Transputer Common Object File Format

Paul Sidnell, Martin Day, Andy Pepperdine, Andy Whitlow

SW-0011-7

APPROVED 1 March, 1991

Contents

1 Introduction

This document sets out the reasons for the creation of the transputer common object format, TCOFF. It falls into two main sections. First there is a general description of the motivations behind the design and features required. This is followed by a more detailed description of the format.

2 Objectives

The major objective in defining a standard object file format for transputer software is to increase the compatibility of software products from different companies. This will allow users to mix object modules and libraries built with different compilers and assemblers and for vendors to supply object modules and libraries compatible with a large range of compilers. This should lead to an even more rapidly increasing software base for the transputer range of processors.

In defining the object format we hope to enable all current manufacturers to switch to the new object format without any major changes in the way they compile their code. Obviously all compilers are going to have to be modified to some extent but we hope that everyone will be able to switch to the new format by simple changes.

2.1 Migration to the new standard

Whilst this description is to be freely available, it isn't going to be possible for all manufacturers to switch straight to the new standard simultaneously. There is going to be a changeover period during which customers are going to have object modules in both old and new formats which they are going to want to link together.

The chosen solution is to supply a conversion program to convert object files in the old format to the new one. This program would initially be used to convert all object and library files that the customer has produced and will also be needed after each compilation that produces object code in the old format.

2.2 The file format

It would be useful if other software tools were able to read and copy object files without any knowledge of the operand types and data record formats that make up the file. It is therefore proposed that an object file is made up of records composed of a tag and then a length count followed by that many bytes of data. This means that tools do not need to know the format of every record type in order to quickly read through a file only searching for certain kinds of record. The tag indicates the meaning and format of the remainder of the data in the block.

Parameters of various record types can be either numbers or strings. Numbers are be stored using a compaction technique. Strings are represented as a number giving the length of the string followed by that many bytes.

2.3 Design strategy

The object file format should enable a large number of linking strategies to be used. For example the relative positioning of items of static data may take place at compile, link or run time. It should be possible to convert all current object file formats to the new standard format.

Because of the need to support these different linking schemes the linker will need to be relatively complicated compared to most current implementations. This extra complication will, however, add a lot of flexibility and it should be possible to convert most current object file formats to the new standard using only a subset of the new linker commands.

2.4 Contents of the object file

The following is a description in fairly general terms of the proposed object file format giving some justification for the features included. A concise description of the linker command tags follows in the next section.

(1) Modules

An object file is composed of one or more modules. A module is a self-contained collection of code, data and linking information together with definitions of imported and exported symbols and symbols local to the module.

A collection of C functions and static data definitions in one file would typically produce, after compilation, an object file containing a single module. FORTRAN would produce an object file with one module per subroutine.

Code may be compiled for execution on only a particular set of processors running in certain modes. Current processors may be run in one of two modes, halt on error or ignore error. Code can be compiled to behave in various ways when an error occurs. Those currently supported by the occam compiler are to halt the entire processor or stop only the process that set the error. Code can also be compiled in a universal error mode which is compatible with both of the above.

Libraries will be created and maintained by a librarian utility prorgam. Libraries will consist of an object file with an index at the front. This will make selection of the appropriate object modules for inclusion more efficient. Libraries may also be built from linked units to allow for configuration programs to choose from a selection of loaders.

The linker will check that all code being linked is compatible with the target processor type and will all run in the same mode. When searching libraries only those modules compatible with the target processor will be considered for inclusion in the output file. Since more than one module in a library may be used for the same purpose, eg the same function but compiled for two compatible processor classes, linkers must have some algorithm for picking the 'best' module for the job. The simplest method would be for a librarian to order the library modules by increasing target generality, ie. more specific (and therefore efficient) first. A linker may then pick the first compatible module it finds and be sure that it is the best choice.

Future variants and new generations of transputers may support different instruction sets and execution modes. The compatibility checking scheme must be able to cope with this.

To reduce the number of object files in a system several object files may be combined into one file. This is done by simple concatenation of the files.

(2) Placing text

Compilers will direct the linker to place text (all data is referred to as text whether executable code, fixed tables, initial values for variables or uninitialised blocks) in one of a number of named sections. These sections could then be concatenated together into a single contiguous image or scattered throughout memory, possibly, in the case of executable code, being shared by several concurrent processes. The run time environment will dictate what approach is to be used. By including the appropriate relocation information in the linker output file the decision could be put off until run-time. It is the compilers responsibility to supply linker directives such that this is possible.

Sections can either be externally visible or local to a particular module (note that even when a section is local to one module labels can still be defined inside it that are externally visible). They will be tagged to indicate whether the text inside them is executable, readable, writable etc. or any combination thereof.

(3) Text records

Text records specify initial data values or code to be placed in a particular section. Each text record is appended to the end of the previous text record added to the section. The first text record is placed at offset zero inside the section.

Sections may be extended with text blocks initialised to all zeros. Whether these zeros need to appear in the binary image file produced by the linker will be implementation dependent.

Totally uninitialised blocks of text may be declared. These are useful where data does not need to be given any particular initial value.

The current load point in a section can be altered by a linker directive. One use of this is to support FORTRAN style common blocks (see Detailed description later).

(4) Symbols

Symbols can be either section names or labels. Each type of symbol may be local to the module in which it is defined or global and externally visible, shared between all the modules that are being linked together.

As symbols are declared they will be allocated symbol numbers. These numbers will then be used to refer to the symbol in the linker directives that follow.

A section symbol is a name used to refer to a section. Attribute bits in the symbol definition record indicate whether the symbol is a section symbol. When a module wishes to refer to a section it will use the symbol number assigned to the section symbol.

Label symbols are defined by expressions giving their value. This could be at a fixed offsets within a section, an address or a more complicated value. Label symbols may only be defined once. If the symbol is global only one module may define it but all may reference it.

It is also possible to declare local anonymous symbols which do not have a name, only a reference number. This saves compilers having to invent a unique name for each local symbol.

(5) Expressions

Expressions in linker directives will be stored in a prefix form. Operands can either be symbols, constants or other expressions.

Symbols are represented by a value indicating that a symbol operand follows. This is followed by the symbol number.

Constants are represented by a value indicating that a constant operand follows, followed by the value of the constant.

Special operands include the address of the current load point, the size of a specified section and the size in bytes of the machine word of the target processor.

When a label symbol is used as an operand the value used will be the position in the final text that corresponds to the location at which it was defined, as an offset in bytes from the zeroth byte of that text.

When a section symbol is used as an operand the value is taken to be a label, corresponding to the first byte of the section, as above.

Operators in expressions include add, subtract, multiply, divide, remainder, minimum and maximum. All arithmetic is carried out as unchecked signed 32 bit integer arithmetic.

Some directives may require expressions to be evaluated at the time they are first encountered by the linker. Other expressions may not need to be evaluated until all object modules and libraries have been read.

(6) Patching text

Linker text patching directives specify what adjustments the linker should make to the text.

There are two basic types of patch: patching a transputer instruction including a prefix sequence and patching values of various sizes.

Transputer instructions can be patched in one of two ways. Either the compiler must leave a slot in the code big enough for the linker to patch in the instruction sequence or the linker must be able to open up a hole in the code to the minimum size necessary.

The first method is easiest to implement but will produce bulkier and less efficient code than the second method.

The second method is more complicated and requires many more patches at link time since jumps around an instruction to be patched cannot be resolved at compile time. The linker will also have to adjust the positions of any labels that get moved as a result of changing the instruction size. A word alignment directive will also be needed to force certain pieces of code onto word boundaries (eg. any constant tables built into the code). An algorithm to perform this operation is given in the Transputer Instruction Set - A compiler writer's guide [INMOS'86] (section 4.4 Generating prefix sequences). This will need a small modification in order to support word alignment but otherwise remains unchanged.

When patching a value, it will be written into the text with the least significant byte first.

It may be useful to perform each patch type with either the value of the expression given, or the value divided by the machine word size in bytes, since most references to global static data will require this.

2.5 Run time initialisation

There are two methods of initialising writable global data.

The first method is to provide an initialisation routine in each module to set up the global data defined in that module.

It is proposed that a convention be adopted that a certain section name is reserved for use by the initialising part of the run-time system. Each module would contain linker directives to place into this section the addresses of any routines that need to be run to initialise data before the program proper is started. At startup this section would be scanned and all the routines specified would be called. It may be useful to specify a priority level for each routine so that they can be run in a certain order.

The second method is to include linker directives to initialise the text in a global data section. This is then included as an image in the binary file produced by the linker.

If a program is to be run in parallel with itself or is restarted in memory without the data area being reloaded then the second method presents a problem since the two instances of the program cannot share the same data. The controlling program or operating system will have to provide support in order to make copies of the initialised static data areas for each instance of the program.

3 Detailed description

The description of the Transputer Common Object File Format (TCOFF) given here is in a modified form of IDL (Interface Description Language [L83], [N86]). In particular, the use of IDL here should be read to imply that the order of the fields in each instance of an object is the lexical order found in the object type's description. For example, in the object of the class **OBJECT_FILE**, the field **of_linkable** immediately precedes the field **of_directives** with no padding between them.

An extension to IDL specifies the representation(s) an identifier may have. These values are given in definition statements, where the syntax of the value is:

```
constant (TYPE)
```

constant is any constant in the syntax of occam, and **TYPE** defines the representation size and form of **constant**. Eg. 10 (INT32) defines the constant value 10 in 2's complement form, 32 bits long. Occam was chosen because it defines the representation of all types in an unambiguous manner. The intent is that if real values were ever required, then the IEEE representations would be adopted.

Another extension is used to express the case that a generic object is instantiated with a given parameter. (See **header** for an example of a definition of such an object, and **sm_header** for a reference to one.)

Class and constant names are in upper case, and object names are in lower case. All fields in objects start conventionally with two characters which uniquely identify the object to which they belong.

The definition of the format of the TCOFF falls into two forms; the form that allows any tools to read the file and pick out only those directives it is interested in; and the form that describes what the linker will do with all the directives herein defined. In the last chapter, there is a list of all the IDL definitions in the alphabetic order of their left-hand sides.

3.1 High level syntax

First, the description of the form of the data. All tools can scan this form and select only the directives which are relevant to the tool.

```
object_file     =>    of_linkable    : linkable,
                      of_directives  : OPT SEQUENCE OF directive ;

linkable        =>    ln_header      : header (LINKABLE_TAG) ;
```

An **object_file** is an object with two components. It can be read as a sequence of directives, the first one being a **linkable** directive. **of_linkable** identifies this file as an object file. It is defined as a **directive** (qv.), whose representation is such that the linkers can distinguish this format from all previous formats.

```
linked_file     =>    lf_linked      : linked,
                      lf_directives  : OPT SEQUENCE OF directive ;

linked          =>    lk_header      : header (LINKED_UNIT_TAG) ;
```

A **linked_file** is an object identical in structure to an object file except that it begins with a **linked** directive. These files are produced by the linker and must therefore be distinguishable from linkable files.

```
directive       =>    dr_header      : header,
                      dr_command     : OPT SEQUENCE OF BYTE ;
```

dr_header enables the commands to be scanned without having to look inside them.

```
header (TAG)    =>     hd_tag        : DIRECTIVE_TAG = TAG ;
                       hd_length     : length ;

length          =>     number ;
```

This generic object is described in an extension to IDL. It specifies that a **header** object consists of two fields which may depend on the parameter (**TAG**). In fact, **hd_tag** is defined as of class **DIRECTIVE_TAG** but must have the value of the parameter **TAG**. These values are defined at the end of this description.

hd_tag precedes the length field in this object because we want to define a tag value which must be different from all possible initial records of the older linker files. This tag can then be used to identify this file as being of this new format.

hd_length is the number of bytes in **dr_command**. It is non-negative (ie. zero is a valid length).

```
DIRECTIVE_TAG   ::=   LINKABLE_TAG
                    | LINKED_UNIT_TAG
                    | START_MODULE_TAG
                    | END_MODULE_TAG
                    | SET_LOAD_POINT_TAG
                    | ADJUST_POINT_TAG
                    | LOAD_TEXT_TAG
                    | LOAD_PREFIX_TAG
                    | LOAD_EXPR_TAG
                    | LOAD_ZEROS_TAG
                    | ALIGN_TAG
```

```
| SECTION_TAG
| SYMBOL_TAG
| DEFINE_MAIN_TAG
| SPECIFIC_SYMBOL_TAG
| LOCAL_SYMBOLS_TAG
| DEFINE_LABEL_TAG
| DEFINE_SYMBOL_TAG
| DESCRIPTOR_TAG
| KILL_ID_TAG
| BYTE_PATCH_TAG
| WORD_PATCH_TAG
| REP_START_TAG
| REP_END_TAG
| COMMENT_TAG
| MESSAGE_TAG
| VERSION_TAG
| LIB_INDEX_START_TAG
| LIB_INDEX_END_TAG
| INDEX_ENTRY_TAG
;
```

These tags define the directive which the linker will recognise. They will always be found immediately preceding a length field in a directive.

3.2 Interpretation by a Linker

In order that the definition can be read easily, there is a convention that wherever a **header** is defined it is found immediately within the object that corresponds to the full **directive**. Thus **dr_command** is equivalent to the remainder of the object containing the **header** and is of length **hd_length**.

```
OBJECT_FILE      =>    module_list
                     | library
                     ;
```

An object of class **OBJECT_FILE** is either a sequence of modules, or is a library. It is assumed that the librarian is a separate tool that combines a number of objects of type **object_file** into one of type **library**, but which can still be described by essentially the same description.

```
LINKED_FILE      =>    lk_module_list
                     | lk_library
                     ;
```

An object of class **LINKED_FILE** is identical in format to a **unit** except that a **linked** directive is used in place of **linkable** directive.

```
module_list      =>    ml_body        : OPT SEQUENCE OF unit ;

lk_module_list   =>    lm_body        : OPT SEQUENCE OF lk_unit ;

unit             =>    un_linkable    : linkable,
                       un_body        : module ;

lk_unit          =>    lu_linkable    : linked,
                       lu_body        : module ;

module           =>    md_start       : start_module,
                       md_body        : OPT SEQUENCE OF LINK_COMMAND,
                       md_end         : end_module ;

start_module     =>    sm_header      : header (START_MODULE_TAG),
                       sm_cpus        : SET OF TRANS_FUNCTION,
                       sm_attrib      : SET OF ATTRIBUTES,
                       sm_language    : LANGUAGE,
                       sm_name        : string ;
```

sm_cpus defines functionality that the transputer must have for the code to execute correctly. The linker can check the compatibility of all modules being linked together.

sm_attrib defines which attributes the module has. These attributes include such things as word size, error response, and communication method (direct instruction or via library calls).

sm_language defines the source language used to create the module.

```
LANGUAGE         ::=   LANG_NOT_KNOWN
                     | LANG_LINKED
                     | LANG_OCCAM
                     | LANG_OCCAM_HARNESS
                     | LANG_ANSI_C
                     | LANG_FORTRAN_77
```

```
                | LANG_ISO_PASCAL
                | LANG_MODULA_2
                | LANG_ADA
                | LANG_ASSEMBLER
                ;
```

sm_name is the name of the module. Some languages such as Modula 2 require named modules. Other languages may leave this field unused or simply use the source file name.

Modules may be nested in this format, although it is not anticipated that any of the existing compilers can make use of this fact. (See also **symbol**).

It is expected that compilers will generate a single **unit** for each compilation of a source file. A unit starts with an indication of the format it is in and the generated code follows. Several **unit**s can be combined into a single **LINKED_FILE** by simple concatenation, hence the format indicators can occur between units in this case.

```
end_module      =>    em_header     : header (END_MODULE_TAG) ;
```

This directive merely shows where the most recent module terminates. If it is nested within another module, then the state is unstacked.

3.2.1 Directives

```
LINK_COMMAND      ::=    SIMPLE_DIRECTIVE
                       | replicator
                       | module
                       ;

SIMPLE_DIRECTIVE ::=     set_load_point
                       | adjust_point
                       | load_text
                       | load_prefix
                       | load_value
                       | load_zeros
                       | align
                       | section
                       | symbol
                       | define_main
                       | specific_symbol
                       | local_symbols
                       | define_label
                       | define_symbol
                       | descriptor
                       | kill_id
                       | patch
                       | comment
                       | message
                       | version
                       ;
```

These are the directives that the linkers are expected to recognise. Other tools need recognise only a subset.

The linker operates with the concept of a **current load point**. This is the position at which all text is placed. The loading of text increments current load point by the size of the text.

The loading is done into **section**s. These can be named. Linkers and compilers can choose their names appropriately for the areas of text to be located in proximity to one another. Linkers can state preferred names if it is required to combine output from different compilers in a sensible way.

The load point is initially undefined and each module which is loaded must direct the linker to place the load point in an existing section. Thereafter, text is loaded at the current load point unless it is explicitly reset.

```
set_load_point  =>    sl_header     : header (SET_LOAD_POINT_TAG),
                      sl_location   : ident ;
```

Sets the load point to current end of section **sl_location**. The **ident** in **sl_location** must have been defined by a **section** directive specifying it as a section. If the section is empty then the end of the section is in fact the start of the section.

```
adjust_point    =>    aj_header     : header (ADJUST_POINT_TAG),
                      aj_offset     : VALUE ;
```

This directive increments the current load point by the value in **aj_offset**. It remains in the same section.

The value in **aj_offset** must be determined at the time the **adjust_point** directive is found. There can be no unresolved symbols in the expression.

VALUE is defined in the next subsection.

This directive can be used in defining FORTRAN COMMON blocks and their initial values. When a BLOCKDATA statement is used to initialise a COMMON block (named 'common', say), then the sequence:

```
set_load_point      common
adjust_point        MINUS_OP common load_point
load_text           text
```

will do the job. The **set_load_point** directive sets the load-point to the *end* of the section named 'common'. The **adjust_point** then adds to the current position the value of ("location of 'common'" minus "location of current load-point"); in other words, it sets the load point at the start of the section 'common'. Text is now overlayed at this point.

If there is no initial value, then the sequence:

```
set_load_point      common
adjust_point        PLUS_OP MINUS_OP common load_point size_of_common
```

can be used. This sets the load point to be 'size_of_common' bytes from the start of the section. If the section was smaller, it will have been stretched. If the section was larger, it remains the same size. Note that in the second example the contents of this section is undefined, since no initialised text has been loaded.

```
load_text     =>   lt_header     : header (LOAD_TEXT_TAG),
                   lt_text       : string ;
```

Loads **lt_text** into the next bytes from the current load point. The load point is updated to the byte beyond the loaded text.

```
load_prefix   =>   lx_header     : header (LOAD_PREFIX_TAG),
                   lx_size       : number,
                   lx_value      : VALUE,
                   lx_instr      : instruction ;

instruction   =>   in_op_code    : number ;
```

Places **lx_instr** into the text and fixes up the prefixes to occupy the minimum space. The value of the prefix will be taken from **lx_value**.

lx_size is the number of bytes the instruction must take. The linker will leave that number and fill in the prefix with **lx_value**. As a special case, a size of zero indicates that the linker will use the fewest number of prefix operations that will accommodate the value. Otherwise, the instruction will use the fewest prefixes at the least significant end of the space, with the remainder padded with *pfx 0* operations.

lx_instr contains the instruction to be included in the final byte of the patch. This is the transputer opcode in the range [0..15].

```
load_value    =>   lv_header     : header (LOAD_EXPR_TAG),
                   lv_size       : number,
```

```
                   lv_value      : VALUE ;
```

Places **lv_value** into the text at this point using **lv_size** bytes. The value will be stored with the least significant bit at the lowest address (ie. "little-endian" like the transputers). If the size is given as zero, then the machine wordsize is used.

```
load_zeros     =>  lz_header     : header (LOAD_ZEROS_TAG),
                   lz_count      : number ;
```

Places **lz_count** bytes containing binary zeros at the current load point and updates the load point to the next byte beyond them.

```
align          =>  al_header     : header (ALIGN_TAG),
                   al_modulo     : number ;
```

Causes the load point to be aligned on the next higher offset from the start of the current section (mod **al_modulo**). It is assumed that the linkers and configurers will be sensible and that each section will start at an address with the most stringent alignment requirement for the target machine. If **al_modulo** is zero, then the alignment is taken to be to the wordsize of the target machine (ie, 2 or 4 bytes). The gap will be filled with *pfx 0* operations.

```
symbol         =>  sy_header     : header (SYMBOL_TAG),
                   sy_usage      : SET OF USAGE,
                   sy_symbol     : string ;
```

Assigns to **sy_symbol** the ident next in sequence. The first **symbol** in a module defines the symbol to have the ident 0, and subsequent ones have idents incremented by one for each in sequence. The idents are valid only within a module, the numbering starting afresh with each new module encountered.

If a module is nested inside another one, then the count of identifiers is not re-initialised to zero when the module starts. Instead, the count continues upward until the end of the module is found, when the count is reset to what it was on entry to the module. This enables references to be made from the inner one to symbols defined in the outer one without ambiguity, and without having to make them globally known.

The usage of a symbol is given by **sy_usage**. At the end of a module, all the local symbols become out of scope and can no longer be referenced.

sy_symbol is the name given to the symbol for humans to read. For local symbols that the compiler generates for its own use, **sy_symbol** can be a null string.

Note: The sizes of workspace and vectorspace are required by some systems. These values are attributes of entry points to routines. One method of implementing it in this format is as follows:

Define a convention for naming the attributes (eg. add 'WS to the name to get a symbol meaning the worspace size for this entry point). Use the linker's expression handling to give this symbol a value.

```
USAGE          ::=   LOCAL_USAGE
                   | EXPORT_USAGE
                   | IMPORT_USAGE
                   | WEAK_USAGE
                   | CONDITIONAL_USAGE
                   | UNINDEXED_USAGE
                   | PROVISIONAL_USAGE
```

```
                   | ORIGIN_USAGE
                   ;
```

LOCAL_USAGE means that the identifier is known only within this module - the linker can expect it to be defined before the end of the module, and it is not kept beyond the end.

EXPORT_USAGE means that the definition of the symbolic name is passed to all other modules. Only one module can export a given name.

IMPORT_USAGE means that the references to this name are defined elsewhere.

A symbol may have only one of the **LOCAL_USAGE**, **EXPORT_USAGE** or **IMPORT_USAGE** attributes.

WEAK_USAGE means that the label does not have to be resolved during linking. If it is not, then the value is zero (absolute). This can be used to set up chains of control blocks at link time, as well as only including library modules if the compiler dictates that they are actually necessary. Such a symbol may only have the **IMPORT_USAGE** attribute.

CONDITIONAL_USAGE means that the definition of the name should only be done if it has not already been defined, ie. if this symbol is defined many times, only the first definition is used. This enables a compiler to provide a reference to a label to be used as the head of a chain without knowing which entry in the chain will be the head.

UNINDEXED_USAGE means that the librarian should not include this symbol in the index at the head of the library. It is meaningful only if the symbol is also defined in the module and is marked as **EXPORT_USAGE**.

PROVISIONAL_USAGE means that the definition of the name is *fixed* when that name is no longer in scope, when it has been killed or at the end of linkage, ie. if this symbol is defined many times, only the last definition is used. Any new definition overrides an existing one. Symbols of this type are used in situations where the value may have to be changed as linkage progresses. For example, many expressions may depend upon the size of some module table, but they must not be evaluated until the final size of this table is known, ie. at the end of linkage.

A symbol may have only one of the **CONDITIONAL_USAGE** or **PROVISIONAL_USAGE** attributes.

ORIGIN_USAGE specifies that a symbol may be used by **specific_symbol** directives (see later). Such a symbol may only be either **IMPORT** or **EXPORT**. It is an error to attempt to define such a symbol or use it in an expression. The compiler must ensure that this symbol must not corrupt the users name space, for example by being a valid function name. Only one such symbol may be exported by a module.

```
section        =>  se_header     : header (SECTION_TAG),
                   se_section    : SET OF SECTION_TYPE,
                   se_usage      : SET OF USAGE,
                   se_symbol     : string ;


SECTION_TYPE   ::=   WRITE_SECTION
                   | READ_SECTION
                   | EXECUTE_SECTION
                   | DEBUG_SECTION
                   | VIRTUAL_SECTION
                   ;
```

Symbols may also be used to represent sections by using the **section** directive. This directive is similar to the **symbol** directive, but has an extra set of protection attributes in **se_section**. It should be noted that the effect of **section** is identical to **symbol** if no protection attributes are given, ie. a normal symbol is produced.

Since section names are generated by compilers, it is the responsibility of the compiler that they do not corrupt the users name space.

The symbol formed by creating a section will take the value of the offset from the start of all output text to the location in the text where the section was placed.

Sections may have only the **LOCAL_USAGE** or **EXPORT_USAGE** attributes. Sections are assumed to have the **UNINDEXED_USAGE** attribute by the librarian.

When a symbol defines a section, then the section can be used in various ways, specified by the following section types.

WRITE_SECTION means that the section contains data, and can be written to. This would be used for static variables and FORTRAN COMMON blocks.

READ_SECTION means that the data in it may only be read. This would be used for constant pools and initialisation values.

EXECUTE_SECTION implies that the section contains code.

Any combination of **WRITE_SECTION**, **READ_SECTION** and **EXECUTE_SECTION** may be used to represent the intended limitations of the section contents . If hardware support for such protection mechanisms exist, then these mechanisms will be used to enforce the protection regime.

VIRTUAL_SECTION causes the linker to build a section that contains no actual text. Valid operations within such a section include adjusting the load point and defining labels. No patches or text my be placed in such a section. Such a section is used as a convienient method of calculating offsets and sizes of some run time resource. This section will not occupy any space in the output. All labels defined within such a section will take the value of their position within (offset from the start of) the section.

Since the section will never be output, the location of the section in the output is always set to zero. The size of the section will be the maximum load point position used.

DEBUG_SECTION causes the linker to build a section for the loader to keep as debug information. This section will be kept separate from other text sections. Labels will be calulated in an identical manner to **VIRTUAL_SECTION**s.

A section may not be both virtual and debug. Also, the read, write and execute attributes do not apply to debug or virtual sections.

```
define_main     =>    dm_header      : header (DEFINE_MAIN_TAG),
                      dm_entry       : ident ;
```

This directive defines **dm_entry** as the main entry point to the program. It will normally be in the run-time system for the language, which will then call the user's program.

```
specific_symbol =>    sp_header      : header (SPECIFIC_SYMBOL_TAG),
                      sp_usage       : SET OF USAGE,
                      sp_symbol      : string,
                      sp_origin      : ident ;
```

This directive defines a symbol in a similar manner to **symbol** except that it associates the symbol with an origin symbol. The linker will treat the the name of the **sp_origin** symbol as an extension of **sp_symbol**.

This mechanism allows names to have scope by distinguishing symbols of the same name with differing origin symbols.

Similarly, the name of the origin symbol may be a function of some aspect of the module from which it was exported, providing security for languages such as occam with strict module dependancies.

Note that **sp_origin** must be of type **ORIGIN_USAGE**.

Typically, when exporting such a symbol, a single origin symbol name is first formulated such that it does not corrupt the users name space and is then exported with the **symbol** directive. All symbols subsequently exported from the module use the id of this origin symbol in the **sp_origin** field of the **specific_symbol** directive.

When importing such a symbol, the module of origin is scanned until the origin symbol is encountered and a symbol of the same name imported into the module under construction. This id is then placed in the sp_origin field of all symbols subsequently imported that must originate from the module being scanned.

The linker must consider two identical symbols with differing origins as being unique.

If a non specific symbol is imported with the **symbol** directive then it may be resolved by a symbol exported by the **specific_symbol** directive. This allows for mixed language programming. There is, however, no way of determining which specific version is used if there is more than one.

The linker must consider it an error if it detects a specific and non specific symbol of the same name to be exported in a single link.

```
local_symbols   =>    lo_header     : header (LOCAL_SYMBOLS_TAG),
                      lo_count      : number ;
```

This is a quicker and more compact way of defining local labels. It is equivalent to **lo_count** copies of a definition of an anonymous local label by means of the **symbol_id** directive.

```
define_label    =>    dl_header     : header (DEFINE_LABEL_TAG),
                      dl_ident      : ident ;
```

Assigns the address where the current load point is as the value of the symbol given by **dl_ident**. If this ident is used within an expression, then the intent is that it is the actual address which is used (and not eg. the offset within a named section). Linkers are expected to place restrictions on the contexts where such labels can be used (see **expression**). It is an error to attempt to define a label which is already defined (unless it is a **CONDITIONAL_USAGE** or **PROVISIONAL_USAGE** symbol).

```
define_symbol   =>    ds_header     : header (DEFINE_SYMBOL_TAG),
                      ds_ident      : ident,
                      ds_value      : VALUE ;
```

Assigns the value in **ds_value** as the value of the symbol defined in **ds_ident**. Note that all the attributes of the expression in **ds_value** should be passed on as attributes of the symbol. Eg. if A is a label in section X, and c is a numerical constant, then it would be expected that (+ A c) would represent another location in section X at offset c from A.

```
descriptor      =>   de_header     : header (DESCRIPTOR_TAG),
                     de_symbol     : ident,
                     de_language   : LANGUAGE,
                     de_string     : string ;
```

The descriptor is primarily to allow occam compilers to find the parameter profile, channel usage and workspace requirements for an entry point, defined by **de_symbol**. The contents of the descriptor in **de_string** can be in any format suitable for the appropriate compiler. Note that a C compiler could put a prototype here for documentation purposes.

```
kill_id         =>   ki_header     : header (KILL_ID_TAG),
                     ki_ident      : ident ;
```

The symbol defined by **ki_ident** is removed from the symbol name table. This will allow a redefinition by another module for the same name. All existing resolutions to the name are not changed; ie. if a reference to the ident has already been seen, it continues to refer to the old definition.

```
patch           =>   pt_header     : header (PATCH_TAG),
                     pt_location   : VALUE,
                     pt_size       : number,
                     pt_value      : VALUE ;

PATCH_TAG       ::=  BYTE_PATCH_TAG
                   | WORD_PATCH_TAG
```

The **BYTE_PATCH_TAG** means that **pt_value** in the **patch** directive will be added into the patch location. **WORD_PATCH_TAG** will divide the value by the wordsize of the target machine before adding it into the patch. It is expected that there will be a lot of such patches, and hence this will save 2 bytes per patch (**DIVIDE** and **word_length**).

The final value of **pt_value** is added to the contents of what has been loaded into the text at **pt_location**. The location can be a **VALUE** in order to allow patching of fixed size tables without generating large numbers of local **ident**s (eg. the debug information from compilers is typically in this form). In this case, the patch location can be represented as "debug-section-name plus offset".

The value will be added in a "little-endian" way (ie. least significant bit at the lowest address). The size of the patch is given by **pt_size** in bytes, where a length of 0 means that the length is that of a word in the target processor. The intent of this directive is to allow language systems to extract information from various compilation units so that the run-time system can allocate the appropriate resources.

It is expected that linkers will restrict some combinations of values of the fields in this directive (eg. sizes must be less than 16, or must be a power of 2, etc.).

```
replicator      =>   rp_start      : rep_start,
                     rp_body       : OPT SEQUENCE OF REPL_DIR,
                     rp_end        : rep_end ;

rep_start       =>   rs_header     : header (REP_START_TAG),
                     rs_count      : number ;

rep_end         =>   re_header     : header (REP_END_TAG) ;

REPL_DIR        ::=  set_load_point
                   | adjust_point
                   | section
```

```
        | symbol
        | local_symbols
        | patch
        | load_text
        | load_prefix
        | load_value
        | load_zeros
        | align
        | define_label
        | define_symbol
        | kill_id
        | replicator
        | comment
        | message
        ;
```

The directives in **rp_body** will be executed **rs_count** times. Replicators can be nested to any implementation limit. This is intended to reduce the amount of text required to implement initialisation of large data area with replication of values (eg. FORTRAN DATA statements with implied DO loops).

It is expected that the directives in the body of the replicator will normally only be **load_text** directives and that linkers may restrict what they will allow there. **REPL_DIR** defines the directives it seems reasonable to expect can be implemented everywhere. Note that the definition directives are only sensible if they are conditional, provisional, or if there is also a **kill_id** in the replicator too.

```
comment       =>   cm_header     : header (COMMENT_TAG),
                   cm_copy       : BOOLEAN,
                   cm_print      : BOOLEAN,
                   cm_text       : string ;
```

This is designed for use with other tool's where the linker is required to ignore **cm_text**. Eg. the compilers could supply filenames and version numbers in comments in the linker file.

If **cm_copy** is TRUE, then the comment directive is copied into the linker's output in an implementation defined manner. If it is FALSE, then it is discarded. This provides a mechanism for compilers to pass information through to other tools. For instance, the date and time of compilation and the language the module was written in could be stored in the linker output for each component module.

If **cm_print** is FALSE then a lister tool knows that the **cm_string** is not ascii.

```
message       =>   ms_header     : header (MESSAGE_TAG),
                   ms_level      : ERROR_LEVEL,
                   ms_text       : string ;

ERROR_LEVEL   ::=  NORMAL_MSG
                 | WARNING_MSG
                 | ERROR_MSG
                 ;
```

The text in **ms_text** is printed out on some implementation defined device. This is designed to provide the language implementations with a way of informing the user of error cases and levels of library modules included. The linker may or may not copy the directive to its output in some form.

In addition, if **ms_level** has the value **ERROR_MSG**, then the linking is terminated immediately with a suitable operating system dependent return code. If **ms_level** is **WARNING_MSG**, then when the linking is finished, the linker will return to the operating system with a warning level return code (if applicable). If the value is **NORMAL_MSG**, then the message is simply printed. (The division is explicitly given in the directive because different host operating systems use different conventions for what is a normal return and what is not. The linker should know its host and convert the code appropriately.)

```
version         =>    vn_header      : header (VERSION_TAG),
                      vn_tool_id     : string,
                      vn_origin      : string ;
```

This record is used for information concerning the history of the module. **vn_tool_id** holds the name of the tool used to create the module and **vn_origin** holds the name of that tools primary input file.

```
library         =>    lb_linkable    : linkable,
                      lb_lookup      : lib_index,
                      lb_version     : OPT version,
                      lb_body        : OPT SEQUENCE OF module ;

lk_library      =>    ll_linkable    : linked,
                      ll_lookup      : lib_index,
                      ll_version     : OPT version,
                      ll_body        : OPT SEQUENCE OF module ;
```

A **library** has an index at the front to cut down the search times when the linker is searching for unresolved references. The presence of **lib_index** determines that the file contains a **library**. A tool is available for combining several **module_list**s into a **library**. This tool may make use of **lb_version** to store information concerning the origin of the modules.

An **lk_library** is a library consisting exclusively of linked units. This is envisaged to be a variety of bootstrap loaders for use by a configurer.

```
lib_index       =>    li_start       : lib_index_start,
                      li_entries     : OPT SEQUENCE OF index_entry,
                      li_end         : lib_index_end ;

lib_index_start =>    ls_header      : header (LIB_INDEX_START_TAG),

lib_index_end   =>    le_header      : header (LIB_INDEX_END_TAG) ;
```

The **lib_index** is searched by the linker first. Only those entries that refer to symbols that are unresolved at the time of search are included in the final output. Hence, if a **module** defines more than one symbol, then there should be one entry for each label, each referring to the same **module** in the **library**.

```
index_entry     =>    ie_header      : header (INDEX_ENTRY_TAG),
                      ie_position    : INT32 (unsigned),
                      ie_cpus        : SET OF TRANS_FUNCTION,
                      ie_attrib      : SET OF ATTRIBUTES,
                      ie_language    : LANGUAGE,
                      ie_descriptor  : string,
                      ie_symbol      : string ;
```

The value held by **ie_position** defines the position within the library of the module which will resolve the identifier **ie_symbol**. Its value is the number of bytes from the start of the file at which the first byte of the module (ie. the **START_MODULE_TAG** byte is to be found. It is represented in a 32-bit integer form in order to make the sizes of the index entries determinable as they are created. Then only the values of **ie_position** need be filled in.

In common with the other forms integer in the transputer, the value is stored in the file in a ”little-endian” manner; ie. the first byte in the file represents the least significant part.

Althought this restricts the size of a library to 4Gb, this is not expected to cause a problem in the lifetime of the product.

ie_cpus and **ie_attrib** can be used either to check the compatibility or to select from alternatives where a **library** includes multiple definitions for the same label.

ie_language is copied from the symbols module of origin and **ie_descriptor** is copied from any descriptor associated with the symbol, or is zero length.

3.2.2 Values and Expressions

Every symbol has a **value**. If the symbol is a label, then the value is the address of the location where it will be found in the final text, as an offset in bytes from the first byte of text. This means that, in general, labels are only used as a means of locating a point in the text relative to some base point. The resulting file should be relocatable without further patching.

The symbol could be given a value by means of a **define_symbol** directive. In this case, a general expression can be given from which a value can be computed. This value could be an address, with the same restrictions as if it had been defined like a label; or it could be independent of where the program is loaded ("position independent").

Note that the difference between two labels defined in the same section is always position independent, although if the labels are in the code section and there are instructions to be patched between them, then the actual value is not known until the patching has been done. If the labels are in different sections, then it depends on the loading strategy of the operating system whether the linker can treat it as position independent or not.

It is expected that linkers might not implement the full generality for expressions in this format. For example, a linker might prepare a program where each code section can be placed independently of all others ("scatter loading"). It might then insist that labels can only be used in expressions which take the difference between two labels in the same section, or which add or subtract constants to/from a label.

```
VALUE          ::=   constant
                   | load_point
                   | symbol_value
                   | section_size
                   | expression
                   | word_length
                   | adjust_prefix
                   ;
```

A **value** might be known at the time it is encountered, or it might contain relocatable references which will only be established at the end of the loading process.

```
constant        =>    co_value_tag  : CO_VALUE_TAG,
                      co_value      : number ;
```

Defines a constant value, which is assumed to be an integer. Linkers may restrict the sizes of the acceptable values.

```
load_point      =>    lp_value_tag  : LP_VALUE_TAG ;
```

Has the value of the current load point. This need not be computable at the time it is encountered. It is invalid if the load point has not been initialised.

```
symbol_value    =>    sv_value_tag  : SV_VALUE_TAG,
                      sv_identifier : ident ;
```

Has the value of the symbol. If the symbol is a label, then the value is the address at which the linker has placed or will place it. If the symbol usage is **WEAK** and it is not resolved, then the value is zero.

Only in the case where the linker is preparing the module for execution at a predetermined location, can it assume that the label has a known address; in other cases, it is interpreted as a base-point plus offset. The offset is always computable at link time, but the base-point will be known only symbolically. For linkers that are preparing modules for execution by an operating system, then the base-point may be the start of the module on loading. If the operating system can support scatter-loading (ie. sections are to be loaded wherever the OS decides), then the base-point may be the start of the containing section.

```
ident          =>   id_index      : number ;
```

The index number of the symbol (see **section**).

```
section_size   =>   ss_value_tag  : SS_VALUE_TAG,
                    ss_identifier : ident ;
```

Has the value of the number of bytes in the section at the end of linking. This enables run-time systems to get the sizes of tables (eg. the module table for module linkage).

```
adjust_prefix  =>  ap_value_tag   : AP_VALUE_TAG,
                   ap_operand     : VALUE ;
```

Has the value of **ap_operand** minus the length of **ap_operand** when it is prefix encoded. This is used in prefix patches where the position of the patch is known but the actual patched value must be relative to the end of the patch.

```
expression     =>   ex_oper_tag   : OPER_VALUE_TAG,
                    ex_operand1   : VALUE,
                    ex_operand2   : VALUE ;
```

An expression is given in prefix form. **ex_operand1** is what is normally written on the left side of the operator, and **ex_operand2** is the right-hand side.

It is expected that linkers will restrict the type of expressions that they will allow. For example, they could restrict labels to operands in expressions of the form (- A B), where A and B are labels in the same named section, thus making the expression relocatable.

```
OPER_VALUE_TAG  ::=   PLUS_OP
                    | MINUS_OP
                    | TIMES_OP
                    | DIVIDE_OP
                    | REM_OP
                    | MAX_OP
                    | MIN_OP
                    ;
```

All operations are carried out as unchecked signed 32 bit integer arithmetic.

These operators do the obvious for the names they have.

```
word_length    =>   WL_VALUE_TAG ;
```

word_length is the number of bytes in the target machine's word. This may be required for linking modules that may execute on both 16 and 32 bit machines.

3.3 Encoding Methods

The general encoding methods used within the TCOFF format are described below.

3.3.1 Numbers

```
number          =>    nm_sign          : OPT SIGN_INDICATOR,
                      nm_pos_number    : CODED_NUMBER ;

CODED_NUMBER    ::=   simple_number
                    | prefix_1_number
                    | prefix_2_number
                    | prefix_4_number
                    | prefix_8_number
                    ;

simple_number   =>    sn_number        : <0 .. 250>(BYTE) ;

prefix_1_number =>    p1_prefix        : PFX_1_TAG,
                      p1_number        : BYTE ;

prefix_2_number =>    p2_prefix        : PFX_2_TAG,
                      p2_number        : INT16 ;

prefix_4_number =>    p4_prefix        : PFX_4_TAG,
                      p4_number        : INT32 ;

prefix_8_number =>    p8_prefix        : PFX_8_TAG,
                      p8_number        : INT64 ;
```

The intention is that the format of **number** will allow flexible extension in the future and not seriously inconvenience us now.

The decoding algorithm for numbers can be described as follows:

> Look at the first byte; let the code be n (unsigned, 0 $<=$ n $<=$ 255).
> If n $<=$ 250, then the value is n.
> If 251 $<=$ n $<=$ 254, then the value is (unsigned) in next 2^{n-251} bytes.
> If n $=$ 255, then the ones-complement of the value follows in coded form.

If the number is negative, then it is represented as the sign bit followed by its ones-complement in coded form. If the number is greater than 250, then it placed in the smallest field of 1, 2, 4 or 8 bytes and the appropriate prefix is prepended (251, 252, 253 or 254 respectively).

This algorithm looked most attractive when we examined some real data from various compilers. There are very few negative numbers and a large number of small positive ones.

Note that the combination of 255 (meaning negative) can not be followed by another 255; so we have at least one reserved combination for the future.

INT32 is used in TCOFF in library indices where a fixed size field is required. An **INTnn** number has **nn** bits. The bytes are ordered in increasing significance, i.e. smallest first (little endian).

64 bit numbers of any format can not presently be supported.

3.3.2 Strings

Strings are used in TCOFF for storing sequences of bytes

```
string          =>    st_length      : number,
                      st_chars       : OPT SEQUENCE OF BYTE ;
```

st_length is the length of **st_chars**. It does not include itself and is non-negative.

3.3.3 Sets

Sets are encoded numbers which represent groups of attributes. The position of a set bit signifies the fact that a particular attribute is present. Each individual item in a particular set is represented by a power of two. A set is constructed by or'ing together the elements required in a set. It should be noted that where a bit is not used by any member of a set, it is reserved as being clear. This allows for compatible future extension. A set may also be accompanied by a "reserved" number which represents a group of unused bits which must at all times be set to '1'. This enables some as yet unspecified attribute to be removed , should the need arise with some future transputer.

3.4 Specifying Transputer Attributes

Transputer attributes are define using two words, the **TRANS_FUNCTION** word and the **ATTRIBUTES** word. In order to provide maximum expandability the meaning of these words is defined according to the architecture of the transputer in question.

Currently 2 architectures are defined:

1 **T Architecture** : T212, T222, T225, M212, T400, T414, T425, T800, T801, T805, TA, TB. (T426, T806).

2 **H Architecture** : H1

The architecture of a transputer is encoded in TCOFF by a set of 5 bits in the **TRANS_FUNCTION** word.

```
ARCHITECTURE     ::=   ARCH_T
                     | ARCH_H
                     | ARCH_RESERVED_1
                     | ARCH_RESERVED_2
                     | ARCH_RESERVED_3
                     ;
```

The reserved architecture bits are reserved unset. Note that the architecture bits are mutually exclusive.

The definition of the rest of the bits in the **TRANS_FUNCTION** and **ATTRIBUTES** word depends on which of the architecture bits is set. The IDL definition of the **TRANS_FUNCTION** is thus given by:

```
trans_function  =>    tf_arch          : ARCHITECTURE;
                      tf_functionality : SET OF TRANS_FUNC;
```

The type **TRANS_FUNC** is currently defined as follows:

```
TRANS_FUNC ::=   INSTR_CORE             -- (T)
               | INSTR_FMUL             -- (T)
               | INSTR_FP_SUPPORT       -- (T)
               | INSTR_DUP              -- (T)
               | INSTR_WSUBDB           -- (T)
               | INSTR_MOVE2D           -- (T)
               | INSTR_CRC              -- (T)
               | INSTR_BITOPS           -- (T)
               | INSTR_FPU_CORE         -- (T)
               | INSTR_FPTESTERR        -- (T)
               | INSTR_LDDEVID          -- (T)
               | INSTR_DEBUG_SUPPORT    -- (T)
               | INSTR_TIMER_DISABLE    -- (T)
               | INSTR_LDMEMSTARTVAL    -- (T)
               | INSTR_POP              -- (T)
               | INSTR_RESERVED_SET     -- (T)
               | H_INSTR_RESERVED_SET   -- (H)
               ;
```

Those values marked with a **(T)** are for use with the T architecture only, similarly for **(H)**.

The **ATTRIBUTES** word is easier to define as it is simply a set of bits. Possible values are given by the following IDL definition:

```
ATTRIBUTES ::=   ATTRIB_WORD_16          -- (T) 16 bit words
               | ATTRIB_WORD_32          -- (T) 32 bit words
               | ATTRIB_MEMSTART18       -- (T) memstart at 18
               | ATTRIB_MEMSTART28       -- (T) memstart at 28
               | ATTRIB_MEMSTARTLEQ28    -- (T) either 18 or 28
               | ATTRIB_INSTR_IO         -- (T) direct channel I/O
               | ATTRIB_CALL_IO          -- (T) I/O as calls
               | ATTRIB_FPU_CALLING      -- (T) fpu calling convention
               | ATTRIB_NON_FPU_CALLING  -- (T) non fpu calling convention
               | ATTRIB_UNIVERSAL        -- (T) compatible with halt or stop
               | ATTRIB_HALT             -- (T) halt processor on error
               | ATTRIB_STOP             -- (T) stop process on error
               | H_ATTRIB_UNIVERSAL      -- (H) compatible with halt or stop
               | H_ATTRIB_HALT           -- (H) halt processor on error
               | H_ATTRIB_STOP           -- (H) stop process on error
               | H_ATTRIB_RESERVED_SET   -- (H)
               ;
```

The use of the **(T)** and **(H)** is the same as for **TRANS_FUNC**.

Each architecture is now discussed in more detail.

3.4.1 T Architecture

The subset of **TRANS_FUNC** applicable to the T architecture can be given by the following IDL definition:

```
T_TRANS_FUNC ::=   INSTR_CORE
                 | INSTR_FMUL
                 | INSTR_FP_SUPPORT
                 | INSTR_DUP
                 | INSTR_WSUBDB
                 | INSTR_MOVE2D
                 | INSTR_CRC
                 | INSTR_BITOPS
                 | INSTR_FPU_CORE
                 | INSTR_FPTESTERR
                 | INSTR_LDDEVID
                 | INSTR_DEBUG_SUPPORT
                 | INSTR_TIMER_DISABLE
                 | INSTR_LDMEMSTARTVAL
                 | INSTR_POP
                 | INSTR_RESERVED_SET
                 ;
```

Each of these values represents some functionality of the transputer that will execute the code. The code can be executed only if the target cpu implements ALL the functionality in the set, or a super-set of it. If a value is not in the set, then the code does not use it (eg. no floating point operations). It is expected that linkers will do some checking on these values for consistency. Below is listed each attribute and the instruction capabilities it represents.

INSTR_CORE	j	ldlp	pfix	ldnl	ldc
	ldnlp	nfix	ldl	adc	call
	cj	ajw	eqc	stl	stnl
	opr	and	or	xor	not
	shl	shr	add	sub	mul
	div	rem	gt	diff	sum
	prod	ladd	lsub	lsum	ldiff
	lmul	lshl	lshr	norm	rev
	xword	cword	xdble	csngl	mint
	bsub	wsub	bcnt	wcnt	lb
	sb	move	ldtimer	tin	talt
	taltwt	enbt	dist	in	out
	outword	outbyte	resetch	alt	altwt
	altend	enbs	diss	enbc	disc
	ret	ldpi	gajw	gcall	lend
	startp	endp	runp	stopp	ldpri
	csub0	ccnt1	testerr	seterr	stoperr
	clrhalterr	sethalterr	testhalterr	testpranal	saveh
	savel	sthf	sthb	stlf	stlb
	sttimer				
INSTR_FMUL	fmul				
INSTR_FP_SUPPORT	unpacksn	roundsn	postnormsn	ldinf	cflerr
INSTR_DUP	dup				
INSTR_WSUBDB	wsubdb				
INSTR_MOVE2D	move2dinit	move2dall	move2dnonzero	move2dzero	
INSTR_CRC	crcword	crcbyte			
INSTR_BITOPS	bitcnt	bitrevword	bitrevnbits		

INSTR_FPU_CORE	fpldnlsn	fpldnldb	fpldnlsni	fpldnldbi	fpldzerosn
	fpldzerodb	fpldnladdsn	fpldnladddb	fpldnlmulsn	fpldnlmuldb
	fpstnlsn	fpstnldb	fpstnli32	fpentry	fprev
	fpdup	fpurn	fpurz	fpurp	fpurm
	fpgt	fpeq	fpordered	fpnan	fpnotfinite
	fpuchki32	fpuchki64	fpur32tor64	fpur64tor32	fprtoi32
	fpi32tor32	fpi32tor64	fpb32tor64	fpunoround	fpint
	fpadd	fpsub	fpmul	fpdiv	fpuabs
	fpremfirst	fpremstep	fpusqrtfirst	fpusqrtstep	fpusqrtlast
	fpuexpinc32	fpuexpdec32	fpumulby2	fpudivby2	fpchkerror
	fpuseterror	fpuclearerror			
INSTR_FPTESTERR	fptesterr				
INSTR_LDDEVID	lddevid				
INSTR_DEBUG_SUPPORT	break	clrj0break	setj0break	testj0break	
INSTR_TIMER_DISABLE	timerdisableh	timerdisablel	timerenableh	timerenablel	
INSTR_LDMEMSTARTVAL	ldmemstartval				
INSTR_POP	pop				

INSTR_RESERVED_SET is a number which defines a bit pattern. This bit pattern defines which remaining instruction bits must be set. These bits have no defined meaning but allow for future extension.

Note that bits that are unused by any of the above instructions are reserved as being cleared.

The subset of **ATTRIBUTES** used by the T architecture is as follows:

```
T_ATTRIBUTES ::=   ATTRIB_WORD_16          -- (T) 16 bit words
                 | ATTRIB_WORD_32          -- (T) 32 bit words
                 | ATTRIB_MEMSTART18       -- (T) memstart at 18
                 | ATTRIB_MEMSTART28       -- (T) memstart at 28
                 | ATTRIB_MEMSTARTLEQ28    -- (T) either 18 or 28
                 | ATTRIB_INSTR_IO         -- (T) direct channel I/O
                 | ATTRIB_CALL_IO          -- (T) I/O as calls
                 | ATTRIB_FPU_CALLING      -- (T) fpu calling convention
                 | ATTRIB_NON_FPU_CALLING  -- (T) non fpu calling convention
                 | ATTRIB_UNIVERSAL        -- (T) compatible with halt or stop
                 | ATTRIB_HALT             -- (T) halt processor on error
                 | ATTRIB_STOP             -- (T) stop process on error
                 ;
```

ATTRIBUTES is a set, as in **TRANSPUTER_FUNCTION**. It can be further sub-divided into groups, each of which is itself a set. A single item from each subset must be present to make a valid attribute set. These further subsets are as follows:

```
AT_WORD_LENGTH  ::=   ATTRIB_WORD_16          -- 16 bit words
                    | ATTRIB_WORD_32          -- 32 bit words
                    ;
```

```
AT_MEMSTART      ::=   ATTRIB_MEMSTART18         -- memstart at 18
                     | ATTRIB_MEMSTART28         -- memstart at 28
                     | ATTRIB_MEMSTARTLEQ28      -- don't care
                     ;

AT_ERR_MODE      ::=   ATTRIB_UNIVERSAL          -- compatible with halt or stop
                     | ATTRIB_HALT               -- halt processor on error
                     | ATTRIB_STOP               -- stop process on error
                     ;

AT_IO_MODE       ::=   ATTRIB_INSTR_IO           -- direct channel I/O
                     | ATTRIB_CALL_IO            -- I/O as calls
                     ;

AT_FP_MODE       ::=   ATTRIB_FPU_CALLING        -- fpu calling convention
                     | ATTRIB_NON_FPU_CALLING    -- non fpu calling convention
                     ;

ATTRIB_RESERVED_SET  ::=   reserved_bits : number; -- reserved for extension
```

AT_WORD_LENGTH defines the word length to be either 16 or 32 bits.

AT_MEMSTART defines the start of memory to be 18, 28 or 'don't care'.

AT_ERR_MODE defines the error behavior.

AT_IO_MODE defines the channel communication behaviour of a module as being implimented directly by instructions or by calls to some external routine which may alter the routing of the channel.

AT_FP_MODE defines floating point calling conventions, ie. whether the fpu stack is used to hold parameters and return values.

ATTRIB_RESERVED_SET is a number which defines a bit pattern. This bit pattern defines which remaining attribute bits must be set. These bits have no defined meaning but allow for future extension.

Note that bits that are unused by any of the above attribute groups are reserved and defined as being cleared.

The above sets specify the relevant characteristics of all current T transputers with room for extension in the future. Below is listed the bit masks of these transputers and supported transputer classes. The **ATTRIBUTES** given below require that relevent items from **AT_ERR_MODE** and **AT_IO_MODE** be added to reflect the attributes of the code. (Note that these definitions are not IDL).

```
T212_INSTR       =>    ARCH_T
                     | INSTR_CORE
                     | INSTR_RESERVED_SET

T212_ATTRIB      =>    ATTRIB_WORD16
                     | ATTRIB_MEMSTART18
                     | ATTRIB_NON_FPU_CALLING
                     | ATTRIB_RESERVED_SET

T222_INSTR       =>    T212_INSTR

T222_ATTRIB      =>    T212_ATTRIB
```

```
T225_INSTR      =>   ARCH_T
                   | INSTR_CORE
                   | INSTR_DUP
                   | INSTR_CRC
                   | INSTR_BITOPS
                   | INSTR_LDDEVID
                   | INSTR_DEBUG_SUPPORT
                   | INSTR_TIMER_DISABLE
                   | INSTR_LDMEMSTARTVAL
                   | INSTR_POP
                   | INSTR_WSUBDB
                   | INSTR_RESERVED_SET

T225_ATTRIB     =>   ATTRIB_WORD_16
                   | ATTRIB_MEMSTART18
                   | ATTRIB_NON_FPU_CALLING
                   | ATTRIB_RESERVED_SET

T400_INSTR      =>   T425_INSTR

T400_ATTRIB     =>   T425_ATTRIB

T414_INSTR      =>   ARCH_T
                   | INSTR_CORE
                   | INSTR_FMUL
                   | INSTR_FP_SUPPORT
                   | INSTR_RESERVED_SET

T414_ATTRIB     =>   ATTRIB_WORD32
                   | ATTRIB_MEMSTART18
                   | ATTRIB_NON_FPU_CALLING
                   | ATTRIB_RESERVED_SET

T425_INSTR      =>   ARCH_T
                   | INSTR_CORE
                   | INSTR_FMUL
                   | INSTR_FP_SUPPORT
                   | INSTR_DUP
                   | INSTR_WSUBDB
                   | INSTR_MOVE2D
                   | INSTR_CRC
                   | INSTR_BITOPS
                   | INSTR_FPTESTERR
                   | INSTR_LDDEVID
                   | INSTR_DEBUG_SUPPORT
                   | INSTR_TIMER_DISABLE
                   | INSTR_LDMEMSTARTVAL
                   | INSTR_POP
                   | INSTR_RESERVED_SET

T425_ATTRIB     =>   ATTRIB_WORD32
                   | ATTRIB_MEMSTART28
                   | ATTRIB_NON_FPU_CALLING
                   | ATTRIB_RESERVED_SET
```

```
T800_INSTR      =>   ARCH_T
                   | INSTR_CORE
                   | INSTR_FMUL
                   | INSTR_DUP
                   | INSTR_WSUBDB
                   | INSTR_MOVE2D
                   | INSTR_CRC
                   | INSTR_BITOPS
                   | INSTR_FPU_CORE
                   | INSTR_FPTESTERR
                   | INSTR_RESERVED_SET

T800_ATTRIB     =>   ATTRIB_WORD32
                   | ATTRIB_MEMSTART28
                   | ATTRIB_FPU_CALLING
                   | ATTRIB_RESERVED_SET

T801_INSTR      =>   ARCH_T
                   | INSTR_CORE
                   | INSTR_FMUL
                   | INSTR_DUP
                   | INSTR_WSUBDB
                   | INSTR_MOVE2D
                   | INSTR_CRC
                   | INSTR_BITOPS
                   | INSTR_FPU_CORE
                   | INSTR_FPTESTERR
                   | INSTR_LDDEVID
                   | INSTR_DEBUG_SUPPORT
                   | INSTR_TIMER_DISABLE
                   | INSTR_LDMEMSTARTVAL
                   | INSTR_POP
                   | INSTR_RESERVED_SET

T801_ATTRIB     =>   ATTRIB_WORD32
                   | ATTRIB_MEMSTART28
                   | ATTRIB_FPU_CALLING
                   | ATTRIB_RESERVED_SET

T805_INSTR      =>   T801_INSTR

T805_ATTRIB     =>   T801_ATTRIB

TA_INSTR        =>   ARCH_T
                   | INSTR_CORE
                   | INSTR_FMUL
                   | INSTR_RESERVED_SET

TA_ATTRIB       =>   ATTRIB_WORD32
                   | ATTRIB_MEMSTARTLEQ28
                   | ATTRIB_NON_FPU_CALLING
                   | ATTRIB_RESERVED_SET
```

```
TB_INSTR        =>    ARCH_T
                    | INSTR_CORE
                    | INSTR_FMUL
                    | INSTR_FP_SUPPORT
                    | INSTR_RESERVED_SET

TB_ATTRIB       =>    ATTRIB_WORD32
                    | ATTRIB_MEMSTARTLEQ28
                    | ATTRIB_NON_FPU_CALLING
                    | ATTRIB_RESERVED_SET
```

3.4.2 H Architecture

It is expected that for the H architecture the **TRANS_FUNCTION** and **ATTRIBUTES** words can be thought more of a set of 64 bits rather than two disjoint 32 bit sets although the terms **TRANS_FUNCTION** and **ATTRIBUTES** are still used for clarity.

At this time there is only one example of the H architecture and so most information required can be gleaned from the H architecture bit alone. However for maximum future expansion half of the unused bits are defined set and half defined unset.

The subset of the **TRANS_FUNC** set applicable to the H architecture can be represented thus:

```
H_TRANS_FUNCTION  ::=  H_INSTR_RESERVED_SET;
```

No specific functionality bits exist because at present the ARCH_H bit is used to define the entire functionality of the H1. This definition is expandable in that future H series transputers with more or less functionality can be easily represented while remaining backwards compatible. As for T architecture, it is expected that linkers will do some checking on these values for consistency.

H_INSTR_RESERVED_SET is a number which defines a bit pattern. This bit pattern defines which remaining instruction bits must be set. These bits have no defined meaning but allow for future extension.

The subset of the **ATTRIBUTES** set applicable to the H1 is as follows:

```
H_ATTRIBUTES ::=   H_ATTRIB_UNIVERSAL      -- (H) compatible with halt or stop
                 | H_ATTRIB_HALT           -- (H) halt processor on error
                 | H_ATTRIB_STOP           -- (H) stop process on error
                 | H_ATTRIB_RESERVED_SET   -- (H)
                 ;
```

As for the T architecture this set can be further subdivided as follows:

```
H_AT_ERR_MODE ::=   H_ATTRIB_UNIVERSAL      -- compatible with halt or stop
                  | H_ATTRIB_HALT           -- halt processor on error
                  | H_ATTRIB_STOP           -- stop process on error
                  ;

H_ATTRIB_RESERVED_SET ::= reserved_bits : number; -- reserved for extension
```

Since there is currently only one example of an H architecture processor there is only one variable which needs representing in the **ATTRIBUTES** word, that is, the error mode.

H_AT_ERR_MODE defines the error behavior.

H_ATTRIB_RESERVED_SET is a number which defines a bit pattern. This bit pattern defines which remaining attribute bits must be set. These bits have no defined meaning but allow for future extension.

Note that bits that are unused by any of the above attribute groups are reserved and defined as being cleared.

The above sets specify the relevant characteristics of all current H transputers with room for extension in the future. Below is listed the definition of these transputers and supported transputer classes. The **ATTRIBUTES** given below require that relevent items from **H_AT_ERR_MODE** be added to reflect the attributes of the code. (Note these definitions are not IDL)

```
H1_INSTR        =>    ARCH_H
                    | H_INSTR_RESERVED_SET

H1_ATTRIB       =>    H_ATTRIB_RESERVED_SET
```

4 Alphabetic List

4.1 Syntax of TCOFF

```
adjust_point    =>    aj_header      : header (ADJUST_POINT_TAG),
                      aj_offset      : VALUE ;

adjust_prefix   =>    ap_value_tag    : AP_VALUE_TAG,
                      ap_operand      : VALUE ;

align           =>    al_header      : header (ALIGN_TAG),
                      al_modulo      : number ;

ARCHITECTURE    ::=   ARCH_T
                    | ARCH_H
                    | ARCH_RESERVED_1
                    | ARCH_RESERVED_2
                    | ARCH_RESERVED_3
                    ;

ATTRIBUTES      ::=   ATTRIB_WORD_16          -- (T) 16 bit words
                    | ATTRIB_WORD_32          -- (T) 32 bit words
                    | ATTRIB_MEMSTART18       -- (T) memstart at 18
                    | ATTRIB_MEMSTART28       -- (T) memstart at 28
                    | ATTRIB_MEMSTARTLEQ28    -- (T) either 18 or 28
                    | ATTRIB_INSTR_IO         -- (T) direct channel I/O
                    | ATTRIB_CALL_IO          -- (T) I/O as calls
                    | ATTRIB_FPU_CALLING      -- (T) fpu calling convention
                    | ATTRIB_NON_FPU_CALLING  -- (T) non fpu calling convention
                    | ATTRIB_UNIVERSAL        -- (T) compatible with halt or stop
                    | ATTRIB_HALT             -- (T) halt processor on error
                    | ATTRIB_STOP             -- (T) stop process on error
                    | H_ATTRIB_UNIVERSAL      -- (H) compatible with halt or stop
                    | H_ATTRIB_HALT           -- (H) halt processor on error
                    | H_ATTRIB_STOP           -- (H) stop process on error
                    | H_ATTRIB_RESERVED_SET   -- (H)
                    ;

BOOLEAN         ::=   BOOL_TRUE
                    | BOOL_FALSE ;

CODED_NUMBER    ::=   simple_number
                    | prefix_1_number
                    | prefix_2_number
                    | prefix_4_number
                    | prefix_8_number
                    ;

comment         =>    cm_header      : header (COMMENT_TAG),
                      cm_copy        : BOOLEAN,
                      cm_print       : BOOLEAN,
                      cm_text        : string ;
```

```
constant         =>    co_value_tag  : CO_VALUE_TAG,
                       co_value      : number ;

define_label     =>    dl_header     : header (DEFINE_LABEL_TAG),
                       dl_ident      : ident ;

define_main      =>    dm_header     : header (DEFINE_MAIN_TAG),
                       dm_entry      : ident ;

define_symbol    =>    ds_header     : header (DEFINE_SYMBOL_TAG),
                       ds_ident      : ident,
                       ds_value      : VALUE ;

descriptor       =>    de_header     : header (DESCRIPTOR_TAG),
                       de_symbol     : ident,
                       de_language   : LANGUAGE,
                       de_string     : string ;

directive        =>    dr_header     : header,
                       dr_command    : OPT SEQUENCE OF BYTE ;

DIRECTIVE_TAG    ::=   LINKABLE_TAG
                     | LINKED_UNIT_TAG
                     | START_MODULE_TAG
                     | END_MODULE_TAG
                     | SET_LOAD_POINT_TAG
                     | ADJUST_POINT_TAG
                     | LOAD_TEXT_TAG
                     | LOAD_PREFIX_TAG
                     | LOAD_EXPR_TAG
                     | LOAD_ZEROS_TAG
                     | ALIGN_TAG
                     | SECTION_TAG
                     | SYMBOL_TAG
                     | DEFINE_MAIN_TAG
                     | SPECIFIC_SYMBOL_TAG
                     | LOCAL_SYMBOLS_TAG
                     | DEFINE_LABEL_TAG
                     | DEFINE_SYMBOL_TAG
                     | DESCRIPTOR_TAG
                     | KILL_ID_TAG
                     | PATCH_TAG
                     | REP_START_TAG
                     | REP_END_START
                     | COMMENT_TAG
                     | MESSAGE_TAG
                     | VERSION_TAG
                     | LIB_INDEX_START_TAG
                     | LIB_INDEX_END_TAG
                     | INDEX_ENTRY_TAG
                     ;

end_module       =>    em_header     : header (END_MODULE_TAG) ;
```

```
ERROR_LEVEL     ::=   NORMAL_MSG
                    | WARNING_MSG
                    | ERROR_MSG
                    ;

expression      ::=   ex_oper_tag   : OPER_VALUE_TAG,
                      ex_operand1   : VALUE,
                      ex_operand2   : VALUE ;

header (TAG)    =>    hd_tag        : DIRECTIVE_TAG = TAG ;
                      hd_length     : length ;

ident           =>    id_index      : number ;

index_entry     =>    ie_header     : header (INDEX_ENTRY_TAG),
                      ie_position   : INT32 (unsigned),
                      ie_cpus       : trans_function,
                      ie_attrib     : SET OF ATTRIBUTES,
                      ie_language   : LANGUAGE,
                      ie_descriptor : string,
                      ie_symbol     : string ;

instruction     =>    in_op_code    : number ;

kill_id         =>    ki_header     : header (KILL_ID_TAG),
                      ki_ident      : ident ;

LANGUAGE        ::=   LANG_NOT_KNOWN
                    | LANG_LINKED
                    | LANG_OCCAM
                    | LANG_OCCAM_HARNESS
                    | LANG_ANSI_C
                    | LANG_FORTRAN_77
                    | LANG_ISO_PASCAL
                    | LANG_MODULA_2
                    | LANG_ADA
                    | LANG_ASSEMBLER
                    ;

lib_index       =>    li_start      : lib_index_start,
                      li_entries    : OPT SEQUENCE OF index_entry,
                      li_end        : lib_index_end ;

lib_index_end   =>    le_header     : header (LIB_INDEX_END_TAG) ;

lib_index_start =>    ls_header     : header (LIB_INDEX_START_TAG),

library         =>    lb_linkable   : linkable,
                      lb_lookup     : lib_index,
                      lb_version    : OPT version,
                      lb_body       : OPT SEQUENCE OF module ;
```

```
LINK_COMMAND    ::=  SIMPLE_DIRECTIVE
                   | replicator
                   | module
                   ;

linkable        =>   ln_header       : header (LINKABLE_TAG) ;

linked_file     =>   lf_linked       : linked,
                     lf_directives   : OPT SEQUENCE OF directive ;

LINKED_FILE     =>   lk_module_list
                   | lk_library
                   ;

linked          =>   lk_header       : header (LINKED_UNIT_TAG) ;

lk_library      =>   ll_linkable     : linked,
                     ll_lookup       : lib_index,
                     ll_version      : OPT version,
                     ll_body         : OPT SEQUENCE OF module ;

lk_module_list  =>   lm_body         : OPT SEQUENCE OF lk_unit ;

lk_unit         =>   lu_linkable     : linked,
                     lu_body         : module ;

load_point      =>   lp_value_tag    : LP_VALUE_TAG ;

load_prefix     =>   lx_header       : header (LOAD_PREFIX_TAG),
                     lx_size         : number,
                     lx_value        : VALUE,
                     lx_instr        : instruction ;

load_text       =>   lt_header       : header (LOAD_TEXT_TAG),
                     lt_text         : string ;

load_value      =>   lv_header       : header (LOAD_EXPR_TAG),
                     lv_size         : number,
                     lv_value        : VALUE ;

load_zeros      =>   lz_header       : header (LOAD_ZEROS_TAG),
                     lz_count        : number ;

local_symbols   =>   lo_header       : header (LOCAL_SYMBOLS_TAG),
                     lo_count        : number ;

message         =>   ms_header       : header (MESSAGE_TAG),
                     ms_level        : ERROR_LEVEL,
                     ms_text         : string ;

module          =>   md_start        : start_module,
                     md_body         : OPT SEQUENCE OF LINK_COMMAND,
                     md_end          : end_module ;

module_list     =>   ml_body         : OPT SEQUENCE OF unit ;
```

```
number          =>    nm_sign       : OPT SIGN_INDICATOR,
                      nm_pos_number : CODED_NUMBER ;

object_file     =>    of_linkable   : linkable,
                      of_directives : OPT SEQUENCE OF directive ;

OBJECT_FILE     ::=   module_list
                    | library
                    ;

OPER_VALUE_TAG  ::=   PLUS_OP
                    | MINUS_OP
                    | TIMES_OP
                    | DIVIDE_OP
                    | REM_OP
                    | MAX_OP
                    | MIN_OP
                    ;

patch           =>    pt_header     : header (PATCH_TAG),
                      pt_location   : ident,
                      pt_size       : number,
                      pt_value      : VALUE ;

PATCH_TAG       ::=   BYTE_PATCH_TAG
                    | WORD_PATCH_TAG
                    ;
prefix_1_number =>    p1_prefix       : PFX_1_TAG,
                      p1_number       : BYTE ;

prefix_2_number =>    p2_prefix       : PFX_2_TAG,
                      p2_number       : INT16 ;

prefix_4_number =>    p4_prefix       : PFX_4_TAG,
                      p4_number       : INT32 ;

prefix_8_number =>    p8_prefix       : PFX_8_TAG,
                      p8_number       : INT64 ;

rep_end         =>    re_header     : header (REP_END_TAG) ;

rep_start       =>    rs_header     : header (REP_START_TAG),
                      rs_count      : number ;

REPL_DIR        ::=   set_load_point
                    | adjust_point
                    | section
                    | symbol
                    | local_symbols
                    | patch
                    | load_text
                    | load_prefix
                    | load_value
                    | load_zeros
```

```
                          | align
                          | define_label
                          | define_symbol
                          | kill_id
                          | replicator
                          | comment
                          | message
                          ;

replicator      =>     rp_start       : rep_start,
                       rp_body        : OPT SEQUENCE OF REPL_DIR,
                       rp_end         : rep_end ;

section         =>     se_header      : header (SECTION_TAG),
                       se_section     : SET OF SECTION_TYPE,
                       se_usage       : SET OF USAGE,
                       se_symbol      : string ;

section_size    =>     ss_value_tag   : SS_VALUE_TAG,
                       ss_identifier  : ident ;

SECTION_TYPE    ::=    WRITE_SECTION
                     | READ_SECTION
                     | EXECUTE_SECTION
                     | DEBUG_SECTION
                     | VIRTUAL_SECTION
                     ;

set_load_point  =>     sl_header      : header (SET_LOAD_POINT_TAG),
                       sl_location    : ident ;

SIMPLE_DIRECTIVE ::=     set_load_point
                       | adjust_point
                       | load_text
                       | load_prefix
                       | load_value
                       | load_zeros
                       | align
                       | symbol_id
                       | define_main
                       | specific_symbol
                       | local_symbols
                       | define_label
                       | define_symbol
                       | descriptor
                       | kill_id
                       | patch
                       | comment
                       | message
                       | version
                       ;

simple_number   =>     sn_number      : <0 .. 250>(BYTE) ;
```

```
specific_symbol =>    sp_header      : header (SPECIFIC_SYMBOL_TAG),
                      sp_usage       : SET OF USAGE,
                      sp_symbol      : string,
                      sp_origin      : ident ;

start_module    =>    sm_header      : header (START_MODULE_TAG),
                      sm_cpus        : trans_function,
                      sm_attrib      : SET OF ATTRIBUTES,
                      sm_language    : LANGUAGE,
                      sm_name        : string ;

string          =>    st_length      : number,
                      st_chars       : OPT SEQUENCE OF BYTE ;

symbol          =>    sy_header      : header (SECTION_TAG),
                      sy_usage       : SET OF USAGE,
                      sy_symbol      : string ;

symbol_value    =>    sv_value_tag   : SV_VALUE_TAG,
                      sv_identifier  : ident ;

TRANS_FUNC     ::=     INSTR_CORE           -- (T)
                     | INSTR_FMUL           -- (T)
                     | INSTR_FP_SUPPORT     -- (T)
                     | INSTR_DUP            -- (T)
                     | INSTR_WSUBDB         -- (T)
                     | INSTR_MOVE2D         -- (T)
                     | INSTR_CRC            -- (T)
                     | INSTR_BITOPS         -- (T)
                     | INSTR_FPU_CORE       -- (T)
                     | INSTR_FPTESTERR      -- (T)
                     | INSTR_LDDEVID        -- (T)
                     | INSTR_DEBUG_SUPPORT  -- (T)
                     | INSTR_TIMER_DISABLE  -- (T)
                     | INSTR_LDMEMSTARTVAL  -- (T)
                     | INSTR_POP            -- (T)
                     | INSTR_RESERVED_SET   -- (T)
                     | H_INSTR_RESERVED_SET -- (H)
                     ;

trans_function  =>    tf_arch           : ARCHITECTURE;
                      tf_functionality : SET OF TRANS_FUNC;

unit            =>    un_linkable    : linkable,
                      un_body        : OPT SEQUENCE OF module ;

USAGE            ::=  LOCAL_USAGE
                    | EXPORT_USAGE
                    | IMPORT_USAGE
                    | WEAK_USAGE
                    | CONDITIONAL_USAGE
                    | PROVISIONAL_USAGE
                    | UNINDEXED_USAGE
                    | ORIGIN_USAGE
                    ;
```

```
VALUE           ::=   constant
                    | load_point
                    | symbol_value
                    | section_size
                    | expression
                    | word_length
                    | adjust_prefix
                    ;

version         =>    vn_header     : header (VERSION_TAG),
                      vn_tool_id    : string,
                      vn_origin     : string ;

word_length     =>    WL_VALUE_TAG ;
```

4.2 Bit Representations

```
ARCHITECTURE  (part of TRANS_FUNCTION word)

  ARCH_T              ::=    0x100000  (number) ;
  ARCH_H              ::=    0x200000  (number) ;
  ARCH_RESERVED_1     ::=    0x400000  (number) ;
  ARCH_RESERVED_2     ::=    0x800000  (number) ;
  ARCH_RESERVED_3     ::=    0x1000000 (number) ;

BOOLEAN

  BOOL_FALSE          ::=    0 (number) ;
  BOOL_TRUE           ::=    1 (number) ;


DIRECTIVE_TAG

  LINKABLE_TAG         ::=    1 (number) ;
  START_MODULE_TAG     ::=    2 (number) ;
  END_MODULE_TAG       ::=    3 (number) ;
  SET_LOAD_POINT_TAG   ::=    4 (number) ;
  ADJUST_POINT_TAG     ::=    5 (number) ;
  LOAD_TEXT_TAG        ::=    6 (number) ;
  LOAD_PREFIX_TAG      ::=    7 (number) ;
  LOAD_EXPR_TAG        ::=    8 (number) ;
  LOAD_ZEROS_TAG       ::=    9 (number) ;
  ALIGN_TAG            ::=   10 (number) ;
  SECTION_TAG          ::=   11 (number) ;
  DEFINE_MAIN_TAG      ::=   12 (number) ;
  LOCAL_SYMBOLS_TAG    ::=   13 (number) ;
  DEFINE_LABEL_TAG     ::=   14 (number) ;
  DEFINE_SYMBOL_TAG    ::=   15 (number) ;
  KILL_ID_TAG          ::=   16 (number) ;
  BYTE_PATCH_TAG       ::=   17 (number) ;
  REP_START_TAG        ::=   18 (number) ;
  REP_END_TAG          ::=   19 (number) ;
  COMMENT_TAG          ::=   20 (number) ;
  MESSAGE_TAG          ::=   21 (number) ;
  LIB_INDEX_START_TAG  ::=   22 (number) ;
  LIB_INDEX_END_TAG    ::=   23 (number) ;
  INDEX_ENTRY_TAG      ::=   24 (number) ;
  WORD_PATCH_TAG       ::=   25 (number) ;
  DESCRIPTOR_TAG       ::=   26 (number) ;
  VERSION_TAG          ::=   27 (number) ;
  LINKED_UNIT_TAG      ::=   28 (number) ;
  SYMBOL_TAG           ::=   30 (number) ;
  SPECIFIC_SYMBOL_TAG  ::=   31 (number) ;

ERROR_LEVEL

  NORMAL_MSG       ::=    1 (number) ;
  WARNING_MSG      ::=    2 (number) ;
  ERROR_MSG        ::=    3 (number) ;
```

```
H_ATTRIBUTES

  H_ATTRIB_UNIVERSAL     ::= 0x0     (number);
  H_ATTRIB_HALT          ::= 0x80    (number);
  H_ATTRIB_STOP          ::= 0x100   (number);

  H_ATTRIB_RESERVED_SET  ::= 0x3F87F (number);

H_TRANS_FUNCTION

  H_INSTR_RESERVED_SET   ::= 0x3FFF  (number) ;

LANGUAGE

  LANG_NOT_KNOWN      ::=   1 (number);
  LANG_LINKED         ::=   2 (number);
  LANG_OCCAM          ::=   3 (number);
  LANG_ANSI_C         ::=   4 (number);
  LANG_FORTRAN_77     ::=   5 (number);
  LANG_ISO_PASCAL     ::=   6 (number);
  LANG_MODULA_2       ::=   7 (number);
  LANG_ADA            ::=   8 (number);
  LANG_ASSEMBLER      ::=   9 (number);
  LANG_OCCAM_HARNESS  ::=  10 (number);

NUMBERS
  PFX_1_NUMBER    ::=  251 (BYTE) ;
  PFX_2_NUMBER    ::=  252 (BYTE) ;
  PFX_4_NUMBER    ::=  253 (BYTE) ;
  PFX_8_NUMBER    ::=  254 (BYTE) ;
  SIGN_INDICATOR  ::=  255 (BYTE) ;

OPER_VALUE_TAG

  PLUS_OP         ::=    6 (number) ;
  MINUS_OP        ::=    7 (number) ;
  TIMES_OP        ::=    8 (number) ;
  DIVIDE_OP       ::=    9 (number) ;
  REM_OP          ::=   10 (number) ;
  MAX_OP          ::=   11 (number) ;
  MIN_OP          ::=   12 (number) ;

SECTION_TYPE

  WRITE_SECTION   ::=   0x01 (number) ;
  READ_SECTION    ::=   0x02 (number) ;
  EXECUTE_SECTION ::=   0x04 (number) ;
  DEBUG_SECTION   ::=   0x08 (number) ;
  VIRTUAL_SECTION ::=   0x10 (number) ;

T_ATTRIBUTES

  ATTRIB_WORD_16         ::= 0x1     (number);
  ATTRIB_WORD_32         ::= 0x2     (number);
```

```
  ATTRIB_MEMSTART18       ::= 0x8      (number);
  ATTRIB_MEMSTART28       ::= 0x10     (number);
  ATTRIB_MEMSTARTLEQ28    ::= 0x0      (number);

  ATTRIB_UNIVERSAL        ::= 0x0      (number);
  ATTRIB_HALT             ::= 0x80     (number);
  ATTRIB_STOP             ::= 0x100    (number);

  ATTRIB_INSTR_IO         ::= 0x800    (number);
  ATTRIB_CALL_IO          ::= 0x0      (number);

  ATTRIB_NON_FPU_CALLING  ::= 0x40000  (number);
  ATTRIB_FPU_CALLING      ::= 0x80000  (number);

  ATTRIB_RESERVED_SET     ::= 0x3E040  (number);

T_TRANS_FUNCTION

  INSTR_CORE           ::= 0x1       (number) ;
  INSTR_FMUL           ::= 0x2       (number) ;
  INSTR_FP_SUPPORT     ::= 0x4       (number) ;
  INSTR_DUP            ::= 0x8       (number) ;
  INSTR_WSUBDB         ::= 0x10      (number) ;
  INSTR_MOVE2D         ::= 0x20      (number) ;
  INSTR_CRC            ::= 0x40      (number) ;
  INSTR_BITOPS         ::= 0x80      (number) ;
  INSTR_FPU_CORE       ::= 0x100     (number) ;
  INSTR_FPTESTERR      ::= 0x200     (number) ;
  INSTR_LDDEVID        ::= 0x400     (number) ;
  INSTR_DEBUG_SUPPORT  ::= 0x800     (number) ;
  INSTR_TIMER_DISABLE  ::= 0x1000    (number) ;
  INSTR_LDMEMSTARTVAL  ::= 0x2000    (number) ;
  INSTR_POP            ::= 0x4000    (number) ;
  INSTR_RESERVED_SET   ::= 0xF8000   (number) ;

USAGE

  LOCAL_USAGE        ::=   0x01 (number) ;
  EXPORT_USAGE       ::=   0x02 (number) ;
  IMPORT_USAGE       ::=   0x04 (number) ;
  WEAK_USAGE         ::=   0x08 (number) ;
  CONDITIONAL_USAGE  ::=   0x10 (number  ;
  UNINDEXED_USAGE    ::=   0x20 (number) ;
  PROVISIONAL_USAGE  ::=   0x40 (number) ;
  ORIGIN_USAGE       ::=   0x80 (number) ;

VALUE_TAG

  CO_VALUE_TAG     ::=    1 (number) ;
  LP_VALUE_TAG     ::=    2 (number) ;
  SV_VALUE_TAG     ::=    3 (number) ;
  SS_VALUE_TAG     ::=    4 (number) ;
  WL_VALUE_TAG     ::=    5 (number) ;
  AP_VALUE_TAG     ::=   13 (number) ;
```

5 References

[L83] D.A. Lamb. Sharing Intermediate Representations - The Interface Description Language. Technical Report CS-83-129, Computer Science Department, Carnegie-Mellon University, May 1983.

[N86] J.R. Nestor, W.A. Wulf, D.A. Lamb. IDL - Interface Description Language: Formal Description. Technical Report, Software Engineering Institute, Pittsburgh, PA, Feb. 1986.

[INMOS '87] The Transputer Instruction Set - A Compiler Writers Guide. INMOS Ltd 72 TRN 119