for any programming language on target architectures. SHAPEs are therefore designed to provide an architecture neutral abstraction of memory management making no assumptions about the properties of targets (word length, alignment constraints etc.).

1.3.2 Static Values

Apart from run-time values, there is another set of values in this TDF definition. These are the pieces of TDF program themselves, which are output by compilers. These TDF values are classified into their own system of categories which we refer to as SORTs. SORTs are analogous to the syntactic classes found in high level programming languages - identifiers, expressions, types etc. For instance, SHAPE is one of the SORTs.

All pieces of TDF program, whatever SORT they are, are by definition static (ie. known at translate-time). Values generated by program, whatever SHAPE they are, are in general dynamic (ie. known only at run-time). However, it may sometimes be possible to evaluate run-time expressions at translate-time, in which case they are static after all and may offer opportunities for optimisation.

1.3.3 SORTs and SHAPEs: an Example

The treatment of integers provides a good example of the relation between SORTs and SHAPEs. Pieces of TDF program which, when evaluated at run-time will generate values, are of SORT EXP. (EXP stands for 'expression'.) Each EXP can be characterised by the SHAPE of the value which it will generate. For instance, an EXP which will generate an integer value is said to have INTEGER SHAPE. Values of this SHAPE can describe any run-time integer - eg. a dynamically calculated index of an array.

Pieces of program which by contrast stand for integers known at translate-time are of SORT NAT. (NAT stands for 'natural number'.) They are not EXPs which have to be evaluated in order to generate their integer values. Instead, they already are integer values. A piece of TDF program of SORT NAT can describe any compile-time known integer - eg. a statically calculated bound for trimming an array.

1.3.4 SHAPE- and SORT-correctness

TDF relies on the programming language compiler to determine to what extent the SHAPE-correctness of programs is enforced. (An example of SHAPE-incorrectness is the multiplication of two POINTERs.) Compilers from strongly typed languages will naturally produce SHAPE-correct programs.

Likewise, the SORT-correctness of the TDF produced by a compiler is dependent on the correctness of the compiler implementation. Neither SORT-correctness nor SHAPE-correctness need be checked by a TDF translator.

1.4 Identification of Values

TDF provides two different methods of identifying values by names, one static and one dynamic. Identifiers which statically identify pieces of TDF program are called TOKENs. They loosely correspond to ANSI C's parameterised macros but are a great deal more powerful. Identifiers in TDF program that dynamically identify run-time values are of SORT TAG. These identifiers correspond to the names of variables and procedures in programming languages such as ANSI C.

TDF identifiers, be they TAGs or TOKENs, do nothing more than set up name/value correspondence. All the syntactic "sugar" associated with identifiers in programming languages - the use of mnemonic identifiers, the complexities of overloading and hiding - is provided solely to aid the human readability of programs. It provides no information which assists in the production of efficient machine code and hence has no relevance to TDF. All such syntactic "sugar" is eliminated by compilers to TDF.

TAGs correspond to identifiers in programming languages. But TOKENs are a concept devised specifically to handle the issues that arise when software is distributed via an ANDF, as opposed to being compiled and translated on a single machine. They are addressed in the following section.

1.5 Tokenisation

When a TOKEN is used to stand for a piece of TDF program, that piece of program is said to have been "tokenised". A TOKEN identifies a (possibly parameterised) piece of program which can be of any SORT. Significantly, the definition of the TOKEN - in procedure or macro terms, its body - can be supplied at a number of different times in the production or installation process:

- It might be supplied by the producer and bound together with the program to which it relates, in which case that definition will be distributed identically to all targets. A typical usage would be to make a commonly occurring piece of program the subject of a token definition in order to compress the size of the distributed TDF. The substitution of the definition for the TOKEN will be performed by the installer.

- The definition might be supplied by the installer. There are two variations of this:

    - a piece of program might be used so frequently that its definition is known by all installers. This is a similar usage to the one above but eliminates the need to distribute the TOKEN's definition, which compresses the TDF even more.

    - the piece of program to be substituted for the TOKEN might be target specific - e.g. the datastructure used by a print procedure.

- The TOKEN may be recognised by the translator and implemented directly - ie. not actually considered to stand for a piece of TDF program at all. There are a number of uses for this approach:

    - A TOKEN might be used to represent an operation such as vector inner product. A producer might supply an architecture neutral definition of the TOKEN. But an installer on a machine s· :h as a CRAY might choose to ignore the portable definition and make full use of the CRAY's parallelism in implementing the vector inner product.

    - A TOKEN might be used to represent an operation that was implemented by many architectures, but not by all. (IEEE floating point operations are an example of this.) Because TDF is an architecture neutral representation of program, it cannot provide constructs describing such operations - that would make it impossible to implement fully on certain architectures. Useful but non-universal operations could nonetheless be accessed by using TOKENs to stand for them. Program making use of such TOKENs would of course be translatable only on architectures which provided the required operations.

    - If a new language were invented requiring a new feature to be added to TDF, it could be defined as a TOKEN, which installers implemented according to its definition.

- The TOKEN might be bound during linking to an external function that has been precompiled from a programming language, or directly written in assembler. The mechanism for doing this is defined as part of the installation process.

Some of these uses of TOKENs require installers to know the TOKENs' meanings. Installer writers will therefore need to refer to a list of such TOKENs and meanings as well as to this Specification when writing their installers.

1.6 TDF Terminology

Before going further, we need to explain some of the terminology and notation used later on in this document. Firstly, we will define the terms used to specify the behaviour of TDF translators: and then we will introduce the conventions used to describe TDF program constructs.

1.6.1 Specifying Translator Behaviour

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

- "undefined": means that translators 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 translator must perform P.

- "should": when the phrase "P should be done" (or similar phrase involving "should") is used, translators are advised to perform P, and compiler 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 translator may assume that P holds without having to check it. If, in fact, a compiler 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 translator shall define everything which is said to be "target-defined".

1.6.2 Describing Program Construction

As mentioned in §1.1, the linear stream of bits which constitutes a TDF program encodes the abstract syntax tree of that program. As may be imagined, the encoded form of TDF is not a convenient medium for describing the structure of TDF program to the human reader! So in this document we talk in terms of the abstract syntax tree: but before we do this, we need to state the notation which we are going to use.

§1.3.2 explained that pieces of TDF program are categorised into different SORTs, analogous to the syntactic classes of high-level programming languages. Some SORTs consist of a fixed number of named alternatives. To indicate a particular alternative, we simply write its name. For instance, the two alternatives for the SORT BOOL appear as:

```
true

false
```

Other SORTs consist simply of integers or a subset of integers which can be written down in the usual way, eg. a NAT:

3

Certain SORTs can consist of a tuple of components (ie. they are Cartesian products of other SORTs). To write these down we list their components. For instance, a VARIETY may consist of a pair of SIGNED_NATs:

(0,255)

The SORTs EXP and SHAPE are recursively defined, with a considerably richer set of primitives and constructs than the other SORTs have.

In text, the names of EXP constructs will appear in lower case italics.

Primitive EXPs - ie. EXPs which do not require arguments - are simply named, as in:

make_top

The application of an EXP construct is denoted as follows:

```
goto(2,
     make_top
)
```

As with EXPs, primitive SHAPEs are simply named, as in:

PROC

And the application of a SHAPE construct is denoted as follows:

POINTER(PROC)

In text, the names of SHAPE constructs will appear in upper case. (The reader may already have noticed that the names of the SORTs also appear in upper case - as does the word SORT.)

Since the SHAPEs of values produced when EXPs are evaluated are important, we generally state the SHAPE when specifying TDF constructs. For instance, an EXP which evaluates to produce a value of SHAPE PROC is described as an EXP PROC. TAGs, which name run-time values, and LABELs which identify pieces of program expecting run-time values to be supplied to them, are likewise qualified.

The following example, which specifies the construct *truncate*, shows how the EXP and SHAPE notations look in practice:

```
truncate
  ov_err: ERROR_TREATMENT,
  v: VARIETY,
  arg: EXP FLOAT(F)

    -> EXP INTEGER(v)
```

The construct's arguments (three 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 any English description which may accompany the notation and for cross-referencing within the notation. It has no other significance.

The example given above indicates that *truncate* takes three arguments. The first argument, *ov_err*, is of SORT ERROR_TREATMENT. The second, *v*, is of SORT VARIETY. The third argument, *arg*, is an expression of SORT EXP, and as mentioned before we append the SHAPE of the EXP, FLOAT(F). *arg* is the piece of program which will deliver the floating point number to be truncated.

After the "->" comes the SORT of the result of *truncate*. The result is an EXP INTEGER(v) - a piece of program which, when evaluated, will deliver a value whose SHAPE is INTEGER(v), the truncated floating point number. The "v" is an example of cross-referencing within the specification of a construct.

Section §1.3.4 stated that SHAPE-correctness need not be checked by a TDF translator. If constructs are formed with EXP arguments whose SHAPEs deviate from those prescribed in this document, the effect is undefined.

The format for the description of the construction of a SHAPE is similar to that for EXPs. For instance, the SHAPE construct SOME:

```
SOME
  s:SHAPE

    -> SHAPE
```

takes one SHAPE argument, which for the purposes of any accompanying English text is named *s*, and yields a SHAPE result.

Four further conventions are needed in order to describe TDF constructs. Some constructs may take a variable number of arguments. For instance, *sequence* may take any number of components greater than zero. We write this as:

$$\Pi_{i=1}^{n} \text{EXP } Y_i$$

The symbol "Π" indicates a cartesian product; *i* ranges from 1 to *n*; and the EXP Y_i are the components. In addition it is sometimes necessary to add qualifying predicates, which we enclose in curly brackets, as in:

$$\Pi_{i=1}^{n} \text{EXP } Y_i \ \{ n > 0 \}$$

Some constructs have arguments which may optionally be omitted. To indicate this in the definition of the construct, we enclose the SORT of the optional argument in brackets and apply a postfix _OPTION, e.g.

(BOOL)_OPTION

meaning either a BOOL or nothing.

The absence of an optional argument in the application of a construct is denoted by leaving a blank space where the argument would have been - eg.:

```
make_tokdef(74,
            ,
            make_int((0,255),65)
           )
```

The second argument has been omitted here.

With an underst^nding of the notation used to describe TDF and the meaning of the terms used to describe translator behaviour, we can now look at how TDF achieves its complete architecture neutrality with the help of some examples.

1.7 TDF: Architecture Neutrality

The achievement of complete architecture neutrality has been the first priority in designing TDF. The slightest shortfall from this goal would seriously undermine its usefulness as a software distribution format. This section explains how TDF allows target-dependent features of programming languages to be completely factored out of producers and dealt with exclusively in each architecture's installer.

This complete separation of concerns means that a producer can be used to produce TDF for installation on any architecture with no alteration whatsoever.

1.7.1 Architecture Neutral Memory Allocation through SHAPEs

The design of SHAPE constructs which provide a totally symbolic description of the representation of run-time values is a central issue in the design of TDF and so merits a detailed explanation in this section.

The following example provides an illustration. Values of the C type:

```
struct{unsigned char c; double f;}
```

will typically be given the TDF SHAPE:

```
TUPLE(INTEGER(0, 255), FLOAT(2, 56, 0, 8))
```

when compiled to TDF. (TUPLE is TDF's SHAPE construct describing cartesian products, INTEGER describes integers, and FLOAT describes floating point numbers.)

The TUPLE SHAPE shown above is a straightforward mapping of the C type, preserving the information that it is a 'struct'. When compiling in an architecture neutral fashion one cannot afford to throw away this information. The reason for this is that different architectures have different alignment rules. Without the knowledge that one was dealing with a 'struct', correct and efficient translation to machine code in this case would be impossible. For example, on a machine which placed no restriction on accessing words or floating point numbers at odd byte boundaries, one could compactly represent this structure in 9 bytes - 1 for the 'char' and 8 for the 'double'; a less liberal one which favoured word addressing might need 3 bytes of padding after the 'char', so requiring 12 bytes in total; and a really illiberal one might require 16 bytes by insisting that 'doubles' start on 8-byte boundaries.

Clearly, it is no use simply specifying the number of bytes required for storing the 'struct', since this will vary from architecture to architecture. What is required, and what TDF offers, is the ability to give all the information about the 'struct' which is necessary for individual architectures' optimum space allocations to be determined.

Besides TUPLE, other SHAPE constructs cover integers, floating point numbers, procedures, pointers, unions, static and dynamic arrays and so on in a similarly architecture neutral fashion.

But as well as providing SHAPEs which allow translate-time memory management to be described in an architecture neutral fashion, TDF needs to provide support for the manipulation of target-dependent offset information at **run-time**, as exemplified by C's pointer arithmetic.

It does this using the SHAPE construct OFFSET.

1.7.2 Architecture Neutral Pointer Arithmetic

The SHAPE construct OFFSET is needed in order to achieve completely architecture neutral pointer arithmetic. Addition to a pointer, *p*, to an array of values of SHAPE X to obtain a pointer to the array's third value provides an example:

```
add_to_ptr(p,
           offset_mult(array_element_offset(X),
                       make_int((0,255),2)
                      )
          )
```

array_element_offset takes the SHAPE X and calculates the distance between successive elements in an array of values of SHAPE X. The SHAPE of the value which it delivers is OFFSET(X,X), meaning that it measures the offset between a value of SHAPE X and another value of SHAPE X. *offset_mult* multiplies the OFFSET(X,X) by 2 to produce another OFFSET(X,X) value. *add_to_ptr* then adds that OFFSET(X,X) to the pointer, *p*, to produce a new pointer which points to the third element of the array pointed to by *p*.

TDF's OFFSETs allow the inherently target-dependent business of pointer arithmetic to be described in an target-independent way. Some OFFSET calculations can be performed at install-time, while others have of necessity to wait until run-time. But in each case, the calculations are performed when knowledge of the target's characteristics is available.

1.8 TDF CAPSULEs and Linking

Architecture neutral memory allocation and pointer arithmetic are important aspects of target neutrality. But in order to complete the "shrink-wrapped" software scenario, it also necessary for the software distribution medium to be able to accommodate the linkage of target-**independent** software with target-**dependent** software.

In the conventional scenario, application software is linked with target-dependent libraries on the developer's hardware, compiled, and then distributed - necessarily only to the architecture to which the target-dependent libraries related. Compilation and linkage take place on the developer's hardware: all that happens on the user's hardware is that the application runs.

But in the "shrink-wrapped" scenario, linking to target-dependent software before distribution is inadmissible. Instead, this linking must be carried out on the target. As a medium for "shrink-wrapped" software distribution, TDF has been designed to express the information needed to achieve this linking on the target. (In fact, as will be seen, TDF's approach to linking is completely general, and permits linking between any combination of target-independent and target-dependent software on either the developer's or the user's hardware.)

Separate pieces of TDF program are called CAPSULEs. In the following section we explain how CAPSULEs express the necessary linking information and how linking is performed on the target.

1.8.1 The Content of CAPSULEs and the TDF Builder

TDF CAPSULEs will normally reside in separate files in the host operating system. Inside the file will be the linear stream of bits referred to in §1.1, encoding the CAPSULE's contents.

A CAPSULE contains the **definitions** of a number of TAGs and TOKENs. When it is translated to machine code and the code is loaded, the values associated with a CAPSULE's TAGs will be made available to the system linker in the usual way. Before giving the definitions of its TAGs and TOKENs, a CAPSULE will provide **declarations** of them: these indicate the SHAPEs of the TAGs and the SORTs of the TOKENs. The declarations precede the definitions in order to allow for recursive definitions.

TDF permits pieces of program which are incomplete - probably because they lack some target-dependent component - to be distributed to target machines. Being incomplete, such CAPSULEs cannot be translated straightaway. For instance, a distributed CAPSULE might use a TOKEN which stands for the SHAPE corresponding to the ANSI C type "FILE". The target-independent CAPSULE cannot itself provide the definition of that TOKEN because the definition is target-dependent. It therefore needs to be merged with a local, target-dependent CAPSULE which does contain the TOKEN's definition before the TDF can be understood by an installer and translated.

This merging process is carried out by a program written specially to support TDF, known as the **TDF Builder**. Before we investigate the action of the TDF Builder, we need to look at the contents of CAPSULEs in a little more detail.

Strings provide the naming information with system linkers need in order to match up the values which separate pieces of code make available or use. The TDF Builder also uses strings to match up values and program fragments made available or used by CAPSULEs when it merges them before translation. Inside CAPSULEs, however, values and program fragments are identified not by strings but by TAGs and TOKENs. And so in order to make them accessible to the TDF Builder and to system linkers, a CAPSULE associates strings with those of its TAGs and TOKENs which are to be the subject of building or linking. As well as associating strings with TAGs and TOKENs, CAPSULEs give some additional information about the TAGs and TOKENs: they indicate whether or not the TAG or TOKEN is declared, defined or used in the CAPSULE.

The TDF Builder's task is not just a matter or extracting all the declarations, definitions and string associations from its argument CAPSULEs and throwing them all together to construct a new, bigger CAPSULE: it may have to reconcile the different TAGs and TOKENs which the argument CAPSULEs have used to name the same things. For instance, one CAPSULE may give a definition for the TOKEN 67 and associate it with the string "clib_file" while another uses the TOKEN 102 and associates it with the string "clib_file". Both CAPSULEs are talking about the same thing, but since they were produced independently they quite naturally call it by different names - TOKENs - locally. To resolve this clash, the TDF Builder associates a renaming with each CAPSULE's set of declarations and definitions. In this case, it might add 67->135 to the first CAPSULE's set and 102->135 to the second CAPSULE's set.

In summary, then, we have seen that a CAPSULE contains a set of declarations and definitions of TAGs and TOKENs. It associates strings with some of the TAGs and TOKENs, for the benefit of system linkers and the TDF Builder. It gives information as to whether each of these TAGs and TOKENs is declared, defined and used in the CAPSULE. A composite CAPSULE which has been put together by the TDF Builder may contain internal renamings in order to resolve disagreements between the sets of declarations and definitions which came from different CAPSULEs.

A CAPSULE which is capable of being translated will not necessarily give definitions for all the TAGs and TOKENs referred to by the code which it contains. Values corresponding to TAGs whose definitions it does not provide will have to be linked in by the system linker after translation: and fragments of program corresponding to TOKENs whose definitions it does not provide will have to be known by the installer (as explained in §1.5).

(The TDF Builder has been presented as being used purely to merge target-independent CAPSULEs with target-independent ones. In fact, it could be employed to merge any combination of target-independent and target-dependent CAPSULEs since it is completely indifferent to the origin and status of the CAPSULEs which it processes.)

2 Definition

Having set the scene by describing the top-level structure of a TDF CAPSULE, we are now in a position to look at the finer detail of TDF. This involves giving an account of all the TDF SORTs and the constructs which can create them. Each of the SORTs is described in turn, beginning with those which equate most easily to the syntactic classes of high-level programming language - eg. types, expressions etc. In fact, SHAPE and EXP - the TDF SORTs which correspond to these - are the richest SORTs and the descriptions of them occupy the greater part of this document.

There are 25 SORTs:

SHAPE	FLOATING_VARIETY	BOOL	USAGE	TOKDEF
EXP	TAG	ERROR_TREATMENT	UNIT	TAGDEF
NAT	LABEL	CAPSULE	TOKDEC	TOKLINK
SIGNED_NAT	NTEST	TOKEXTERN	SORTNAME	TAGLINK
VARIETY	STRING	TAGEXTERN	TAGDEC	TOKEN

2.1 SHAPE

SHAPEs give TDF translators symbolic size and representation information about run-time values. Values of the same SHAPE will be represented in the same way and occupy the same amount of memory at run-time on a given architecture.

The construction of SHAPEs is recursive and is built up from a set of **primitive** SHAPEs which describe values such as bits and procedures, and SHAPE **constructors** for describing values such as tuples, arrays (both statically and dynamically sized), pointers and unions.

2.1.1 Primitive SHAPEs

There are four primitive SHAPEs.

2.1.1.1 BOTTOM

BOTTOM is the SHAPE which describes pieces of program which do not return any result. Examples include *goto* and *return*.

2.1.1.2 TOP

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.

2.1.1.3 BIT

BIT is the SHAPE which describes values which have only two possible conditions - true or false.

2.1.1.4 PROC

PROC is the SHAPE which describes procedure values.

2.1.2 SHAPE Constructors

Compound SHAPEs are SHAPEs which are not primitive. They are created by SHAPE constructors which take arguments.

2.1.2.1 INTEGER SHAPEs

Most integer arithmetic operations - *plus*, *minus* etc. - work in the same way on different kinds of integer. But on most architectures if the operation is dyadic, the arguments must be of the same kind and the result will also be of that kind.

The different kinds of integer are disti guished by having different VARIETYs. (The SORT VARIETY was introduced in §2.5.) SHAPEs describing integers are constructed by the SHAPE construct INTEGER, taking a value of SORT VARIETY as its argument:

variety: VARIETY

-> SHAPE

Thus:

INTEGER(0,255)

is a SHAPE describing an integer value whose VARIETY is (0,255), specifying that it may lie between 0 and 255 inclusive, and for which a translator can accordingly plan space.

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 VARIETYs of the two arguments must be the same. 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 words: on a 16-bit machine these two would, but on a 15-bit machine they would not. The only way to be sure that two INTEGERs are going to be represented in the same number of words on **all** architectures is to stipulate that their VARIETYs must be 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 error occurs. Whether this occurs in particular circumstances is target-defined, because installers' decisions on representation are inherently target-defined. Thus the calculation of 200+250, where both values are of SHAPE INTEGER(0,255) may or may not cause an overflow depending on how the installer represents INTEGER(0,255)'s. If it uses only 8 bits, an overflow will occur: if it uses 16, there will be no overflow.

2.1.2.1.1 Recommendations about Integer VARIETYs

Two recommendations are made about the use of integer VARIETYs.

- **First recommendation**: the SIGNED_NATs delimiting VARIETY should reflect as precisely as possible what is needed by the program. This choice should not be influenced by knowledge of what is available on common machines (except where the purpose is specifically to take advantage of such knowledge). It is the task of the TDF translator to make intelligent decisions.

- **Second recommendation**: where possible, VARIETYs should be tokenised in such a way that useful selective alterations may be made when a program reaches the target machine. It may be that certain operations involving integers can usefully be transformed to make optimum use of an architecture's facilities. So that the relevant integer VARIETYs can be substituted selectively, the integer arguments to these operations should belong to a particular tokenised VARIETY, and other integers to another VARIETY.

2.1.2.2 Floating Point SHAPEs

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 also of this kind.

The different kinds of floating point number are called FLOATING_VARIETYs. (FLOATING_VARIETYs were introduced in §2.6.) SHAPEs describing floating point values are constructed by the SHAPE construct FLOATING, taking a value of SORT FLOATING_VARIETY as its argument:

fv: FLOATING_VARIETY

-> SHAPE

Thus:

FLOATING(10,30,-5,15)

is the SHAPE of a floating point value of FLOATING_VARIETY (10,30,-5,15). This signifies that its BASE is 10, it has 30 digits in its MANTISSA, its MINIMUM_EXPONENT is -5 and its MAXIMUM_EXPONENT is 15.

BASE is the base with respect to which the remaining numbers are given.

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

MINIMUM_EXPONENT is the required integer, *n*, such that BASE raised to the power *n* can be represented as a non-zero floating point number.

MAXIMUM_EXPONENT is the required integer such that BASE raised to that power is representable as a floating point number of the variety.

The BASE specified need bear no relation to the base for floating point numbers in any target architecture. For instance, the BASE may be 10, while the implementation may be binary.

The use of a FLOATING_VARIETY in TDF expresses the intention that a correct program will only use the values implied by the requirements. A TDF translator is required to make available a representation such that, if only values within the requirements are produced, no overflow error will occur. The effect of using values outside the requirements is undefined, but an overflow error may be produced.

Any number of FLOATING_VARIETYs may be asked for by a TDF program, though it is recommended that the number should be severely limited. The space taken in the TDF for transmission of FLOATING_VARIETYs should be minimised by tokenising (§1.5) the required FLOATING_VARIETYs and using the TOKENs instead of the full form.

2.1.2.2.1 Recommendations about FLOATING_VARIETYs

Two recommendations are made about the use of FLOATING_VARIETYs in TDF.

- **First recommendation**: when arguments are chosen to define a FLOATING_VARIETY their values should reflect as precisely as possible what is needed by the program. This choice should not be influenced by knowledge of what is available on common machines. It is the task of the TDF translator to make intelligent decisions.

- **Second recommendation**: FLOATING_VARIETYs should be tokenised in such a way that useful selective alterations may be made purely in the target machine. It may be that a certain operations involving floating point values can usefully be transformed to make best use of an architecture's facilities. So that the relevant floating point VARIETYs can be selectively substituted, the floating point arguments to these operations should belong to a particular tokenised FLOATING_VARIETY, and other floating point values to another FLOATING_VARIETY.

2.1.2.3 POINTER SHAPEs

A POINTER is a value which points to a space allocated in the computer's memory. The POINTER constructor takes a SHAPE argument:

arg: SHAPE

-> SHAPE

The argument SHAPE describes the value to which the pointer points. It will not be TOP. Eg.:

POINTER(INTEGER(0,255))

The provision of an argument SHAPE gives TDF translators the freedom to implement POINTERs in different ways depending on the SHAPE of the values to which they point. Otherwise the rule that values with equal SHAPEs have the same representation on any given architecture would prevent this.

The lifetime of an POINTER depends on the manner of its creation. If it arises from a *variable* construct, its lifetime extends over the body of that construct. If it arises from explicit use of a library routine, its lifetime depends on the content of the library routine.

There shall be an upper bound to the size of representation of POINTERs. This is important in the construction of circular SHAPEs (§2.1 2.12).

2.1.2.4 TUPLE SHAPEs

The SHAPE constructor TUPLE describes cartesian products. It takes a variable number of SHAPE arguments:

components:$\Pi_{i=1}^{n}$ (s_i:SHAPE) { n >= 0 }

-> SHAPE

None of the *components* will be TOP.

Translators shall represent the component values of a TUPLE value in memory in the same order as they occur in the TUPLE construct. Furthermore, the representation of the first n fields of a TUPLE shall be unaltered by adding an additional field at the end. This requirement satisfies C++'s need for pointers to be able to access different subsets of the same TUPLE value.

2.1.2.5 PARAM_PACK SHAPEs

The SHAPE constructor PARAM_PACK describes collections of one or more values gathered together for supplying as parameters to a procedure:

components:$\Pi_{i=1}^{n}$ (s_i:SHAPE) { n >= 0 }

-> SHAPE

Some architectures require that procedure parameters be laid out in a special way, differently from (say) TUPLEs. In order to maintain architecture neutrality, TDF reflects this requirement by providing PARAM_PACK. TDF PROCs always take one argument, which will be of SHAPE PARAM_PACK(..).

All the operations applicable to TUPLEs are applicable to PARAM_PACKs.

2.1.2.6 UNION SHAPEs

The UNION SHAPE constructor describes values which may take one of a number of SHAPEs:

> alternatives: $\Pi_{i=1}^{n}$ (s_i:SHAPE) { n > 0 }
>
> -> SHAPE

None of the *alternatives* will be TOP. A discriminant to determine which alternative is present is **not** a part of the value. If it is needed, such discrimination must be performed elsewhere.

2.1.2.7 OFFSET SHAPEs

The SHAPE constructor OFFSET describes values which measure offsets in memory. (It should be emphasised that these are in general **run-time** values).

OFFSET takes two SHAPE arguments in order to allow OFFSETs to and from different SHAPEs to be represented differently. This gives installer writers the latitude necessary to deal with complex memory layouts:

> sh1: SHAPE
> sh2: SHAPE
>
> -> SHAPE

An OFFSET (X,Y) measures the offset between a value of SHAPE X and a value of SHAPE Y in a datastructure.

2.1.2.8 NOF SHAPEs

The NOF SHAPE constructor describes arrays of values whose size is known at translate-time:

> s: SHAPE,
> n: NAT
>
> -> SHAPE

s gives the SHAPE of the constituent elements of the array, and *n* says how many there are. The SHAPE *s* will not be TOP.

2.1.2.9 SOME SHAPEs

The SOME SHAPE constructor describes arrays of values whose size is not known at translate-time:

```
s: SHAPE

-> SHAPE
```

SOME SHAPEs are not on the same footing as the other SHAPEs. Since the size of values of SHAPE SOME(X) is not determinable at translate-time, they cannot be accommodated in a procedure's workspace. Nor can any compound value which contains a SOME(X), and whose size depends on the size of that SOME(X). The only SHAPE construct which breaks this dependency is POINTER. It does so because POINTERs are defined to have the same size regardless of the size of the array to which they point: translators shall implement POINTER(SOME(X))'s in the same size no matter how many X's there are.

SHAPEs whose size does not depend on the size of a constituent SOME(X) are known as "SOME-free". For example,

```
UNION(INTEGER(V),
      FLOAT(F)
     )
```

and

```
TUPLE(NOF(POINTER(SOME(PROC))),
      INTEGER(V)
     )
```

are SOME-free, whereas

```
TUPLE(NOF(SOME(PROC)),
      INTEGER(V)
     )
```

is not.

Certain TDF constructs stipulate that the SHAPE of an argument EXP will be SOME-free. These stipulations work together to ensure that no TDF program will attempt to introduce a value whose size is not determinable at translate-time into a procedure's workspace.

2.1.2.10 ENV SHAPEs

The ENV SHAPE constructor describes values which give access to the local values named and the LABELs visible in the course of a particular procedure application. An EXP construct (*obtain_nl_tag*) is provided which takes an ENV argument and extracts a named value from the procedure application to which the ENV relates. Another construct (*goto_nl*) takes a number of arguments, including an ENV and a LABEL_VALUE, and passes control to the relevant LABEL in the procedure application concerned.

The ENV constructor takes a TAG argument:

```
t: TAG

-> SHAPE
```

t identifies the ENV as relating to an application of the procedure named *t* in the enclosing CAPSULE. The reason for the TAG argument is that without knowing which procedure an ENV related to, it would be impossible at translate-time to output the code necessary to extract the procedure's local values or jump to one of its locations.

2.1.2.11 LABEL_VALUE SHAPEs

The LABEL_VALUE SHAPE constructor describes values which enable non-local or "long" jumps from one procedure to another.

There is one constructor, which takes a SHAPE argument:

```
sh: SHAPE

-> SHAPE
```

sh is the SHAPE of the value which will be passed when the LABEL_VALUE is used to perform a jump.

2.1.2.12 Circular SHAPEs

Circular SHAPEs can be constructed using TOKENs. The following TOKDEF provides an example: it defines "X" to be the SHAPE of a list of INTEGER(V)'s.

```
make_tokdef(X,
                ,
                TUPLE(INTEGER(V), POINTER(X))
               )
```

There will be a POINTER in the cycle. This, together with the fact that there is an upper bound on the size of representation of POINTERs, means that this method cannot be used to construct SHAPEs whose memory requirement is infinite.

2.2 EXP

A value of SORT EXP is a piece of program that generates or manipulates run-time values. EXP is by far the richest SORT, with 99 constructs. There are few primitive EXPs: most are constructors which take mixtures of EXPs and arguments of other SORTs. There are constructs delivering EXPs that correspond to the declarations, program structure, procedure calls, assignments, pointer manipulations, arithmetic operations, tests etc. of programming languages.

The EXP constructs can conveniently be broken down into ten broad classes concerned with:

Declarations and Naming
Integers and Arithmetic
Floating Point Values
POINTERs
Procedures
Program Structure and Flow of Control
OFFSETs
NOFs and SOMEs
TUPLEs, PARAM_PACKs and UNIONs
Miscellaneous

These are described in the following sections. (Remember that the notation used here was introduced in §1.6.)

2.2.1 Declarations and Naming

2.2.1.1 identify

```
register: BOOL,
local: BOOL,
name: TAG X,
def: EXP X,
body: EXP Y


  -> EXP Y
```

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

The value delivered by *identify* is that produced by the evaluation of *body*. Thus the SHAPE of the value delivered by *identify* is the same as the SHAPE, Y, of *body*.

The BOOL, *register*, gives information about the usage of *name*: if true, it indicates that *name* is heavily used within *body* and that allocation to a register, if possible, would be advantageous.

The BOOL, *local*, gives information about possible external access to *name*: if true, *name* will not be supplied as an argument to *obtain_nl_tag* - ie. there will be no non-local access to the value which *name* names. This information is of value in deciding whether certain optimisations are possible.

The TAG given for *name* will not be re-used within the current UNIT. No rules for the effect of the hiding of one TAG by another, equal TAG are given: this will not happen. See §2.2.1.4 for a discussion of this point.

In the case where *def* is simply *obtain_tag(t)*, translators should produce no code, since this usage of *identify* amounts to a mere renaming of *t* as *name*. Similarly, if *def* is constructed by a succession of *field* constructs on *obtain_tag(t)*, translators should produce no code, since this usage amounts to the naming of a part of a value which has already been named.

2.2.1.2 variable

```
register: BOOL,
local: BOOL,
name: TAG POINTER(X),
init: EXP X,
body: EXP Y


  -> EXP Y
```

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

The value delivered by *variable* is that produced by the evaluation of *body*. Thus the SHAPE of the value delivered by *variable* is the same as the SHAPE, Y, of *body*.

The BOOL, *register*, gives information about the usage of *name*: if true, it indicates that *name* is heavily used within *body* and that allocation to a register, if possible, would be advantageous.

The BOOL, *local*, gives information about possible external access to *name*: if true, *name* will not be supplied as an argument to *obtain_nl_tag* - ie. there will be no non-local access to the value which *name* names. This information is of value in deciding whether certain optimisations are possible.

The TAG used for *name* will not be re-used within the current UNIT. No rules for the effect of the hiding of one TAG by another, equal TAG are given; this will not happen. See §2.2.1.4 for a discussion of this point.

The POINTER associated with *name* has a lifetime limited to the execution of *body*. Any attempt to use it when *body* is not being executed is undefined.

The sharing properties of the POINTER are discussed in §2.2.4.1.1.

When compiling programming languages which permit uninitialised variable declarations, *make_value* is used to provide the undefined *init* EXP.

2.2.1.3 obtain_tag

name: TAG X

-> EXP X

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

2.2.1.4 Binding: Discussion

The following constructs introduce TAGs:

identify
variable
conditional
repeat
labelled
make_proc
make_tagdef
make_id_tagdec
make_var_tagdec

During the evaluation of each of these constructs (apart from *make_id_tagdec* and *make_var_tagdec*) a value, *v*, is produced which is bound to a TAG, *t*, during the evaluation of an EXP. The TAG is in scope during the evaluation of the EXP. This means that during the evaluation of the EXP, evaluation of *obtain_tag(t)* (or *obtain_nl_tag(e,s,t)*) will produce the value *v*. The behaviour of *make_id_tagdec* and *make_var_tagdec* is different: the TAGs which they introduce are in scope throughout the CAPSULE which contains them.

Each of the values introduced in a TDF UNIT will be named by a different TAG, and so no scope rules are needed. (UNITs are self-contained collections of declarations and definitions of TOKENs and TAGs. See §2.17 for details.)

2.2.2 Integers and Arithmetic

2.2.2.1 make_int

v: VARIETY,
value: SIGNED_NAT

-> EXP INTEGER(v)

An integer value is delivered whose value is given by *value*, and whose VARIETY is given by *v*. The integer value *value* will lie between the bounds of *v*. This ensures that *value* is representable as an integer of VARIETY *v*.

2.2.2.2 plus

ov_err: ERROR_TREATMENT,
arg1: EXP INTEGER(V),
arg2: EXP INTEGER(V)

-> EXP INTEGER(V)

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. The sum of *a* and *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in VARIETY *V*, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore* and the VARIETY, *V*, is non-negative, the calculation is performed modulo 2^bits(*V*).

If *ov_err* is *ignore* and the VARIETY is negative, the effect of overflow is undefined.

2.2.2.3 minus

ov_err: ERROR_TREATMENT,
arg1: EXP INTEGER(V),
arg2: EXP INTEGER(V)

-> EXP INTEGER(V)

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. The difference of *a* and *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in VARIETY *V*, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore* and the VARIETY, *V*, is non-negative, the calculation is performed modulo 2^bits(*V*).

If *ov_err* is *ignore* and the VARIETY is negative, the effect of overflow is undefined.

2.2.2.4 mult

```
 ov_err: ERROR_TREATMENT,
 arg1: EXP INTEGER(V),
 arg2: EXP INTEGER(V)

   -> EXP INTEGER(V)
```

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. The product of *a* and *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in VARIETY *V*, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore* and the VARIETY, *V*, is non-negative, the calculation is performed modulo 2^bits(*V*).

If *ov_err* is *ignore* and the VARIETY is negative, the effect of overflow is undefined.

Translators should if possible optimise multiplication by powers of 2 and any relevant constants.

2.2.2.5 Kinds of Division: Discussion

Two classes of division (D) and remainder (M) construct are defined. The two classes have the same definition if both operands have the same sign. Neither is defined if the second argument is zero.

Class 1:

$p \, D1 \, q = n$

where $p = n*q + (p \, M1 \, q)$
$\mathrm{sign}(p \, M1 \, q) = \mathrm{sign}(q)$
$0 \leq |p \, M1 \, q| < |q|$

Class 2:

$p \, D2 \, q = n$

where $p = n*q + (p \, M2 \, q)$
$\mathrm{sign}(p \, M2 \, q) = \mathrm{sign}(p)$
$0 \leq |p \, M2 \, q| < |q|$

2.2.2.6 div1

```
ov_err: ERROR_TREATMENT,
div0_err: ERROR_TREATMENT,
arg1: EXP INTEGER(V),
arg2: EXP INTEGER(V)

  -> EXP INTEGER(V)
```

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. *a* D1 *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in VARIETY *V*, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore* and the VARIETY is negative, the effect of overflow is undefined.

If *b* is zero a divide-by-zero error is caused and handled in the way specified by *div0_err*. If *div0_err* is *ignore* its effect is undefined.

Translators should if possible optimise division by constants, especially powers of 2.

2.2.2.7 div2

```
ov_err: ERROR_TREATMENT,
div0_err: ERROR_TREATMENT,
arg1: EXP INTEGER(V),
arg2: EXP INTEGER(V)

  -> EXP INTEGER(V)
```

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. *a* D2 *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in VARIETY *V*, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore* and the VARIETY is negative, the effect of overflow is undefined.

If *b* is zero a divide-by-zero error is caused and handled in the way specified by *div0_err*. If *div0_err* is *ignore* its effect is undefined.

Translators should if possible optimise division by constants, especially powers of 2. This is possible if *V* is non-negative.

2.2.2.8 mod

```
div0_err: ERROR_TREATMENT,
arg1: EXP INTEGER(V),
arg2: EXP INTEGER(V)

  -> EXP INTEGER(V)
```

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. *a* M1 *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If *b* is zero a divide-by-zero error is caused and handled in the way specified by *div0_err*. If *div0_err* is *ignore* its effect is undefined.

Translators should if possible optimise modulus by powers of 2.

2.2.2.9 rem2

```
divO_err: ERROR_TREATMENT,
arg1: EXP INTEGER(V),
arg2: EXP INTEGER(V)

  -> EXP INTEGER(V)
```

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. *a* M2 *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If *b* is zero a divide-by-zero error is caused and handled in the way specified by *div0_err*. If *div0_err* is *ignore* its effect is undefined.

2.2.2.10 exact_divide

```
arg1: EXP INTEGER(V),
arg2: EXP INTEGER(V)

  -> EXP INTEGER(V)
```

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. The quotient of *a* and *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments. *b* will be an exact divisor of *a*.

2.2.2.11 negate

```
ov_err: ERROR_TREATMENT,
arg: EXP INTEGER(V)

  -> EXP INTEGER(V)
```

arg is evaluated to produce an integer value, *a*. The negation of *a* is delivered as the result of the construct, which has the same SHAPE as the construct's argument.

If the result cannot be expressed in VARIETY *V*, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore*, the effect of overflow is undefined.

2.2.2.12 abs

ov_err: ERROR_TREATMENT,
arg: EXP INTEGER(V)

-> EXP INTEGER(V)

arg is evaluated to produce an integer value, *a*. The absolute value of *a* is delivered as the result of the construct, which has the same SHAPE as the construct's argument.

If the result cannot be expressed in VARIETY *V*, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore*, the effect of overflow is undefined.

2.2.2.13 Number Conversion: Discussion

There is no automatic conversion between integer VARIETYs.

Conversions between integer VARIETYs are carried out by *change_var*. In every case, if the same integer is expressible in the destination VARIETY, this integer expressed in the destination VARIETY is the result.

Certain other conversions are provided which are easy to implement in 2's complement machines, and possible in other representations.

When a negative signed integer is converted to a non-negative VARIETY whose *maxint* is greater than both the modulus of the *minint* and the *maxint* of the source VARIETY, the resulting value is obtained by adding one more than the *maxint* of the target VARIETY.

When an integer is converted to a non-negative VARIETY with *maxint* less than either the modulus of the *minint* or the *maxint* of the source VARIETY, the result is the remained (M1) on division by the number one greater than the *maxint* of the target VARIETY.

All other conversions are target-defined.

2.2.2.14 change_var

w: VARIETY,
arg: EXP INTEGER(V)

-> EXP INTEGER(w)

arg is evaluated to produce an integer value, *a*. If *a* is expressible in VARIETY *w*, then it is delivered as the result of the construct. The result has the SHAPE INTEGER(*w*).

Certain other special target-dependent conversions are defined in §2.2.2.13. No other conversions are defined.

2.2.2.15 shift_left

ov_err: ERROR_TREATMENT,
arg1: EXP INTEGER(V1),
arg2: EXP INTEGER(V2)

-> EXP INTEGER(V1)

arg1 and *arg2* are evaluated to produce values *a* and *places*. The result is equivalent to:

```
if places < 0
then div1(ov_err, impossible, a, $2^{-\text{places}}$)
else mult(ov_err, a, $2^{\text{places}}$)
```

Translators should optimise cases where the number of shifts is a constant.

2.2.2.16 shift_right

```
ov_err: ERROR_TREATMENT,
arg1: EXP INTEGER(V1),
arg2: EXP INTEGER(V2)

  -> EXP INTEGER(V1)
```

arg1 and *arg2* are evaluated to produce values *a* and *places*. The result is equivalent to:

```
if places > 0
then div1(ov_err, impossible, a, $2^{\text{places}}$)
else mult(ov_err, a, $2^{-\text{places}}$)
```

Translators should optimise the cases where the number of shifts is a constant.

2.2.2.17 round

```
ov_err: ERROR_TREATMENT,
v: VARIETY,
arg: EXP FLOAT(F)

  -> EXP INTEGER(v)
```

arg is evaluated to produce a floating point value, *a*. If the nearest integer to *a* is expressible in VARIETY *v*, then a value of that integer is created and delivered.

However, if that nearest integer cannot be expressed in VARIETY *v*, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore* and the VARIETY, *v*, is non-negative, the calculation is performed modulo 2^bits(*v*).

If *ov_err* is *ignore* and the VARIETY is negative, the effect of overflow is undefined.

2.2.2.18 truncate

```
ov_err: ERROR_TREATMENT,
v: VARIETY,
arg: EXP FLOAT(F)

  -> EXP INTEGER(v)
```

arg is evaluated to produce a floating point value, *a*. If the integer part of *a* is expressible in VARIETY *v*, then a value of that integer is created and delivered.

However, if that nearest integer cannot be expressed in VARIETY *v*, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore* and the VARIETY, *v*, is non-negative, the calculation is performed modulo 2^bits(*v*).

If *ov_err* i *ignore* and the VARIETY is negative, the effect of overflow is undefined.

2.2.2.19 bits_to_integer

```
v: VARL TY,
ov_err: E ROR_TREATMENT,
arg: EXP NC (INTEGER(0,1), N)

  -> EXP INTEGER(v)
```

arg is evaluated to produce an NOF(INTEGER(0,1), N) value, *r*. This value is converted to an integer, *a*, of VARIETY *v*, which is delivered.

The manner in which *a* is calculated depends on the VARIETY *v*. If *v* is a non-negative VARIETY - ie. its lower bound is greater than or equal to zero - *a* is derived as follows:

$$\Sigma_{i=0}^{N} r_i * .^{i}$$

However, if *v* is a negative VARIETY - ie. its lower bound is less than zero - *a* is derived as follows:

$$\begin{array}{l} (\textit{if } r_N = 1 \\ \textit{then } -1 \\ \textit{else } 1) * \Sigma_{i=0}^{N-1} r_i * 2^i \end{array}$$

If *a* cannot be expressed in the VARIETY *v*, an overflow error is caused and handled in the way specified by *ov_err*.

If *ov_err* is *ignore* and the VARIETY *v* is non-negative, the calculation is performed modulo 2^bits(*v*).

If *ov_err* is *ignore* and the VARIETY *v* is negative, the effect of overflow is undefined.

2.2.2.20 div_rem1

```
ov_err: ERROR_TREATMENT,
div0_err: ERROR_TREATMENT,
arg1: EXP INTEGER(V),
arg2: EXP INTEGER(V)

  -> EXP TUPLE(INTEGER(V), INTEGER(V))
```

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. A TUPLE of (*a* D1 *b*, *a* M1 *b*) is delivered as the result.

If the result cannot be expressed in the VARIETY *V*, an overflow error is caused and handled in the manner specified by *ov_err*. This only occurs for negative VARIETYs in the special case of dividing *minint* by -1.

If *ov_err* is *ignore* and the VARIETY is negative, the effect of overflow is undefined.

If *b* is zero a divide-by-zero error is caused and handled in the way specified by *div0_err*. If *div0_err* is *ignore* its effect is undefined.

2.2.2.21 div_rem2

```
ov_err: ERROR_TREATMENT,
div0_err: ERROR_TREATMENT,
arg1: EXP INTEGER(V),
arg2: EXP INTEGER(V)

  -> EXP TUPLE(INTEGER(V), INTEGER(V))
```

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same VARIETY. A TUPLE of (*a* D2 *b*, *a* M2 *b*) is delivered as the result.

If the result cannot be expressed in the VARIETY *V*, an overflow error is caused and handled in the manner specified by *ov_err*. This only occurs for negative varieties in the special case of dividing *minint* by -1.

If *ov_err* is *ignore* and the VARIETY is negative, the effect of overflow is undefined.

If *b* is zero a divide-by-zero error is caused and handled in the way specified by *div0_err*. If *div0_err* is *ignore* its effect is undefined.

2.2.2.22 integer_test

ntest: NTEST,
bad: LABEL TOP,
arg1: EXP INTEGER(V),
arg2: EXP INTEGER(V)

-> EXP TOP

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same integer VARIETY. These values are compared using the test *ntest*. If the test succeeds, the construct delivers a value of SHAPE TOP. If it fails, control passes to the LABEL *bad* with a value of SHAPE TOP. Since the only way in which the construct can deliver a result is when the test succeeds, the SHAPE of the result of the construct is itself TOP.

To give an example, if *ntest* is *greater*, then if *a* is greater than *b* the construct delivers a value of SHAPE TOP. If *a* is not greater than *b* is false, control passes to the LABEL *bad*.

2.2.2.23 integer_test_i

ntest: NTEST,
var: VARIETY,
arg1: EXP INTEGER(V),
arg2: EXP INTEGER(V)

-> EXP INTEGER(var)

arg1 and *arg2* are evaluated to produce integer values, *a* and *b*, of the same integer VARIETY. These values are compared using the test *ntest*. If the test succeeds, 1 is delivered. Otherwise, 0 is delivered. The SHAPE of the result is INTEGER(*var*). *var* will accommodate the values 0 and 1.

2.2.2.24 integer_to_bits

arg: EXP INTEGER(V)

-> EXP NOF(BIT, n)

arg1 is evaluated to produce an integer value *a*. A value *r* of SHAPE NOF(BIT, *n*) is created and delivered, where *n* shall be the smallest number of bits required to represent the full (ie. *minint* to *maxint*) range of values in INTEGER(V).

The value *r* is chosen so that if *a* is non-negative

$$a = \Sigma_{i=0}^{n-1} r_i * 2^i$$

.. and if *a* is negative

$$a = \Sigma_{i=0}^{n-1} r_i * 2^i - \text{maxint}(V) - 1$$

On twos-complement machines, translators should not need to generate any code to implement this construct.

2.2.2.25 Character Sets: Discussion

TDF, as a representation of program, does not manipulate characters explicitly. Instead, they are represented by integers. Conventions for mapping characters onto integers are required.

Characters appear in programs, and need to correspond to the characters which appear on the printers and displays of target machines. But the hardware of target machines can use a multiplicity of different collating sequences for characters. In order to achieve portability of TDF programs it is necessary to choose a standard representation for characters in the TDF itself. Translation to the collating sequence for the hardware devices then should occur only on the point of transmission to those devices.

Since ANSI C is compatible with ASCII and Ada makes it mandatory, TDF standardises on ASCII.

Other character sets, such as Japanese, may need to be represented as strings written in programs. But not all target machines have Japanese printers. To conform with the need for portability of TDF programs a similar standard represention of characters in TDF and translation at the device will be needed, for those programs and target machines which use Japanese characters. Multi-byte characters will probably be used. Similar standards are needed for all such character sets. These will have to be standardised as the need arises.

The customisation of user's programs to give messages in the user's own language can be achieved by tokenising the messages (or the collection of messages) and making the substitutions during installation of the program.

2.2.3 Floating Point Values

2.2.3.1 make_floating

f: FLOATING_VARIETY,
sign: BOOL,
mantissa: STRING,
base: NAT,
exponent: SIGNED_NAT

-> EXP FLOAT(f)

mantissa will be a STRING of characters, each of which is either ASCII's decimal point symbol or is greater than or equal to ASCII's zero. It will be readable in base *base*.

The BOOL *sign* determines the sign of the value to be delivered. If it is true, the value will be positive: if false, negative.

A floating point value *v* of FLOATING_VARIETY *f* is created and delivered. The value is the nearest to

$$\text{mantissa'} \times (\text{base}^{\text{exponent}})$$

where mantissa' is *mantissa* read in the base *base*, with the sign determined by *sign*.

v will be representable in the FLOATING_VARIETY *f*.

2.2.3.2 floating_plus

ov_err: ERROR_TREATMENT,
arg1: EXP FLOAT(F),
arg2: EXP FLOAT(F)

-> EXP FLOAT(F)

arg1 and *arg2* are evaluated to produce floating point values, *a* and *b*, of the same FLOATING_VARIETY. The sum of *a* and *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in FLOATING_VARIETY *F*, an overflow error is caused and handled in the way specified by *ov_err*. If *ov_err* is *ignore* its effect is undefined.

2.2.3.3 floating_minus

```
ov_err: ERROR_TREATMENT,
arg1: EXP FLOAT(F),
arg2: EXP FLOAT(F)

  -> EXP FLOAT(F)
```

arg1 and *arg2* are evaluated to produce floating point values, *a* and *b*, of the same FLOATING_VARIETY. The difference of *a* and *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in FLOATING_VARIETY *F*, an overflow error is caused and handled in the way specified by *ov_err*. If *ov_err* is *ignore* its effect is undefined.

2.2.3.4 flo[illegible]g_mult

```
ov_err: ERROR_TREATMENT,
arg1: EXP FLOAT(F),
arg2: EXP FLOAT(F)

  -> EXP FLOAT(F)
```

arg1 and *arg2* are evaluated to produce floating point values, *a* and *b*, of the same FLOATING_VARIETY. The product of *a* and *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in FLOATING_VARIETY *F*, an overflow error is caused and handled in the way specified by *ov_err*. If *ov_err* is *ignore* its effect is undefined.

2.2.3.5 floating_div

```
ov_err: ERROR_TREATMENT,
div0_err: ERROR_TREATMENT,
arg1: EXP FLOAT(F),
arg2: EXP FLOAT(F)

  -> EXP FLOAT(F)
```

arg1 and *arg2* are evaluated to produce floating point values, *a* and *b*, of the same FLOATING_VARIETY. The quotient of *a* and *b* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments.

If the result cannot be expressed in FLOATING_VARIETY *F*, an overflow error is caused and handled in the way specified by *ov_err*. If *ov_err* is *ignore* its effect is undefined.

If *b* is zero a divide-by-zero error is produced and handled in the way specified by *div0_err*. If *div0_err* is *ignore* its effect is undefined.

2.2.3.6 floating_negate

```
ov_err: ERROR_TREATMENT,
arg: EXP FLOAT(F)

  -> EXP FLOAT(F)
```

arg1 is evaluated to produce a floating point value, *a*. The negation of *a* is delivered as the result of the construct, which has the same SHAPE as the construct's argument.

If the result cannot be expressed in the FLOATING_VARIETY *F*, an overflow error is caused and handled in the way specified by *ov_err*. If *ov_err* is *ignore* its effect is undefined.

2.2.3.7 float

```
ov_err: ERROR_HANDLER,
f: FLOATING_VARIETY,
arg: EXP INTEGER(V)

  -> EXP FLOAT(f)
```

arg is evaluated to produce an integer value, *a*. An equal floating point value of FLOATING_VARIETY *f* is created and delivered. Any rounding necessary is target-defined.

If the integer value *a* is not representable in FLOATING_VARIETY *f* an overflow error is generated and handled by *ov_err*. If *ov_err* is *ignore* the effect is undefined.

2.2.3.8 change_floating_variety

```
ov_err: ERROR_TREATMENT,
f: FLOATING_VARIETY,
arg: EXP FLOAT(F)

  -> EXP FLOAT(f)
```

arg is evaluated to produce a floating point value, *a*. A floating point value is created and delivered which has FLOATING_VARIETY *f* and is equal to *a*. This conversion is target-defined.

If *a* cannot be expressed in FLOATING_VARIETY *f*, an overflow error is caused and handled in the way specified by *ov_err*. If *ov_err* is *ignore* its effect is undefined.

2.2.3.9 floating_test

```
ntest: NTEST,
bad: LABEL TOP,
arg1: EXP FLOAT(F),
arg2: EXP FLOAT(F)

  -> EXP TOP
```

arg1 and *arg2* are evaluated to produce floating point values, *a* and *b*, of the same FLOATING_VARIETY. These values are compared using the test *ntest*. If the test succeeds the construct delivers a value of SHAPE TOP. If it fails, control passes to the LABEL *bad* with a value of SHAPE TOP. Since the only way in which the construct can deliver a result is when the test succeeds, the SHAPE of the result of the construct is itself TOP.

To give an example, if *ntest* is *greater*, then if *a* is greater than *b* the construct delivers a value of SHAPE TOP. If *a* is not greater than *b* is false, control passes to the LABEL *bad*.

2.2.3.10 floating_test_i

```
ntest: NTEST,
var: VARIETY,
arg1: EXP FLOAT(F),
arg2: EXP FLOAT(F)

  -> EXP INTEGER(var)
```

arg1 and *arg2* are evaluated to produce floating point values, *a* and *b*, of the same FLOATING_VARIETY. These values are compared using the test *ntest*. If the test succeeds, 1 is delivered. Otherwise, 0 is delivered. The SHAPE of the result is INTEGER(*var*). *var* will accommodate the values 0 and 1.

2.2.4 POINTERs

2.2.4.1 POINTERs: Discussion

Before describing the constructs which manipulate POINTERs, it is useful to introduce three important concepts - sharing, null POINTERs and original POINTERs

2.2.4.1.1 Sharing

Sharing is a concept which relates only to POINTERs. If a POINTER, *a*, points to a space, *a_space*, and a POINTER, *b*, points to a space, *b_space*, and *a_space* and *b_space* overlap, then *a* and *b* are said to share. In other words, if an assignment to *b* can change the result of inspectng the contents of *a*, or vice versa, then *a* and *b* share.

2.2.4.1.2 Null POINTERs

Null POINTERs are required in order to provide a suitable value to put at the end of a list and for similar puposes. Any attempt to obtain the contents of a null POINTER, or to use it as the destination in an assign construct, is defined to produce a detectable error.

If *add_to_ptr* or *subtract_from_ptr* are applied to a null POINTER the effect is undefined.

Null POINTERs cannot share.

2.2.4.1.3 Original POINTERs

A POINTER is an original POINTER if it is created by an evaluation of *variable*, a procedure application (which involves the creation of a POINTER to the parameter) or a library routine which delivers a fresh POINTER.

A POINTER is said to be derived from an original POINTER if and only if it is either a copy of that POINTER or obtained from it by a succession of the following constructs:

add_to_ptr
ptr_field
subtract_from_ptr

Every POINTER is derived from just one original POINTER.

2.2.4.2 add_to_ptr

```
ptr: EXP POINTER(X),
off: EXP OFFSET(X,Y)

  -> EXP POINTER(Y)
```

ptr is evaluated to produce a POINTER *p* and *off* to produce an OFFSET value *o*. A POINTER is created and delivered which points to space for a value of SHAPE Y offset ahead by *o* from the space pointed to by *p*. If *p* is null, the result is undefined.

2.2.4.3 subtract_from_ptr

ptr: EXP POINTER(X),
off: EXP OFFSET(Y,X)

-> EXP POINTER(Y)

ptr is evaluated to produce a POINTER *p* and *off* to produce an OFFSET value *o*. A POINTER is created and delivered which points to space for a value of SHAPE Y offset back by *o* from the space pointed to by *p*. If *p* is null, the result is undefined.

2.2.4.4 ptr_field

ptr_tuple: EXP POINTER(TUPLE $\Pi_{i=1}^{n} S_i$),
{$n \geq 1$} {$1 \leq$ component $\leq n$}
{S_i .. $S_{component}$ are SOME-free}
component: NAT

-> EXP POINTER $S_{component}$

ptr_tuple is evaluated to produce a POINTER *p* to a space, *sp*, containing a value of SHAPE POINTER TUPLE(..). A POINTER to the *component*-th value from the TUPLE in the space *sp* is created and delivered. The result shares with *p*.

(A POINTER(PARAM_PACK(..)) may also be supplied, with the same effect.)

If *p* is a null POINTER, then so is the result. However, they need not be equal null POINTERs.

2.2.4.5 ptr_unpad

component_shape: SHAPE,
ptr: EXP POINTER(X)

-> EXP POINTER(component_shape)

ptr is evaluated to produce a POINTER *p* to a space, *sp*, containing a UNION of SHAPE *arg_shape*. A POINTER(*component_shape*) pointing to that space within *sp* which contains the component value of SHAPE *component_shape* is created and delivered. The result and *p* share.

If *p* is a null POINTER, then so is the result. However, they need not be equal null POINTERs.

2.2.4.6 assign

```
err: ERROR_TREATMENT,
ptr: EXP POINTER(X),
val: EXP Y,              { Y will be an initial segment of X }
no_overlap: BOOL

  -> EXP TOP
```

ptr and *val* are evaluated to produce values, *p* and *v*. The POINTER, *p*, will not be volatile in the sense of ANSI C. The value *v* is put into the space pointed to by *p*. If *p* is a null POINTER then a *null_pointer* error occurs which is handled as specified by *err*. If *err* is *ignore* its effect is undefined.

no_overlap indicates whether or not *v* and the space pointed to by *p* may overlap. If it is true, they will not: if it is false, they may. In the case where they may overlap, translators shall implement the move to have the same effect as if they did not overlap.

If the space to which *p* points does not lie wholly within the space pointed to by the original POINTER from which *p* is derived, the effect is undefined.

2.2.4.7 Initial Segments: Discussion

The definition of *assign* refers to the possibility that one SHAPE may be an "initial segment" of another. This concept applies to three SHAPE constructors:

TUPLE
PARAM_PACK
NOF

The rules are:

TUPLE(X,Y ..) is an initial segment of TUPLE(X,Y,Z ..).

PARAM_PACK(X,Y ..) is an initial segment of PARAM_PACK(X,Y,Z ..).

X is an initial segment of TUPLE(X,Y).

X is an initial segment of PARAM_PACK(X,Y).

NOF(X,m) is an initial segment of NOF(X,n), where m=<n.

X is an initial segment of NOF(X,n), where n>0.

X is an initial segment of X.

"is an initial segment of" is a transitive relation.

2.2.4.8 contents

```
is_null:ERROR_TREATMENT,
sh: SHAPE,              { sh will be SOME-free }
ptr: EXP POINTER(X)        { sh will be an initial segment of X }

  -> EXP sh
```

ptr is evaluated to produce a value *p*. The POINTER *p* will not be volatile in the sense of ANSI C. The content of the space pointed to by *p* is delivered as the result. If *p* is a null POINTER, then a *null_pointer* error is caused and handled according to *is_null*. If *is_null* is *ignore*, the effect is undefined.

If the space to which *p* points does not lie wholly within the space pointed to by the original POINTER from which *p* is derived, the effect is undefined.

2.2.4.9 coerce_ptr_to_initial_segment

```
ptr: EXP POINTER(X),
sh: SHAPE  { sh will be an initial segment of X }

  -> EXP POINTER(sh)
```

ptr is evaluated to produce a POINTER value *p*. The POINTER *p* is delivered as the result of the construct, but now with the SHAPE *sh*. The result is equal to *p*.

2.2.4.10 assign_to_volatile

```
err: ERROR_TREATMENT,
ptr: EXP POINTER(X),
val: EXP Y,             { Y will be an initial segment of X }
no_overlap: BOOL

  -> EXP TOP
```

ptr and *val* are evaluated to produce values, *p* and *v*. The POINTER, *p*, will be volatile in the sense of ANSI C. The value v is put into the space pointed to by p. If *p* is a null POINTER then a *null_pointer* error occurs which is handled as specified by *err*. If *err* is *ignore* its effect is undefined.

no_overlap indicates whether or not *v* and the space pointed to by *p* may overlap. If it is true, they will not: if it is false, they may. In the case where they may overlap, translators shall implement the move to have the same effect as if they did not overlap.

If the space to which *p* points does not lie wholly within the space pointed to by the original POINTER from which *p* is derived, the effect is undefined.

2.2.4.11 contents_of_volatile

```
is_null:ERROR_TREATMENT,
sh: SHAPE,              { sh will be SOME-free }
ptr: EXP POINTER(X)     { sh will be an initial segment of X }

  -> EXP sh
```

ptr is evaluated to produce a POINTER value *p*. The POINTER *p* will be volatile in the sense of ANSI C. The content of the space pointed to by *p* is delivered as the result. If *p* is a null POINTER, then a *null_pointer* error is caused and handled according to *is_null*. If *is_null* is *ignore*, the effect is undefined.

If the space to which *p* points does not lie wholly within the space pointed to by the original POINTER from which *p* is derived, the effect is undefined.

2.2.4.12 move_some

```
no_overlap: BOOL,
ptr1: EXP POINTER(SOME(X)),
ptr2: EXP POINTER(SOME(X)),
off: EXP OFFSET(X,X)

  -> EXP TOP
```

ptr1, *ptr2* and *off* are evaluated to produce POINTER values *p1* and *p2* and an OFFSET value *o*. A quantity of data in the space pointed to by *p1* is moved to the space pointed to by *p2*. The data shifted lies between the start of the space pointed to by *p1* and the value offset from the start by *o*.

no_overlap indicates whether or not the source and destination spaces may overlap. If it is true, they will not: if it is false, they may. In the case where they may overlap, translators shall implement the move to have the same effect as if they did not overlap.

If the space to which *p2* points does not lie wholly within the space pointed to by the original POINTER from which it is derived, the effect is undefined.

2.2.4.13 pointer_test

```
test: NTEST,
bad: LABEL TOP,
ptr1: EXP POINTER(X),
ptr2: EXP POINTER(X)

  -> EXP TOP
```

ptr1 and *ptr2* are evaluated to produce POINTER values, *p1* and *p2*. These values are compared using the test specified by *test*. If the test succeeds, the construct delivers a value of SHAPE TOP. If the test fails, control passes to the LABEL *bad* with a value of SHAPE TOP.

Since the only way in which *pointer_test* can deliver a result is when the test succeeds, the SHAPE of the result of *pointer_test* is itself TOP.

The meaning in this context of the NTESTs *equal* and *not_equal* is straightforward. But the meaning of *greater_than* (and by extension all the others) requires definition. Given a POINTER(X) value, *p*, the value delivered by:

```
add_to_ptr(p,
           array_element_offset(X)
          )
```

is greater than *p*.

If *p1* and *p2* do not share, the effect is implementation defined.

2.2.4.14 pointer_test_i

```
test: NTEST,
var: VARIETY,
ptr1: EXP POINTER(X),
ptr2: EXP POINTER(X)

  -> EXP INTEGER(0,1)
```

ptr1 and *ptr2* are evaluated to produce POINTER values, *p1* and *p2*. These values are compared using the test specified by *test*. If the test succeeds, 1 is delivered. Otherwise 0 is delivered. The SHAPE of the result is INTEGER(*var*). *var* will accommodate the values 0 and 1.

The meaning of the NTEST is as defined under *pointer_test*.

If *p1* and *p2* do not share, the effect is implementation defined.

2.2.4.15 subtract_ptrs

```
ptr1: EXP POINTER(X),
ptr2: EXP POINTER(X)

  -> EXP OFFSET(X,X)
```

ptr1 and *ptr2* are evaluated to produce POINTER values, *p1* and *p2*. If *p1* and *p2* share, then the OFFSET of *p1* from *p2* is delivered as the result.

If *p1* and *p2* do not share, the effect is undefined.

2.2.4.16 ptr_is_null

```
not_null: LABEL TOP,
ptr: EXP POINTER(X)

  -> EXP TOP
```

ptr is evaluated to produce a POINTER value, *p*. If *p* is found to be a null POINTER, the construct delivers a value of SHAPE TOP. If it is not a null POINTER, control passes to the LABEL *not_null* with a value of SHAPE TOP.

2.2.4.17 ptr_not_null

```
is_null: LABEL TOP,
ptr: EXP POINTER(X)

  -> EXP TOP
```

ptr is evaluated to produce a POINTER value, *p*. If *p* is found not to be a null POINTER the construct delivers a value of SHAPE TOP. If it is a null POINTER, control passes to the LABEL *is_null* with a value of SHAPE TOP.

2.2.4.18 Lifetimes: Discussion

This is a convenient point at which to introduce the concept of **lifetime** and discuss its importance to writers of TDF translators.

A danger in ANSI C and other languages is the use of a pointer to a space which is no longer "alive", meaning that the space pointed to is on a stack and has been re-used for some other purpose. Such mistakes can be very hard to detect. Like ANSI C, TDF permits this mistake to be made and specifies that the effect is undefined.

TDF defines the lifetime of POINTERs arising from *variable* constructs to extend over the body of the constructs. Any use of a POINTER **inside** the body of the *variable* construct which gave rise to it has a defined effect: but the effect of any use of it **outside** is undefined. Remember that "undefined" means that translators may refuse to translate the TDF in question or produce code with any effect. Producer writers need to bear in mind that TDF translators are not obliged to police the correctness of their use of POINTERs. It is unlikely that translators will refuse to translate pieces of TDF which involve the use of POINTERs in an undefined fashion because, as we remarked above, such usage can be very hard to detect: instead, it is likely that translators will without warning produce "meaningless" code.

The parameter POINTER created on procedure application behaves similarly: its lifetime extends over the body of the PROC being applied.

Library routines and other programs which are linked in may deliver POINTER results. The lifetime of these POINTERs depends entirely on the content of the programs which deliver them. If the POINTER delivered is the same as a POINTER parameter, then it will have the same lifetime as the parameter: but if the POINTER was generated afresh by the program, then it may have any lifetime - eg. until it is explicitly deallocated by another library routine.

The lifetime rules for the POINTERs arising from *variable* constructs and procedure application permit a conventional stack implementation for TDF, although they do not mandate it.

2.2.5 Procedures

2.2.5.1 Procedures: Discussion

The treatment of procedures varies considerably from language to language. TDF's procedure constructs have been designed in order to cater for a wide range of languages. However, because the ANSI C, C++, FORTRAN 77, COBOL and Pascal procedure mechanisms are not as demanding as those of some other languages (eg. Ada, ML etc.), the subset of TDF described here does not contain the full range of procedure constructs offered by TDF.

All languages' treatments of procedures have one thing in common - a procedure call is a means of applying the same piece of program to different pieces of data. And so the TDF *make_proc* construct allows one to specify a TAG as the formal parameter and to state its SHAPE; the scope of that TAG is the EXP body of the procedure.

But languages differ over treating procedures as data-objects in their own right. Pascal allows one procedure to be a parameter of another, but does not allow assignment of procedure values or the delivery of a procedure as the result of another. ANSI C allows both of these but restricts the declaration of procedures to a global level. In some other languages the use of procedures as first-class data objects is of the essence and provides a very effective means of data encapsulation.

However, despite this diversity of approaches, all these languages agree that the code of a procedure is constant: the most that can vary is the parameter and the non-local values. TDF's concept of procedures reflects this. Procedures will be constructed and named only in TAGDEFs. They will not be constructed anywhere else. First-class procedure values are therefore implemented by binding the relevant *obtain_tag(t)* to a value representing the non-locals, and supplying those non-locals as part of the parameter when the procedure is called. The non-local access required by ANSI C, C++, FORTRAN 77 and Pascal is less demanding: all that is required is that one procedure be able dynamically to obtain a value from the workspace of another procedure. The *obtain_nl* construct, specified below, achieves this.

2.2.5.2 make_proc

```
local: BOOL,
param_shape: SHAPE,          { param_shape will be SOME-free }
param: TAG POINTER(param_shape),
body: EXP BOTTOM

  -> EXP PROC
```

Evaluation of *make_proc* delivers a PROC. When this procedure is applied to a parameter using *apply_proc*, space is allocated to hold a value of SHAPE *param_shape*. The value produced by the parameter, which will be of the correct SHAPE, is used to initialise it. *body* is evaluated. During the evaluation, *param* is bound to an original POINTER pointing to the space. This means that evaluation of *obtain_tag(param)* will produce that pointer.

The SHAPE of *body* will be BOTTOM. This implies that if its evaluation terminates it will be with the evaluation of either a *return* construct or a *jump_nl*.

The TAG used for *param* will not be re-used within the current UNIT. No rules for the effect of the hiding of one TAG by another, equal TAG are given; this will not happen. See §2.2.1.4 for a discussion of this point.

The only TAGs in scope within *body* are the TAGs declared in the CAPSULE which contains the procedure, *param*, and those TAGs introduced inside *body* itself. No other TAGs will be used.

The BOOL, *local*, gives information about possible external access to *param*: if true, *param* will not be supplied as an argument to *obtain_nl_tag* - ie. there will be no non-local access to the value which *param* names. This information is of value in deciding whether certain optimisations are possible.

If a programming language permits more than one parameter, the compiler to TDF will use *make_proc* to construct a TDF procedure whose *param_shape* is PARAM_PACK(..).

The *make_proc* construct will appear only as the EXP in a TAGDEF.

2.2.5.3 make_null_proc

```
  -> EXP PROC
```

A null PROC is created and delivered. If this PROC is applied, the effect is undefined. The null PROC may be tested for using *proc_is_null* or *proc_not_null*.

2.2.5.4 proc_is_null

not_null: LABEL TOP,
procedure: EXP PROC

-> EXP TOP

procedure is evaluated to produce a PROC value, *p*. If *p* is found to be a null procedure, the construct delivers a value of SHAPE TOP. If it is not a null procedure, control passes to the LABEL *not_null* with a value of SHAPE TOP.

2.2.5.5 proc_not_null

is_null: LABEL TOP,
procedure: EXP PROC

-> EXP TOP

procedure is evaluated to produce a PROC value, *p*. If *p* is found not to be a null procedure, the construct delivers a value of SHAPE TOP. If it is not a null procedure, control passes to the LABEL *is_null* with a value of SHAPE TOP.

2.2.5.6 proc_eq

unequal: LABEL TOP,
proc1: EXP PROC,
proc2: EXP PROC

-> EXP TOP

proc1 and *proc2* are evaluated to produce PROC values. The representations of these PROCs are compared. If they are found to be equal, the construct delivers a value of SHAPE TOP. If they are found to be unequal, control passes to the LABEL *unequal* with a value of SHAPE TOP.

2.2.5.7 proc_neq

equal: LABEL TOP,
proc1: EXP PROC,
proc2: EXP PROC

-> EXP TOP

proc1 and *proc2* are evaluated to produce PROC values. The representations of these PROCs are compared. If they are found to be unequal, the construct delivers a value of SHAPE TOP. If they are found to be equal, control passes to the LABEL *equal* with a value of SHAPE TOP.

2.2.5.8 proc_eq_i

```
var: VARIETY,
proc1: EXP PROC,
proc2: EXP PROC

  -> EXP INTEGER(var)
```

proc1 and *proc2* are evaluated to produce PROC values. The representations of these values are compared. If they are found to be equal, 1 is delivered. Otherwise, 0 is delivered. The SHAPE of the result is INTEGER(*var*). *var* will accommodate the values 0 and 1.

2.2.5.9 apply_proc

```
result_shape: SHAPE,    { result_shape will be SOME-free }
proc: EXP PROC,
arg: EXP PARAM_PACK(..)

  -> EXP result_shape
```

proc and *arg* are evaluated to produce values *p* and *a*. The procedure, *p*, is applied to *a*. The result of this application is delivered as the result of the *apply_proc* construct. It will have SHAPE *result_shape*.

If the SHAPEs of the values delivered by all the *return* constructs in *p* are not all equal to *result_shape*, the effect is undefined.

2.2.5.10 apply_current_proc

```
result_shape: SHAPE,    { result_shape will be SOME-free }
arg: EXP PARAM_PACK(..)

  -> EXP result_shape
```

arg is evaluated to produce a value *a*. The procedure which is currently being evaluated is applied recursively to *a*. The result of this recursive application is delivered as the result of the *apply_current_proc* construct. It will have SHAPE *result_shape*. If the SHAPEs of the values delivered by all the *return* constructs in the current procedure are not all equal to *result_shape*, the effect is undefined.

The *apply_current_proc* construct is provided in order to facilitate the optimisation of recursion by installers.

2.2.5.11 return

with: EXP X

-> EXP BOTTOM

with is evaluated to produce a value *w*. The evaluation of the immediately enclosing procedure ceases and the value *w* is delivered as the procedure's result.

Since the *return* construct can never terminate normally, the SHAPE of its result is *bottom*.

2.2.5.12 current_env

-> EXP ENV(T)

A value of SHAPE ENV(T) is created and delivered. It gives access to the values associated with all TAGs introduced in the current procedure, except those which were introduced with BOOL arguments indicating that they would not be the subject of non-local access; and access to any visible LABELs in the current procedure.

T is the TAG introduced in the TAGDEF which contains the current procedure.

current_env provides a sufficient mechanism for accessing non-local values and performing non-local jumps in the case of languages which do not treat procedures as first-class values.

2.2.5.13 obtain_nl_tag

env: EXP ENV(T),
s: SHAPE,
t: TAG s

-> EXP POINTER(X)

env is evaluated to produce an ENV value, *e*. The TAG, *t*, will be introduced in the procedure to which *e* relates and will name a value of SHAPE *s*. A POINTER to that value is created and delivered. Note that the value is drawn from the particular evaluation of the procedure in which the ENV, *e*, was originally created.

The TAG, *t*, will be introduced with a BOOL argument indicating that it may be the subject of non-local access.

2.2.6 Program Structure and Flow of Control

2.2.6.1 sequence

statements: $\Pi_{i=1}^{n}$ EXP Y_i,   { n > 0 }
result: EXP X

-> EXP X

The EXPs in *statements* are evaluated in order. Then *result* is evaluated. The value delivered by *sequence* is the value produced by *result*. Thus the SHAPE of the value delivered by *sequence* is the same as the SHAPE of the value produced by *result*.

2.2.6.2 Availability of LABELs: Discussion

Labels are made available in the arguments of certain control structure constructs. They are available only in the places specified in the descriptions of these constructs. The constructs are:

conditional
repeat
labelled

During the evaluation of some or all of the arguments of these constructs LABELs are bound to some or all of the arguments This means that during the evaluation of the arguments concerned, the evaluation of *goto(l,e)* will cause control to pass to the program fragment bearing LABEL *l*. Only those LABELs which have been introduced in this way are available for use in an *goto* construct.

Non-local or "long" jumping is made possible by the *make_label_value* construct. This delivers a LABEL_VALUE which enables non-local jumping to its argument LABEL. As with *goto*, LABELs are available to *make_label_value* only within the arguments of the three constructs listed above.

2.2.6.3 Jumping with Values: Discussion

In TDF, when control passes from a *goto* or other construct to a LABEL X, a value is transferred and is bound to a TAG X introduced at the same place as the LABEL X. This value will often be of SHAPE TOP, which means that nothing is being transferred, but sometimes a useful value is involved.

This style of jumping is perfectly natural in computers, although as a matter of fact few programming languages permit values to be transferred in this way. TDF provides the facility for two reasons: firstly to allow for its introduction in future systems and languages; and secondly to provide for optimisation of looping constructs. For example, the *while*, *for* etc. constructs of most programming languages have to achieve their effects by side-effecting variables declared elsewhere. There may be cases where this approach can be optimised to jumping with a value in TDF.

In an ANSI C program, *goto* will generally be used with the EXP *make_top*, and it may well be worthwhile tokenising pieces of program to perform such jumps.

2.2.6.4 case

```
control: EXP INTEGER(V),
branches: $\Pi_{i=1}^{n}$(lower$_i$:SIGNED_NAT,
                upper$_i$: SIGNED_NAT,
                branch$_i$: LABEL TOP
               )                    { n > 0 }
  -> EXP TOP
```

control is evaluated to produce an integer value, *c*. Then *c* is tested to see whether it lies inclusively between each of the *lower*$_i$ and *upper*$_i$, in order. If and when one of these tests succeeds, control immediately passes to the LABEL *branch*$_i$ with a value of SHAPE TOP. If *c* lies between none of the pairs of SIGNED_NATs, the construct delivers a value of SHAPE TOP. Since this is the only way in which *case* can deliver a result, the SHAPE of the result of *case* is itself TOP.

The sets of SIGNED_NATs will be disjoint.

Designers of translators should consider when this construct is best implemented by means of a switch jump and when by means of a succession of tests. In particular, the special case where there is only one branch should be optimised - it may be possible to use a compare against bounds instruction; as well as the case of one branch where the SIGNED_NATs are equal - which could be implemented as a simple comparison.

2.2.6.5 conditional

```
local: (BOOL)_OPTION,
tk: (TAG X)_OPTION,
first: EXP W,
alt_label: LABEL X,
alt: EXP Y

  -> EXP 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. However, if a *goto(alt_lab,exp)* (or any other jump to *alt_lab*) is encountered during the evaluation of *first*, then evaluation of *first* will stop, *alt* will be evaluated and its result, *a*, delivered as the result of the whole construct.

Depending on the run-time behaviour of *first*, the result of the construct may be provided by *first* or by *alt*. The SHAPEs W and Y will either be equal to each other or BOTTOM. If they are both BOTTOM, the SHAPE of the result is BOTTOM. If only one of them is BOTTOM, it is the SHAPE of the other. And if both are equal and non-BOTTOM, it is their SHAPE.

During the evaluation of *alt* the value, *e*, produced by *exp* is bound to *tk*. This means that inside *alt* an evaluation of *obtain_tag(tk)* will produce the value *e*, with SHAPE *sh*. The presence of a TAG *tk* is optional. If a TAG is not supplied, then no binding occurs, and X will be TOP.

The BOOL, *local*, will be supplied if *tk* is supplied. It gives information about possible external access to *tk*: if true, *tk* will not be supplied as an argument to *obtain_nl_tag* - ie. there will be no non-local access to the value which *tk* names. This information is of value in deciding whether certain optimisations are possible.

The TAG used for *tk* will not be re-used within the current UNIT. No rules for the effect of the hiding of one tag by another, equal TAG are given; this will not happen. See §2.2.1.4 for a discussion of this point.

Note that *alt_lab* is not available in *alt*. In consequence this construct cannot be used to provide a loop.

2.2.6.6 repeat

```
local: (BOOL)_OPTION,
tk: (TAG X)_OPTION,
start: EXP X,
repeat_label: LABEL X,
body: EXP Y

  -> EXP Y
```

start is evaluated to produce a value *st* of SHAPE X. Then *body* is evaluated. During this evaluation of *body*, *st* is bound to *tk*. This means that inside *body* an evaluation of *obtain_tag(tk)* will produce the value *st*.

If *body* produces a result, *b*, this is delivered as the result of the whole construct. However, if a *goto(repeat_label,exp)* (or any other jump to *repeat_label*) is encountered during the evaluation of *body*, then the evaluation of *body* stops. *body* is then evaluated afresh.

During this new evaluation, the value, *e*, produced by *exp* is bound to *tk*. If a TAG is not supplied, then X will be TOP.

The looping behaviour may be repeated indefinitely.

The BOOL, *local*, will be supplied if *tk* is supplied. It gives information about possible external access to *tk*: if true, *tk* will not be supplied as an argument to *obtain_nl_tag* - ie. there will be no non-local access to the value which *tk* names. This information is of value in deciding whether certain optimisations are possible.

The TAG used for *tk* will not be re-used within the current UNIT. No rules for the effect of the hiding of one TAG by another, equal TAG are given; this will not happen. See §2.2.1.4 for a discussion of this point.

2.2.6.7 labelled

starter: EXP X,
branches: $\Pi_{i=1}^{n}$ (sh$_i$: SHAPE,
    branch_label$_i$: LABEL Y$_i$,   { n > 0 }
    local$_i$: (BOOL)_OPTION,
    tk$_i$: (TAG Y$_i$)_OPTION,
    branch$_i$: EXP Z$_i$
)
-> EXP W

starter is evaluated. If its evaluation runs to completion producing a value, *st*, then *st* is delivered as the result of the whole construct. However, if a *goto(branch_label*$_m$*,exp)* (or any other jump to *branch_label*$_m$) is encountered during the evaluation of *starter*, then the evaluation of *starter* stops. *branch*$_m$ is then evaluated. The result of *exp*, *e*, from the *goto* is bound to *tk*$_m$ (if supplied) during this new evaluation. This means that inside *branch*$_m$ an evaluation of *obtain_tag(tk*$_m$*)* will produce the value *e* with SHAPE *sh*$_m$. (If *tk*$_m$ is not supplied, then Y$_m$ will be TOP.) If the evaluation of *branch*$_m$ runs to completion, then the value which it produces, *b*$_m$, is delivered as the result of the whole construct.

However, if a *goto(branch_label*$_n$*,exp)* (or any other jump to *branch_label*$_n$) is encountered during the evaluation of *branch*$_m$, then the evaluation of *branch*$_m$ stops. *branch*$_n$ is then evaluated. (*n* may equal *m*.) As before, the value produced by *exp* is bound with *tk*$_n$ (if supplied) during the evaluation of *branch*$_n$.

Such jumping may continue indefinitely, but if any of the branches' evaluations runs to completion producing a value, *v*, then that value is delivered as the result of the whole construct.

Depending on their run-time behaviour, the result of the construct may be provided by *starter* or one of the *branches*. The SHAPEs of *starter* and the *branches* must all be equal to each other or BOTTOM. If they are all BOTTOM, the SHAPE of the result is BOTTOM. If they are not all BOTTOM, the SHAPE of the result is the non-BOTTOM SHAPE.

The BOOLs, *local*$_i$, will be supplied if *tk*$_i$ are supplied. They give information about possible external access to tk_i: if true, tk_i will not be supplied as an argument to *obtain_nl_tag* - ie. there will be no non-local access to the value which tk_i names. This information is of value in deciding whether certain optimisations are possible.

The TAGs used for tk_i will not be re-used within the current UNIT. No rules for the effect of the hiding of one TAG by another, equal TAG are given; this will not happen. See §2.2.1.4 for a discussion of this point.

2.2.6.8 goto

dest: LABEL X,
with: EXP X

-> EXP BOTTOM

with is evaluated to produce a value *w*. Contuol then passes to the LABEL *dest* with the value *w*. This construct will only be used where the LABEL *dest* is available.

Since the construct can never terminate normally, the SHAPE of its result is *bottom*.

2.2.6.9 make_label_value

lab: LABEL X

-> EXP LABEL_VALUE(X)

A LABEL_VALUE *lv* is created and delivered which can be used as an argument to *goto_nl*. If and when *goto_nl* is evaluated with *lv* as its argument, control will pass to *lab*.

lv could be supplied to another procedure, enabling that procedure to perform a non-local jump back to the current procedure. If *lv* is passed out the scope in which *lab* is available, the effect of an attempt to jump to the LABEL to which it refers is undefined.

2.2.6.10 goto_nl

env: EXP ENV(T),
dest: EXP LABEL_VALUE(X),
with: EXP X

-> EXP BOTTOM

exp, *dest* and *with* are evaluated to produce values *e*, *d* and *w*. The LABEL to which *d* refers will be available in the procedure application to which *e* refers. Control passes to that LABEL with the value *w*.

Since *goto_nl* can never terminate normally, the SHAPE of its result is BOTTOM.

2.2.7 OFFSETs

2.2.7.1 array_element_offset

sh: SHAPE { sh will be SOME-free }

-> EXP OFFSET(*sh*,*sh*)

An OFFSET value is created and delivered which is the offset between two adjacent elements in an array of values of SHAPE *sh*. Because *sh* is SOME-free, the result is determinable at translate-time.

2.2.7.2 tuple_element_offset

sh: SHAPE, { sh will be TUPLE($\Pi_{i=1}^{m} P_i$) }
n: NAT { n =< m }

-> EXP OFFSET (TUPLE($\Pi_{i=1}^{n-1} P_i$),P_n)

An OFFSET value is created and delivered which measures the offset from the beginning of a value of SHAPE TUPLE($\Pi_{i=1}^{n-1} P_i$) to an adjacent value of SHAPE P_n.

2.2.7.3 offset_add

offset1: EXP OFFSET(X,Y),
offset2: EXP OFFSET(Y,Z)

-> EXP OFFSET(X,Z)

offset1 and *offset2* are evaluated to produce OFFSET values *off1* and *off2*. These are offsets between pairs of values of SHAPEs X and Y, and Y and Z respectively. A new OFFSET is created and delivered which is the sum of these two OFFSETs.

2.2.7.4 offset_subtract

offset1: EXP OFFSET(X,Z),
offset2: EXP OFFSET(X,Y)

-> EXP OFFSET(Y,Z)

offset1 and *offset2* are evaluated to produce OFFSET values *off1* and *off2*. These are offsets between pairs of values of SHAPEs X and Z, and X and Y respectively. A new OFFSET is created and delivered which is the difference between these two OFFSETs.

2.2.7.5 offset_mult

offset: EXP OFFSET(X,X),
number: EXP INTEGER(V)

-> EXP OFFSET(X,X)

offset is evaluated to produce an OFFSET value *off* and *number* to produce an integer value *n*. *off* describes the offset between two values in an array of X's which are *diff* elements apart. A new OFFSET is created and delivered which describes the offset between two values in an array of X's which are *diff*n* elements apart.

2.2.7.6 offset_div

ov_err:ERROR_TREATMENT,
v: VARIETY,
offset1: EXP OFFSET(X,X),
offset2: EXP OFFSET(X,X)

-> EXP INTEGER(v)

offset1 and *offset2* are evaluated to produce OFFSET values *off1* and *off2*. *off1* describes the offset between two values in an array of X's which are *diff1* elements apart. *off2* describes the offset between two values in an array of X's which are *diff2* elements apart. The quotient of *diff1* and *diff2* is delivered as the result of the construct, which has the same SHAPE as the construct's arguments. *diff2* will be an exact divisor of *diff1*.

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

2.2.7.7 offset_negate

offset: EXP OFFSET(X,Y)

-> EXP OFFSET(Y,X)

offset is evaluated to produce an OFFSET value *off*. A new OFFSET value is created and delivered which is the negation of *off*. Its SHAPE is the reverse of *off*'s, because the SHAPEs which it offsets from and to are reversed.

2.2.7.8 offset_test

ntest: NTEST,
bad: LABEL TOP,
off1: EXP OFFSET(X,Y),
off2: EXP OFFSET(X,Y)

-> EXP TOP

off1 and *off2* are evaluated to produce OFFSET values, *o1* and *o2*. These values are compared using the test *ntest*. If the test succeeds, the construct delivers a value of SHAPE TOP. If it fails, control passes to the LABEL *bad* with a value of SHAPE TOP.

The meaning in this context of the NTESTs *equal* and *not_equal* is straightforward. But the meaning of the others requires definition.

The value delivered by:

```
offset_add(array_element_offset(X),
           array_element_offset(X)
          )
```

is greater than:

array_element_offset(X).

The value delivered by:

offset_negate(array_element_offset(X))

is less than:

array_element_offset(X).

2.2.7.9 offset_test_i

ntest: NTEST,
var: VARIETY,
off1: EXP OFFSET(X,Y),
off2: EXP OFFSET(X,Y)

-> EXP INTEGER(var)

off1 and *off2* are evaluated to produce OFFSET values, *o1* and *o2*. These values are compared using the test *ntest*. If the test succeeds, 1 is delivered. Otherwise, 0 is delivered. The SHAPE of the result is INTEGER(*var*). *var* will accommodate the values 0 and 1.

The meaning of the NTEST is as defined under *offset_test*.

2.2.8 NOFs and SOMEs

2.2.8.1 make_nof

parts: $\Pi_{i=1}^{n}$ EXP P { n > 0 }

-> EXP NOF(P, n)

The *parts* are evaluated. An NOF is created and delivered which is composed from the values produced, in the same order as they occur in *parts*. Its size is determined by the number of *parts* supplied.

2.2.8.2 n_copies

exp: EXP X,
number: NAT

-> EXP NOF(X, number)

exp is evaluated to produce a value *e*. An NOF value is created and delivered which contains *number* copies of the value *e*.

2.2.8.3 trim_nof

first: NAT,
number: NAT,
arg: EXP NOF(S, N)   { first+number-1 =< N }

-> EXP NOF(S, number)

arg is evaluated to produce an NOF value, *a*. A new NOF value consisting of *number* components from *a*, starting at *first* is created and delivered as the result of *trim_nof*.

2.2.8.4 concat_nof

arg1: EXP NOF(S, M),
arg2: EXP NOF(S, N)

-> EXP NOF(S, M+N)

arg1 and *arg2* are evaluated to produce values *a* and *b* which are NOFs derived from the same SHAPE, *S*. A new value is created and delivered with SHAPE NOF(S,M+N). Its first M components are copies of the components of *a* and the last N components are copies of the components of *b*.

2.2.8.5 and

arg1: EXP S,
arg2: EXP S

-> EXP S   { S = NOF(BIT,N) | INTEGER(V) | BIT }

arg1 and *arg2* have the same SHAPE and that SHAPE is the SHAPE of the result. They may be NOF(INTEGER(0,1),N), INTEGER(V) or BIT. They are evaluated to produce values *a* and *b*. The bit-wise intersection of *a* and *b* is delivered as the result.

2.2.8.6 or

arg1: EXP S,
arg2: EXP S

-> EXP S   { S = NOF(BIT,N) | INTEGER(V) | BIT }

arg1 and *arg2* have the same SHAPE and that SHAPE is the SHAPE of the result. They may be NOF(INTEGER(0,1),N), INTEGER(V) or BIT. They are evaluated to produce values *a* and *b*. The bit-wise union of *a* and *b* is delivered as the result.

2.2.8.7 xor

arg1: EXP S,
arg2: EXP S

-> EXP S { S = NOF(BIT,N) | INTEGER(V) | BIT }

arg1 and *arg2* have the same SHAPE and that SHAPE is the SHAPE of the result. They may be NOF(INTEGER(0,1), N), INTEGER(V) or BIT. They are evaluated to produce values *a* and *b*. The bit-wise exclusive or of *a* and *b* is delivered as the result.

2.2.8.8 not

arg: EXP S

-> EXP S { S = NOF(BIT,N) | INTEGER(V) | BIT }

arg has the same SHAPE as the result. It may be NOF(INTEGER(0,1),N), INTEGER(V) or BIT. It is evaluated to produce a value *a*. The bit-wise negation of *a* is delivered as the result.

2.2.9 TUPLEs, PARAM_PACKs and UNIONs

2.2.9.1 make_tuple

parts: $\Pi_{i=1}^{n}$ EXP P_i

-> EXP TUPLE ($\Pi_{i=1}^{n} P_i$)

The *parts* are evaluated. A TUPLE is created and delivered which is composed from the values produced, in the same order as they occur in *parts*.

2.2.9.2 make_param_pack

parts: $\Pi_{i=1}^{n}$ EXP P_i

-> EXP PARAM_PACK ($\Pi_{i=1}^{n} P_i$)

The *parts* are evaluated. A PARAM_PACK is created and delivered which is composed from the values produced, in the same order as they occur in *parts*.

n may be zero: this case will arise when it is desired to supply no parameters to a procedure.

2.2.9.3 add_to_tuple

tuple: EXP TUPLE ($\Pi_{i=1}^{n} P_i$),
addendum: EXP Q

-> EXP TUPLE ($\Pi_{i=1}^{n} P_i$, Q)

tuple and *addendum* are evaluated to produce values *t* and *a*. A TUPLE having *n*+1 elements is created and delivered which is composed from the elements of *t* followed by *a*.

2.2.9.4 field

component: NAT,
tuple: EXP TUPLE $\Pi_{i=1}^{n} P_i$ { $n \geq 1$ } { $1 \leq$ component $\leq n$ }

-> EXP $P_{component}$

tuple is evaluated to produce a TUPLE value, *t*. The *component*-th field of *t* is delivered as the result of the *field* construct. The SHAPE of the result is the SHAPE of the *component*-th element of *tuple*.

(This construct may also take a PARAM_PACK argument, with the same effect.)

2.2.9.5 pad

union_shape: UNION $\Pi_{i=1}^{n} X_i$, { $n > 0$ }
arg: EXP Y { there will be some k such that $Y = X_k$ }

-> EXP UNION ($\Pi_{i=1}^{n} X_i$)

arg is evaluated to produce a value, *a*. A value of SHAPE *union_shape* is created from *a* and delivered.

2.2.9.6 unpad

alt: SHAPE, { alt will be SOME-free }
union: EXP UNION ($\Pi_{i=1}^{n} P_i$) { n > 1 }
{ there will be some k such that alt = P_k }

-> EXP alt

union is evaluated to produce a value *u*. The SHAPE of *u* will be UNION(..) and one of its components will be *alt*. The value of *u* is then delivered, but now with SHAPE *alt*. If *u* in fact has some other SHAPE, the effect is undefined.

Most translators will not generate any code for this construct. It changes the SHAPE of the expression.

2.2.10 Miscellaneous

2.2.10.1 make_value

sh: SHAPE

-> EXP sh

A value of SHAPE *sh* is created and delivered. The content of the value is not defined. This construct is used to provide the initial value in *variable* declarations when the variable is uninitialised, and in other contexts where the source language does not state what content a value should have.

2.2.10.2 clear_shape

sh: SHAPE

-> EXP sh

A value, *v*, of SHAPE *sh* is created and delivered.

The content of *v* is defined as follows:

if *sh* is BIT, *v* is false;

if *sh* is INTEGER(..), *v* is zero;

if *sh* is FLOATING(..), *v* is zero;

if *sh* is POINTER(..), *v* is a null POINTER;

if *sh* is TUPLE($\Pi_{i=1}^{n} X_i$), *v* is a TUPLE, each of whose elements is the same as the result of *clear_shape(X_i)*;

if *sh* is PARAM_PACK($\Pi_{i=1}^{n} X_i$), *v* is a PARAM_PACK, each of whose elements is the same as the result of *clear_shape(X_i)*;

if *sh* is UNION($\Pi_{i=1}^{n} X_i$), *v* is a UNION; if *v* is subjected to a subsequent *unpad(X_1,v)* the result will be the same as the result of *clear_shape(X_1)*: the result of *unpad* with any SHAPE other than X_1 is undefined;

if *sh* is NOF(X,N), *v* is an NOF containing N values, each of which is the same as the result of *clear_shape(X)*;

if *sh* is OFFSET(X,X), *v* is a nil OFFSET.

sh will not be any SHAPE other than those listed above. *sh* will be SOME-free.

2.2.10.3 make_string

str: STRING,
var: VARIETY

-> EXP NOF(INTEGER(var), L) { L is the length of str }

An EXP holding the string *str* is created and delivered. The SHAPE of the INTEGERs in the NOF is determined by *var*.

2.2.10.4 exp_cond

control: EXP INTEGER(V),
exp1: EXP X,
exp2: EXP X

-> EXP X

control will be a constant evaluable at translate-time. At **translate-time**, it is evaluated to produce a value, *c*. If *c* is non-zero, then *exp1* is selected for translation. If *c* is zero, then *exp2* is selected.

2.2.10.5 Constants: Discussion

The definition of *exp_cond* requires an EXP to be a "constant evaluable at translate-time". For an EXP to satisfy this condition it must be constructed according to the following rules:

it may not contain *obtain_tag(T)* if the TAG, *T*, is not introduced inside the EXP;

it may not contain *assign_to_volatile*;

it may not contain *contents_of_volatile*;

it may not contain *obtain_nl_tag*;

it may not contain *goto(l,..)* if the LABEL, *l*, is not introduced inside the EXP;

it may not contain *make_label_value(l)* if the LABEL, *l*, is not introduced inside the EXP;

it may not contain *goto_nl*;

it may not contain *repeat*;

it may not contain *labelled* constructs where there are jumps between any of the branches;

it may not contain *apply_current_proc*;

it may not contain *return*;

it may not contain *current_env*;

it may not contain *apply_token(T,..)* if the program fragment for which *T* stands contains any EXPs which are not themselves constants evaluable at translate-time.

The combined effect of these rules is to specify that in order to be evaluable at **translate**-time and constant, an EXP must be completely self-contained.

2.2.10.6 make_false

-> EXP BIT

A false BIT is created and delivered.

2.2.10.7 make_true

-> EXP BIT

A true BIT is created and delivered.

2.3 NAT

A value of SORT NAT is a static non-negative integer value of unbounded size.

2.4 SIGNED_NAT

A value of SORT SIGNED_NAT is a static integer value, positive or negative, of unbounded size. There are three constructs:

The first simply comprises an integer.

The second and third are target-defined values - the lowest and highest integers representable in a given VARIETY:

2.4.1 maxint

v: VARIETY

-> SIGNED_NAT

2.4.2 minint

v: VARIETY

-> SIGNED_NAT

2.5 VARIETY

A value of SORT VARIETY describes the different kinds of integer which are available at run-time. There are two constructs:

The first comprises a pair of SIGNED_NATs which indicate the lower and upper bound of integers that must be representable by the integer value at run-time (as discussed in §2.1.2.1):

(SIGNED_NAT, SIGNED_NAT)

The second provides the ability to choose, at translate-time, which of two VARIETYs is to be used:

2.5.1 variety_cond

control: EXP INTEGER(V),
v1: VARIETY,
v2: VARIETY

-> VARIETY

control will be a constant evaluable at translate-time. At **translate-time,** it is evaluated to produce a value, *c*. If *c* is non-zero, then *v1* is selected for translation. If *c* is zero, then *v2* is selected.

variety_cond is used to represent certain uses of ANSI C's "#if".

A VARIETY is said to be "negative" if its lower bound is less than zero, and "non-negative" otherwise.

2.6 FLOATING_VARIETY

A value of SORT FLOATING_VARIETY describes the different kinds of floating point numbers which are available at run-time. There are two constructs:

The first is a cartesian product of two values of SORT NAT and two of SORT SIGNED_NAT.

(NAT, NAT, SIGNED_NAT, SIGNED_NAT)

These give details about the base to be used, the number of digits that must be representable in the mantissa and the minimum and maximum values which the exponent can take (as discussed in §2.1.2.2).

The second provides the ability to choose, at translate-time, which of two FLOATING_VARIETYs is to be used:

2.6.1 floating_variety_cond

```
control: EXP INTEGER(V),
fv1: FLOATING_VARIETY,
fv2: FLOATING_VARIETY

  -> FLOATING_VARIETY
```

control will be a constant evaluable at translate-time. At **translate-time**, it is evaluated to produce a value, *c*. If *c* is non-zero, then *fv1* is selected for translation. If *c* is zero, then *fv2* is selected.

2.7 TAG

A value of SORT TAG is an identifier standing for a run-time value. It is a static non-negative integer of unbounded size.

For ease of exposition, TAGs are frequently qualified in this document by the SHAPE of the value which they stand for. However, this does not mean that SHAPEs feature at all in the representation of TAGs.

2.8 LABEL

A value of SORT LABEL is an identifier which stands for a program location - ie. a destination for jumps.

Like TAGs, LABELs are frequently qualified in this document by a SHAPE. This is the SHAPE of the value which should be passed to the piece of program which the LABEL marks when a jump occurs. However, this does not mean that SHAPEs feature at all in the representation of LABELs.

2.9 NTEST

A value of SORT NTEST identifies one of a number of arithmetic tests. There are six NTEST constructs:

greater_than
greater_than_or_equal
less_than
less_than_or_equal
equal
not_equal

The names are self-explanatory.

2.10 STRING

A value of SORT STRING is a constant string of characters.

2.11 BOOL

A value of SORT BOOL is one which can take only two values. There are two constructs:

true
false

2.12 ERROR_TREATMENT

A value of SORT ERROR_TREATMENT controls program behaviour in the event that a run-time error occurs. There are two constructs:

2.12.1 Impossible

This construct is used when the error cannot occur. For example, if the divide operation is dividing by a constant which is known not to be zero, the div0_err ERROR_TREATMENT should be given the value *impossible*. This permits the translator to avoid creating any code that might have been needed. This argument should be produced by compiler writers whenever possible, since it permits the least and fastest code to be produced.

For example, when translating an arithmetic operation with error treatment *impossible* on VAX, if the program at this point has overflow trap flag set or unset, the trap flag need not be changed.

If the error in question does nevertheless occur, the effect is undefined.

2.12.2 Ignore

This construct is used when the error can occur, but an attempt is to be made to carry on. In some constructs the effect will be undefined: in others a definition is given.

For example, when translating an arithmetic operation with error treatment *ignore* on VAX, if the program at this point has overflow trap set, it will have to be unset.

2.13 CAPSULE

The SORT CAPSULE describes values which are independent pieces of TDF program.

There is only one construct:

2.13.1 make_capsule

unit: UNIT
tok_externs: ($\Pi_{i=1}^{x}$ TOKEXTERN$_i$)_OPTION
tag_externs: ($\Pi_{i=1}^{y}$ TAGEXTERN$_i$)_OPTION

-> CAPSULE

Each EXTERN associates one of the TOKENs or TAGs in the UNIT with an external name. Both *tok_externs* and *tag_externs* are optionally supplied, but at least one EXTERN will be given: otherwise the CAPSULE would be redundant since there would be no means of referring to the program fragments or values defined in its UNIT from outside!

When code produced from a CAPSULE is linked, those TAGs which are the subject of *tag_externs* are eligible for system linking in the normal way: ie. values corresponding to TAGs which are defined in the CAPSULE are made available for external use; and values corresponding to TAGs which are used in the CAPSULE but not defined in it need to be linked in from elsewhere.

(§1.8.1 gives an overview of the structure and function of CAPSULEs.)

2.14 TOKEXTERN

A value of SORT TOKEXTERN expresses the connection between the name by which a program fragment is known inside a CAPSULE - a TOKEN - and a name by which it is to be known outside. There is only one construct:

2.14.1 make_tok_extern

internal: TOKEN,
external: STRING,
u: USAGE

-> TOKEXTERN

A TOKEXTERN is constructed which defines the program fragment known inside a CAPSULE by the name *internal* to be known by the TDF Builder by the name *external*. The USAGE, *u*, indicates whether the TOKEN *internal* is declared, defined and used in the CAPSULE.

The provision of a STRING argument in this construct reflects the TDF Builder's reliance on strings, in harmony with current system linkers. However, the development of new constructs to support other systems of naming is not precluded.

2.15 TAGEXTERN

A value of SORT TAGEXTERN expresses the connection between the name by which a value is known inside a CAPSULE - a TAG - and a name by which it is to be known outside. There is only one construct:

2.15.1 make_tag_extern

internal: TAG,
external: STRING,
u: USAGE

-> TAGEXTERN

A TAGEXTERN is constructed which defines the value known inside a CAPSULE by the name *internal* to be known by the TDF Builder and system linkers by the name *external*. The USAGE, *u*, indicates whether the TAG *internal* is declared, defined and used in the CAPSULE.

The provision of a STRING argument in this construct reflects current linkers' reliance on strings. However, the development of new constructs to support other systems of naming is not precluded.

2.16 USAGE

A value of SORT USAGE indicates whether a TAG or TOKEN is declared, defined and used in a CAPSULE. There are five constructs, which indicate the various possible conditions:

dec
used
dec_def
dec_used
dec_def_used

USAGEs are used to construct TOKEXTERNs and TAGEXTERNs.

2.17 UNIT

A value of SORT UNIT gathers together a number of TOKDECs, TAGDECs, TOKDEFs and TAGDEFs. It places all the TOKDEFs and TAGDEFs in the scope of all the TOKDECs and TAGDECs, enabling them to refer to one another as well as to program fragments and values not defined in the UNIT, but expected to be linked in from without.

There are three constructs:

2.17.1 make_simple_unit

tok: TOKEN,
tokdecs: ($\Pi_{i=1}{}^{w}$ TOKDEC$_i$)_OPTION,
tagdecs: ($\Pi_{i=1}{}^{y}$ TAGDEC$_i$)_OPTION,
tokdefs: ($\Pi_{i=1}{}^{x}$ TOKDEF$_i$)_OPTION,
tagdefs: ($\Pi_{i=1}{}^{z}$ TAGDEF$_i$)_OPTION

-> UNIT

tokdecs and *tagdecs* provide the declarations of TOKENs and TAGs. All these TOKENs and TAGs are in scope in all the *tokdefs* and *tagdefs*, which are the definitions of TOKENs and TAGs.

(The TOKEN *tok* does not relate to any of the TOKENs defined or declared in the UNIT: it serves to identify the UNIT and may be used for diagnostic purposes.)

2.17.2 make_comp_unit

units: $\Pi_{i=1}^{x}$ UNIT$_i$ { i>1 }

-> UNIT

A new UNIT is constructed which declares and defines all the program fragments and values which the members of *units* declare and define. There will be no SORT or SHAPE mismatches - ie. in the case of TOKENs the SORTs will be the same, and in the case of TAGs the SHAPEs will be the same.

make_comp_unit is used by the TDF Builder in the course of constructing a new consolidated CAPSULE. It is not likely to be needed by producer writers, but it nevertheless forms part of the TDF Specification so that the action of the TDF Builder can be represented as a straightforward TDF-to-TDF transformation.

2.17.3 add_linkage

unit: UNIT,
toklinks: ($\Pi_{i=1}^{x}$ TOKLINK$_i$)_OPTION,
taglinks: ($\Pi_{i=1}^{y}$ TAGLINK$_i$)_OPTION

-> UNIT

A new UNIT is constructed from which the program fragments and values declared in *unit* are made visible externally under the names indicated in *toklinks* and *taglinks*. The internal names specified in *toklinks* and *taglinks* must be ones declared in *unit*: otherwise the effect is undefined.

(Although *toklinks* and *taglinks* are optionally supplied, one or other will be supplied. Otherwise, the use of *add_linkage* would be redundant.)

As with *make_comp_unit, add_linkage* is not likely to be needed by producer writers. It is used by the TDF Builder to reconcile internal names in the course of constructing a new consolidated CAPSULE. (§1.8.1 gives the background to this.)

2.18 TOKDEC

A value of SORT TOKDEC declares a TOKEN and is for incorporation into a UNIT. There is one construct:

2.18.1 make_tokdec

tok: TOKEN,
arg_sorts: $(\Pi_{i=1}^{n}$ SORTNAME$_i)$_OPTION,
res_sort: SORTNAME

-> TOKDEC

A TOKDEC announcing that the TOKEN *tok* identifies a fragment of TDF of SORT *res_sort* is constructed. If *arg_sorts* is supplied, the fragment of TDF will be parameterised by *n* argument fragments of TDF whose SORTs are given by *arg_sorts*.

2.19 SORTNAME

A value of SORT SORTNAME denotes a SORT. SORTNAMEs find application in *make_tokdec*, which constructs the declaration of a TOKEN and needs to state what the SORTs of the TOKEN's arguments and result are.

There are 25 constructs - one corresponding to each SORT:

shape_sort	*tokextern_sort*
exp_sort	*tagextern_sort*
nat_sort	*usage_sort*
signed_nat_sort	*unit_sort*
variety_sort	*tokdec_sort*
floating_variety_sort	*sortname_sort*
tag_sort	*tagdec_sort*
label_sort	*tokdef_sort*
ntest_sort	*tagdef_sort*
string_sort	*toklink_sort*
bool_sort	*taglink_sort*
error_treatment_sort	*token_sort*
capsule_sort	

2.20 TAGDEC

A value of SORT TAGDEC declares a TAG and is for incorporation into a UNIT. There are two constructs:

2.20.1 make_id_tagdec

t: TAG,
s: SHAPE

-> TAGDEC

A TAGDEC announcing that the TAG *t* identifies an EXP of SHAPE *s* is constructed. It is **not** capable of being assigned to.

2.20.2 make_var_tagdec

t: TAG,
s: SHAPE

-> TAGDEC

A TAGDEC announcing that the TAG *t* identifies a value of SHAPE POINTER(*s*) is constructed. Being a POINTER, it is capable of being assigned to.

2.21 TOKDEF

A value of SORT TOKDEF gives the definition of a TOKEN, for incorporation into a UNIT. There is one construct:

2.21.1 make_tokdef

tok: TOKEN,
toks: ($\Pi_{i=1}^{x}$ TOKEN)_OPTION,
body: S { S may be any SORT }

-> TOKDEF

A TOKDEF is constructed which defines the TOKEN *tok* to stand for the fragment of TDF, *body*, which may be of any SORT. If *toks* is supplied, then when the TOKEN *tok* comes to be applied, occurrences in *body* of the TOKENs contained in *toks* will be taken to stand for the *x* arguments provided.

2.22 TAGDEF

A value of SORT TAGDEF gives the definition of a TAG, for incorporation into a UNIT. There is one construct:

2.22.1 make_tagdef

tag: TAG,
exp: EXP X

-> TAGDEF

A TAGDEF is constructed which defines the TAG *tag* to stand for the value delivered by *exp*, or a POINTER to the value delivered by *exp*, depending on whether *tag* was introduced by *make_id_tagdec* or *make_var_tagdec*.

exp will be a constant evaluable at load-time.

2.23 TOKLINK

A value of SORT TOKLINK expresses the connection between two TOKENs.
There is only one construct:

2.23.1 make_toklink

internal: TOKEN,
external: TOKEN

-> TOKLINK

A TOKLINK is constructed which defines the program fragment declared inside a UNIT as *internal* to be available to other UNITs in the same CAPSULE under the name *external*.

TOKLINKs are normally constructed by the TDF Builder in the course of resolving name clashes and sharings when constructing a composite CAPSULE. They are not likely to be needed by producer writers.

2.24 TAGLINK

A value of SORT TAGLINK expresses the connection between two TAGs. There is only one construct:

2.24.1 make_taglink

internal: TAG,
external: TAG

-> TAGLINK

A TAGLINK is constructed which defines the value declared inside a UNIT as *internal* to be available to other UNITs in the same CAPSULE under the name *external*.

TAGLINKs are normally constructed by the TDF Builder in the course of resolving name clashes and sharings when constructing a composite CAPSULE. They are not likely to be needed by producer writers.

2.25 TOKEN

A value of SORT TOKEN is an identifier which stands for a program fragment. It is a static non-negative number of unbounded size.

In discussion it is often qualified with the SORT of the program fragment for which it stands, as in:

TOKEN EXP INTEGER(0,255)

... describing a TOKEN which stands for an EXP INTEGER(0,255).

A TOKDEF defines the program fragment for which a TOKEN stands.

The construct *apply_token* substitutes that program fragment for the TOKEN.

2.25.1 apply_token

token: TOKEN,
arguments: $(\Pi_{i=1}{}^{n} S_i)$_OPTION { $n > 0$ } {S_i may be any SORT}

-> R

token will be the subject of a TOKDEC in the UNIT in which this construct occurs, declaring it to take *n* arguments of SORTs S_i and delivering a value of SORT *R*.

The program fragment for which *token* stands is substituted into the TDF program in place of the *apply_token* construct. It may be parameterised by *arguments*. If *arguments* are supplied, they will be of the SORTs specified in *token*'s TOKDEC.

It should be emphasised that the substitution specified by *apply_token* is an action performed on program at translate-time. It is **not** a run-time action. The use of TOKENs is described fully in §1.5.

3 Glossary

This glossary gives a quick explanation of some key TDF terms and references to where more detailed accounts can be found elsewhere in this document.

Architecture Neutrality
Program is architecture neutral if it will run consistently on a variety of target architectures. (See §1.7.)

BOTTOM
BOTTOM is the SHAPE of pieces of program which do not terminate normally. For instance, the SHAPE of *goto(...,..)* is BOTTOM. (See §2.1.1.1.)

CAPSULE
A CAPSULE is an independent piece of TDF program which defines a number of values and program fragments and makes some of them available for linking. (See §1.8.)

EXP
An EXP is a fragment of TDF program which will deliver a value when the program is run. It corresponds to expressions in high-level programming languages. (See §2.2.)

Installer
An Installer is a piece of software which manages the installation of a piece of TDF program on a user's machine. If the TDF program being installed needs to be merged with local, target-dependent software, the TDF Installer will use the TDF Builder to do this. It will then use the Translator to convert the TDF to machine code and then invoke the system linker to produce an executable image. (See §1.1.)

Lifetime
A pointer's lifetime is that zone of program over which an attempt to read its contents is meaningful. (See §2.2.4.18.)

NOF
NOFs (pronounced 'en-of') are arrays of fixed size. (See §2.1.2.8.)

NTEST
NTESTs are used in generic testing constructs to indicate which of a range of possible tests is to be applied - eg. equal, less_than etc. (See §2.9.)

OFFSET
OFFSET is the SHAPE of values which measure displacements in memory. The fact that TDF distinguishes OFFSETs from ordinary INTEGERs means that it can describe pointer arithmetic in a completely architecture neutral fashion. (See §1.7.2.)

Producer
A Producer is a piece of software which generates TDF program, typically by compiling from a high-level programming language. (See §1.1.)

SHAPE
SHAPEs are TDF's analogues of programming languages' types. Unlike types, however, they give only as much information about values as is necessary to describe their memory requirements in an architecture neutral fashion. (See §2.1.)

Sharing
Two pointers share if the spaces to which they point overlap. (See §2.2.4.1.1.)

SORT
SORTs are TDF's analogues of programming languages' syntactic classes, such as identifiers, types, labels etc. (See §1.3.2.)

TAG
A TDF TAG corresponds to a programming language identifier Unlike an identifier, however, it gives no mnemonic information. (See ¶Binding: Discussion¶.)

TAGEXTERNs and TOKEXTERNs
TAGEXTERNs and TOKEXTERNs set up associations between the names of TDF values and program fragments - TAGs and TOKENs - and strings. This assists in the merging of separate pieces of TDF and in system linking. (See §2.15 and §2.14.)

TDF Builder
The TDF Builder is a program written specially to support TDF. It merges separate CAPSULEs together. It is typically used to merge a target-independent CAPSULE supplied by a vendor, with a target-dependent CAPSULE on the user's machine before translation. (See §1.8.1.)

TOKENs and Tokenisation
Tokenisation is similar to macro substitution: TOKENs stand for pieces of TDF program. But the fact that TOKENs can substituted at any point in the progress from TDF on the developer's hardware to machine code on the user's hardware makes them a powerful mechanism for achieving architecture neutrality, code compression and optimisation. (See §1.5 and §1.8.)

TOP
TOP is the SHAPE of pieces of program which do not deliver any useful value. For instance, the SHAPE of *integer_test(..,..,..,..)* is TOP: it carries out a test which may result in a jump to another program location or terminate normally, but delivering no useful value. (See §2.1.1.2.)

Translator
A Translator is a piece of software which translates TDF program into a particular architecture's machine code. (See §1.1.)

TUPLE
TUPLEs are analogous to 'struct's in ANSI C. (See §2.1.2.4.)

UNIT
A UNIT is a collection of declarations and definitions of values and program fragments. A CAPSULE will contain one or more UNITs. (See §2.17.)

REPORT DOCUMENTATION PAGE DRIC Reference Number (If known) ....

Overall security classification of sheet ..................................UNCLASSIFIED ....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No. REPORT 91014	Month MAY	Year 1991
Originators Name and Location RSRE, St Andrews Road Malvern, Worcs WR14 3PS		
Monitoring Agency Name and Location		
Title TDF: SPECIFICATION OF SUBSET TO SUPPORT ANSIC, C++, FORTRAN 77, COBOL AND PASCAL		
Report Security Classification UNCLASSIFIED		Title Classification (U, R, C or S) U
Foreign Language Title (in the case of translations)		
Conference Details		
Agency Reference	Contract Number and Period	
Project Number	Other References	
Authors FOSTER, J M; BRANDRETH, M; CORE, P W; CURRIE, I F; PEELING, N E		Pagination and Ref 93

Abstract

TDF is an intermediate format for distributing software applications devoloped by the United Kingdom's Defence Research Agency, Electronics Division at RSRE, Malvern. Report No 91005 gave an account of the whole of TDF. The present report updates the account of the subset of TDF which supports ANSI C, C++, FORTRAN 77, COBOL and Pascal, described in 91005 as TDF Level 0.

The Introduction gives an overview of the TDF concept and sets the scene for the Definition. This specifies each of the constructs which make up the subset of TDF described here. A Glossary gives a quick explanation of some key TDF terms.

Abstract Classification (U,R,C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)

UNLIMITED

S80/48