occam compiler internal implementation manual (Haynes Manual)

Conor O’Neill

SW-0239-2

DRAFT 20 February, 1991

Contents

1 Introduction

This document describes the internal workings of the occam compiler which is supplied as part of the D4205, D5205, D6205, and D7205 occam 2 toolsets. It is intended to be read by programmers in order to be able to understand and maintain or extend the compiler. Therefore it assumes a fair knowledge of how compilers work in general, and of the occam language.

2 Compiler overview

The occam compiler is a multi-pass, hand-coded, compiler, written in ANSI C, targetted at the complete range of INMOS transputers, both 16-bit and 32-bit. It is designed to generate good code, to take fullest advantage of the transputer architecture. It does not perform local (basic block), global (procedural), or interprocedural optimisation.

3 Building the compiler

This section describes how to turn the sources into an executable compiler.

3.1 Source overview

The occam compiler is written in ANSI C. It consists of approximately 70,000 lines of source code, totalling 2.3 Megabytes, split over approximately 100 files. It is self-contained except for a library which performs calculations for constant folding; this library is shared with other programs. This library consists of about 5000 lines of source code, or 150 Kilobytes.

The occam compiler and occam configurer share the majority of their code. This document will not deal with the configurer-specific details. The two variants are generated by compiling with different command line switches, which control the C pre-processor. Makefiles are provided which automate the build system; different makefiles generate the compiler and configurer, though, again, the makefiles share common components.

3.2 Directory structure

The sources are held in two subdirectories, named ‘frontend’ and ‘backend’, which obviously correspond (more-or-less) to the divisions of the compiler. Most source files have an associated ‘`.h`’ header file in the same directory. The makefiles are held in the top level directory. Another subdirectory named ‘info’ holds miscellaneous information as ASCII text files.

3.3 Makefiles

In the top-level directory there are a collection of makefiles:

makefile	target
`makefile.s3c`	Sun 3 compiler executable (`oc`) compiled with `gcc`.
`makefile.s4c`	Sun 4 compiler executable (`oc`) compiled with `gcc`.
`makefile.tpc`	Transputer compiler bootable (`oc.btl`) compiled with `icc`.

The equivalent makefiles, with ending character '`v`' rather than '`c`', create the configurers.

These share many common sub-files, all called `makefile.*`, which are accessed by `includes` from the top-level makefiles.

Note that to create a Sun 3 compiler you must build on a Sun 3, similarly for a Sun 4. The transputer bootable can be built by either; all you need is the INMOS `icc` cross compiler.

3.4 External dependencies

The compiler uses various external header files which define its interfaces, plus some standard INMOS header files.

- `imstype.h`
  This is a header file containing definitions for standard sized types, (eg. `BIT32`), for each compilation environment.

- `imsmisc.h`
  This is a header file defining miscellaneous stuff like `PUBLIC`, `PRIVATE`, etc, for each compilation environment.

- `imsstd.h`
  This is a header file which contains the non-portable bits and omissions for each compilation environment to make it look more like an ANSI system. (Eg. SUNOS has `unlink()` instead of `remove()`).

- `tcoff.h`
  This is a header file containing the values used to describe TCOFF object files.

- `debughdr.h`
  This is a header file containing the values used in the TCOFF debug information.

- `extlib.h`
  This is a header file listing the entrypoints of the `extlib` library, which is the library used for constant folding. The library file used is `extlib.a` when building UNIX executables, or `extlib.lib` for transputer or VMS executables.

The configurer uses extra external headers, which are used to interface to the common configurer backend. These are listed here for completeness, though they will not be described fully.

They are: `tagdefs.h`, `protos.h`, and `typedef.h`.

The configurer is linked with multiple extra files which also form part of the 'C style' configurer; these are not described here. These routines also require a `misclib` library to be linked.

3.5 Conditional compilation switches

There are a number of names which are designed to be specified on the command line in order to enable various facilities. Some of these are only of historical relevance.

- `CONFIG`
  Indicates that the configurer is to be built rather than the compiler.
- `DEC`
  Indicates that the compiler is being built under DEC's VMS operating system. Not fully supported (ie not tested).
- `GNU`
  Indicates that the compiler is being built to run under UNIX, using the `gcc` compiler.
- `HELIOS`
  Indicates that the compiler is being built to run under HELIOS. Not fully supported.
- `ICHECK`
  Indicates that just the syntax checker is to be built. No longer fully supported.
- `IMS`
  Indicates that the compiler is being built to run on a transputer, supported by `iserver`, using the INMOS C toolset (D4214 etc).
- `LLL`
  Indicates that the compiler is being built to run on a transputer, supported by `afserver`, using the old INMOS/3L C compiler (D414 etc). Not fully supported.
- `MSC`
  Indicates that the compiler is being built using Microsoft C to run under MS-DOS on a PC. Not supported due to memory limitations.
- `OC`
  Indicates that the compiler is to be built, not the configurer.
- `SUN`
  Indicates that the compiler is being built to run under SUNOS, using the SUN `cc` compiler.
- `TCOFF`
  Indicates that the compiler should read and produce TCOFF format object files (as against the 3L format (LFF) which was used with the D705 etc). This switch must always be specified.
- `VIRTUALCHANS`
  Indicates that the compiler should be built to be capable of inserting library calls for input and output operations. Since even if this is enabled, the library calls may be disabled by means of a command line switch when invoking the occam compiler, `VIRTUALCHANS` should always be specified.

Exactly one of `DEC`, `GNU`, `HELIOS`, `IMS`, `LLL`, `MSC`, or `SUN`, must be defined.

Exactly one of `CONFIG`, `ICHECK`, or `OC`, must be defined.

3.6 Compiler diagnostics

There are a number of 'hidden' command line options available for the developer to examine the state of the compiler. These are useful when maintaining the compiler. To see a complete list, run the compiler with command line option `z` only. The most useful options for program maintenance are described below:

Option	Description
`ZA`	Display backend diagnostics (workspace allocation, etc) and assembly output.
	If `ZT` is also specified, will display trees before and after mapping.
	No object file is produced.
`ZB`	Display assembly output. No object file is produced.
`ZD`	Display code just as it would be written to the object file.
	No object file is produced.
`ZL`	Display the output of the lexer.
`ZNO`	Do not display assembly *operands*; rely on the symbolic info.
`ZO`	Intersperse source and assembly output. Not fully supported.
`ZS`	Allocate workspace by scope rather than variable usage.
`ZSC`	Stop after the type checker.
`ZSM`	Stop after workspace mapping.
`ZSP`	Stop after parsing.
`ZST`	Stop after tree transformation phase.
`ZT`	Print the tree.
`ZU`	Stop after Usage checking; also enables usage checking diagnostics.

4 Compiler structure

The front end consists mainly of: A hand-written lexer, which tokenises the input source. A hand-written recursive-descent parser, which builds an abstract syntax tree. A type checker, which traverses the tree, resolves name scoping, and performs occam type checking, and checking of other language rules. A usage and alias checker, which is optionally invoked, to check that names are not aliased, and to check that variables and channels are used correctly in parallel constructs.

The back end consists mainly of: A tree transformation phase, which walks the tree and performs some simplifying transformations. It also performs INLINE expansion, and a few back-end specific transformations for the code generator. A mapping phase, which takes each procedure in turn, and inserts temporaries whenever they are required. It then allocates variables to workspace locations, based on their estimated dynamic frequency of use. A debug phase, which writes the symbolic debug information to the object file. A code generation phase generates code directly into a buffer (not to an assembly file). A code-crunching phase resolves branches, and creates optimal prefix sequences for the jumps, then writes the object file. This also writes the debug information specifying the relationship between code addresses and source lines.

4.1 Compiler frontend structure

The files which comprise the compiler frontend are grouped according to the phase of the compiler, ie. lexer, parser, checker, and usage checker. Each `*.c` file has a corresponding `*def.h` file. This contains declarations which are visible to other files in the same group, (eg, all the `chk*.c` files), but which should not be visible to other parts of the compiler. Each group also supplies header files which are visible to the whole compiler. The names for these are not particularly consistent. (Note: These ‘rules’ are broken more often than they are kept).

The frontend files are currently as follows:

Files	Phase	Description
`chk*.c`	Type checker	Type checking routines.
`chk*def.h`	Type checker	Headers for above.
`chkdef.h`	Type checker	External interface to type checker.
`chkerror.h`	Type checker	Error codes for type checker.
`conf*.c`	Configurer	Configuration routines.
`conf*def.h`	Configurer	Headers for above.
`confdef.h`	Configurer	External interface for configurer.
`confhdr.h`	Configurer	Miscellaneous internal data for configurer.
`desc*.c`		Object file reading/writing routines.
`desc*def.h`		Headers for above.
`deschdr.h`		External interface for object files.
`err*.c`		Error reporting routines.
`errhdr.h`		External interface for error reporting routines.
`includes.h`		Generic include file which simply **`#include`**s many others.
`inst.c`		Routines for **`ASM`** inserts and disassembly.
`instdef.h`		External interface for **`ASM`**, and disassembly.
`instruct.h`		Instruction encodings.
`lex*.c`	Lexer	Lexer routines, and nametable.
`lex*def.h`	Lexer	Headers for above.
`lexconst.h`	Lexer	Miscellaneous lexing constants.
`lexdef.h`	Lexer	External interface for lexer.
`lexerror.h`	Lexer	Error codes for lexer.
`lexhdr.h`	Lexer	Tree tag values.
`list*.*`		List manipulation routines.
`misc*.*`		Miscellaneous support for portability problems.
`nameshdr.h`		Definition of symbol table access functions.
`ochdr.h`		Miscellaneous host-specific checks - used by **`includes.h`**.
`popen*.*`		Opening files using path searching.
`predefhd.h`		Definitions for occam predefines.
`syn*.c`	Parser	Parsing routines.
`syn*def.h`	Parser	Headers for above.
`syndef.h`	Parser	External interface for parser.
`synerror.h`	Parser	Error codes for parser.
`treedef.h`		Definition of tree structure.
`use*.c`	Usage checker	Usage and alias checking routines.
`use*def.h`	Usage checker	Headers for above.
`usedef.h`	Usage checker	External interface for usage and alias checker.
`useerror.h`	Usage checker	Error codes for usage and alias checking.
`usehdr.h`	Usage checker	Common structures, etc, for usage checker.
`vti*.c`		Memory allocation and Tree manipulation routines.
`vti.h`		Abstract interface to tree structures, and external interface for tree manipulation routines.
`vtierror.h`		Error codes for tree manipulation routines.

4.2 Compiler backend structure

The files which comprise the backend are not so well partitioned. Because much of the same work must be performed in the mapping and code generation phases, they are generally combined into the same files.

The backend files are currently as follows:

File	Phase	Description
bind1*.*	Mapper	Top level process mapping routines.
bind2*.*	Mapper	Top level expression mapping routines.
bind3*.*	Mapper	Workspace allocation.
code*.*	Code cruncher	Code crunching and object file generation.
debug*.*		Debug information generation.
gen*.*	Mapper and Code generator	General routines.
generror.h	Mapper and Code generator	Backend error codes.
genhdr.h	Mapper and Code generator	Common structures, etc.
harndef.h		Details of command line option flags, etc.
harness.h		Startup harness and command line parser.
srcout*.*		Support for **ZO** command line option.
tran*.*	Tree transformation	General routines.
version.c		Compiler version string.

5 Compilation strategy and method

This section describes briefly what each phase does, in terms of its inputs and outputs, and its effects on other structures like the tree. Because of the complexity of the compiler, this description is not exhaustive!

5.1 Lexer

The lexer is called from the parser, via a function **nextsymb()**. This sets a global variable **symb** to the 'value' of the next token. These values are the same tokens as are used by the tree tags later on, all denoted by **S_** (but note that not all tokens require a tree node), or **S_NAME** for identifiers.

The lexer deals with line buffering, expanding tabs, etc. Indentation is dealt with by setting global variables to the current line's indentation, etc. These are interrogated by the parser when required. This approach was taken rather than using 'indent' and 'outdent' tokens because 'inline' **VALOF**s complicate the latter method. The lexer does *not* expand **#INCLUDE** directives transparently; these are noticed by the parser, which then tells the lexer to intepret the new file. Similarly for **#USE**, **#PRAGMA** directives, etc.

At startup, the lexer pumps a specially encoded set of strings into the parser, which represent the predefined routine headers. This makes it easy to add predefines when required.

The lexer contains a 'name table'. This is used to turn all names and strings into pointers, so that string equality can be tested by comparing pointers for equality. This name table has no notion of scoping, so multiple variables called **x**, say, will all map to the same name table entry.

5.2 Parser

The parser is simply a recursive descent parser. The top level view of an occam program is a list of occam declarations, but with no process at the end. Eg. it looks something like this:

```
PROC p ()
  ...  body of procedure
:
PROC q ()
  ...  body of procedure
:
```

The parser initially parses a list of declarations, allowing any style of declarations. This is then checked to ensure that no variables are declared at this outermost level.

The parser consists of many routines which parse portions of the language. These correspond (more-or-less) to the syntactic categories of occam, complicated by some of the more difficult aspects of occam, such as the ablility to encounter a declaration followed by a `VALOF` inside an expression.

Each routine for parsing a production normally creates and returns a sub-tree, or returns NULL if there was an error. These sub-trees are combined by the higher level routines to create a full parse tree. This contains all the original program information, (ie you can re-create the original source code), except that comments have been stripped out.

Every construct which declares a name also creates a `namenode`, which is essentially the symbol table entry. These refer back to the name table for the name. The declaration contains a pointer to the `namenode`, and vice-versa. Variable names, etc., appearing elsewhere in the tree (eg. in expressions) are indicated by a pointer to their name table entry. The format of the name table entry is sufficiently like a normal tree node that later phases can distinguish them by examining their tag.

5.3 Type checker

The type checker's first role is to resolve name scoping. It walks the tree 'top-down', and at every name declaration, it finds the corresponding symbol table entry (`namenode`) from one of the fields of the declaration node. The symbol table entries are pushed onto a scope stack. When a name table entry is encountered later in the tree walk, it must be resolved to the correctly scoped symbol. This is done by searching through the scope stack to find the first symbol with the same name. The name table entry in the tree is then replaced by the symbol table entry. (This actually occurs by overwriting a pointer). As symbols go out of scope, the `namenode`s are popped off the scope stack (but they remain in existence, now being only accessable via the tree).

After resolving names, on the 'bottom-up' pass the type checker uses the symbol table information for each name to resolve the types of sub-expressions and to check the types where required. If sub-expressions are constant, they are folded on this bottom-up pass. Type information is inserted into operator nodes. Many other occam rules are checked here, such as requiring constants in certain places, disallowing `ALT`s inside functions, etc.

5.4 Usage and alias checker

This phase begins by walking the tree and attaching a list of free variables to each procedure and function. This list indicates whether each name is read from, written to, input on, or output on. This information is used by both the alias and usage checkers. There are also a couple of checks performed here (eg, that channel parameters and free channels may only be used in a single direction.) This section is not optional.

The alias and usage checkers are optional. The alias checking is performed first; if this is successful usage checking can be applied.

Alias checking walks the tree and looks at abbreviations, and parameter lists, to check that they follow the occam rules that any item of data may only be referred to by a single name in any particular scope. Where this cannot be checked at compile time, a special node type **`overlapcheck`** is added to the tree containing the information which is needed to create the run-time check.

Usage checking involves detecting whether variables and channels are correctly used in parallel constructs. (This includes multiple assignment). Each procedure is checked separately, using free variable information to check procedure instances.

5.5 Tree transformer

This section again walks the tree. It expands procedures and functions which were marked as **`INLINE`** by the programmer. After each function expansion, it again constant folds the expression.

This phase also inserts subscript checks into those array accesses which cannot be checked at compile time. (This includes array segments too). It also expands the **`overlapcheck`**s inserted by the alias checker into an expression which performs the check when evaluated.

Finally, each array access or segment access is processed to make it easier to code generate from. Constant subscripts are folded, so that the resulting array access is given by an array base, plus a constant offset, plus a variable expression.

5.6 Mapper

The mapper performs two main roles. Its major role is to map variables to workspace locations (this is directly analogous to register allocation). To do this, it has to know which temporaries are required. Hence it needs to perform most of the code generation, to work out when temporaries are required. It also turns un-implemented arithmetic operations into library calls, and turns i/o into library calls where required.

For each procedure in turn, in a bottom up order, the mapper walks the code for that routine. When it encounter a variable declaration (or abbreviation, etc), it adds that variable to the list to be allocated. As it walks the tree, it keeps a usage count for each variable, based on an estimated loop count, etc. This approximates the dynamic usage count of each variable.

Where expressions are too complicated to fit into the transputer's register stack, the mapper introduces temporary variables. It also reserves workspace locations which are required for procedure and function calls, as well as augmenting their parameter lists with hidden parameters for static links, vectorspace pointers, and hidden array dimensions. Temporaries are introduced and killed explicitly by the mapper; hence they do not need to follow occam scoping rules (for example, they can exist for 'half' of an expression). Temporaries which are no longer live may be re-used, to reduce the number of different variables.

As it completes each procedure body, it allocates workspace slots. The variables are sorted into order according to their estimated usage. The 'reserved' locations are marked first, then the variables are taken in turn and mapped onto the lowest workspace location which does not clash with any other variables which are in scope at the same time. (This is the only form of liveness analysis which is performed). After the first 16 workspace locations have been mapped, the remaining variables are re-sorted into scope order, since this is more space efficient, and the most used variables will by now have been allocated to all the offsets which require no prefixing. The total required workspace size is recorded as part of the symbol table entry for that routine.

Sub-processes of **PAR** and replicated **PAR** are mapped separately, and the mapper introduces a new node into the tree, (a **spacenode**), which records the space requirements for later. Replicated **PAR**s are considered to be a new lexical level, and all their non-local accesses are performed via a static link.

5.7 Debug information

Having mapped all variables, the symbolic information is now known. The debug phase walks the tree again, top-down, and writes to the object file what is, in effect, a copy of the interesting parts of the code. This includes all variable's details, plus workspace requirements for routines and **PAR** bodies. Note that the mapping of code addresses to source lines cannot be performed yet; this is done after code crunching.

5.8 Code generation

The code generator phase walks the tree in a bottom up order, and generates code for each routine in the same order that their definitions are completed. It also outputs constant arrays which are encountered, again in the same order. Since all workspace offsets have been chosen, it can generate directly to transputer instructions, including operands for primary instructions. It marks labels and jumps to be resolved by the code cruncher. It also inserts markers to indicate the relationship between code address and source code offsets for the debug information.

It does not go through a textual assembly phase.

5.9 Code crunching

The code cruncher resolves jumps for a single procedure at a time, adjusting the prefixing instructions to optimise code size. (This also resolves the debug information). Because of the order in which the routines are generated, at the end of each routine there can be no unresolved references. Therefore the code is then packed up into the code buffer.

When all routines have been generated, these packed routines are written out to the code file, in reverse order. This ensures that all procedure calls are *forward*, and therefore do not need to use nfix instructions. Thus the code in the object file contains routines in the reverse order in which their definitions are completed.

5.10 Tree interface routines

The tree interface routines provide a mechanism for abstracting away from the actual tree implementation. The tree structure fields are never manipulated directly. A set of macros are provided which provide a cleaner interface to the tree, both for setting and reading tree fields.

There are also a number of routines which exist for tree manipulation. There are constructor functions for each node type, but no destructor functions because the memory is never recovered. There are tree walking routines which apply a function to every node of the tree. There are variants of these which do not walk inside expressions. There is also a routine which prints out a tree; this is useful for debugging (see also the command line switches).

6 Tree structure

Each node of the tree contains a tag, denoting the exact type of node, plus location information, indicating which file and line this node was generated by. Tree tags all begin with **S_**.

The tags do not each have a different node struture; they are grouped into various node types according to structure and use. For example, tags **S_ASSIGN** and **S_CASE_INPUT** are both **actionnode**s. The routine **int nodetypeoftag (int tag)** can be used to determine node type given a tag; the macro **int TagOf(treenode *tptr)** can be used to determine a treenode's tag.

The tree is *never* manipulated directly; instead macros are used which provide a 'virtual tree interface'. These macros are found in **vti.h**. For each node type there are a set of macros to access the fields. Each macro which reads a field has a name ending in **...Of**; for example **TagOf** is a macro which will return the tag of any node. The macros for setting fields all begin with **Set...**, for example **SetTag**. There are also macros which return the address of a field, which are formed by replacing the **...Of** with **...Addr**.

Macros valid on all treenodes are:

Macro	Description
int TagOf	Tag value
SOURCEPOSN LocnOf	File and line number information

6.1 actionnode

actionnodes denote simple actions with 'two parts' such as: Assignment (both simple and multiple), **CASE** statements (second part is the list of selections), **CASE input** (ditto), Delayed input, input, output.

The possible tag values for an **actionnode** are:

Tag name	Description
S_ASS	Simple or Multiple Assignment
S_CASE	**CASE** statement
S_CASE_INPUT	**CASE** input
S_DELAYED_INPUT	timer input **AFTER** time
S_INPUT	channel or timer input
S_OUTPUT	channel output

Macros valid on an **actionnode** are:

Macro	Description
treenode *LHSOf	'Left hand side' of process
treenode *RHSOf	'Right hand side' of process

6.2 altnode

altnodes denote branches of an **ALT**. They include the boolean guard expression, together with the input to be performed, and the process to execute. Specifications which precede the input are moved (by the mapper), to precede the process rather than the input, if they are not required for the input itself.

The only possible tag value for an **altnode** is **S_ALTERNATIVE**.

Macros valid on an **altnode** are:

Macro	Description
treenode *AltGuardOf	Boolean expression for guard
treenode *AltInputOf	Input process
treenode *AltBodyOf	Body of the alternative
treenode *AltChanExpOf	Copy of the channel expression from the input
int AltLabelOf	Label number; used in backend

6.3 arraynode

arraynodes are used to denote an array type. They hold a size expression, and a sub type. If the size is constant, the value is constant folded into another field; this is -1 if unknown (eg formal parameters of open array type). Multiple dimensioned arrays are formed as arrays of arrays.

The only possible tag value for an **arraynode** is **S_ARRAY**.

Macros valid on an **arraynode** are:

Macro	Description
treenode *ARDimLengthOf	Expression for this dimension's size
treenode *ARTypeOf	Sub-type of the array
int ARDimOf	Constant folded dimension size

6.4 arraysubnode

arraysubnodes are used to denote an array subscription. They hold fields indicating the base, and the subscript. They are processed in the tree transformation phase to combine 'nested' subscriptions, and to constant fold multiple constant subscripts.

The possible tag values for an **arraysubnode** are:

Tag name	Description
S_ARRAYITEM	Subscription after processing for the backend
S_ARRAYSUB	Subscription
S_RECORDSUB	Configurer - subfield (attribute of a **NODE**)

Macros valid on an **arraysubnode** are:

Macro	Description
treenode *ASBaseOf	Base of the subscription
treenode *ASIndexOf	Index expression
treenode *ASExpOf	**S_ARRAYITEM** only: variable expression
treenode *ASLengthOf	**S_ARRAYITEM** only: length expression, if not constant
int ASExpOf	**S_ARRAYITEM** only: constant folded offset

When the tag is **S_RECORDSUB**, the **ASIndex** field will always consist of a **namenode** with tag **N_FIELD**.

6.5 channode

channodes are used to denote **CHAN** or **PORT** types. They simply hold the protocol for that type.

The only possible tag values for a **channode** are **S_CHAN** and **S_PORT**.

Macros valid on a **channode** are:

Macro	Description
treenode *ProtocolOf	Protocol of the channel or port

6.6 cnode

cnodes are 'constructor' processes, such as **ALT** etc. They simply contain a list of sub-processes.

The possible tag values for a **cnode** are:

Tag name	Description
S_ALT	**ALT** construct
S_ASM	**ASM** construct (assembler insert)
S_DO	**DO** construct (configurer)
S_GUY	**GUY** construct (obsolete assembler insert)
S_IF	**IF** construct
S_PAR	**PAR** construct
S_PLACEDPAR	**PLACED PAR** construct (configurer)
S_PRIALT	**PRI ALT** construct
S_PRIPAR	**PRI PAR** construct
S_SEQ	**SEQ** construct

Macros valid on a **cnode** are:

Macro	Description
treenode *CBodyOf	List of sub-processes
treenode *CTempOf	Used by backend for **ALT**s

6.7 condnode

condnodes are conditional nodes, and are used for the guards of an **IF**. Since they hold a (boolean) expression, together with a process, they are also used for **WHILE** statements, and the selections of a **CASE** statement (where the expression is not necessarily boolean).

The possible tag values for a **condnode** are:

Tag name	Description
S_CHOICE	guard of an **IF** construct
S_SELECTION	selection of a **CASE** construct
S_WHILE	**WHILE** construct

Macros valid on a **condnode** are:

Macro	Description
`treenode *CondGuardOf`	(Boolean) guard expression
`treenode *CondBodyOf`	Body process

When the tag is **`S_SELECTION`**, **`CondGuardOf`** is either a list of constants, or an **`S_ELSE`** node.

6.8 confignode

`confignode`s are only used by the configurer. They hold details of configuration hardware and mapping properties.

The possible tag values for a **`confignode`** are:

Tag name	Description
`S_SET`	set type of a **`NODE`**
`S_CONNECT`	connect links
`S_MAP`	map logical name onto physical name

Macros valid on a **`confignode`** are:

Macro	When valid	Description
`treenode *STDevOf`	`S_SET`	**`NODE`** name
`treenode *STAttrNameOf`	`S_SET`	List of attribute names
`treenode *STAttrExpOf`	`S_SET`	List of attribute values
`treenode *ConnectFromEdgeOf`	`S_CONNECT`	Edge of connection
`treenode *ConnectToEdgeOf`	`S_CONNECT`	Edge of connection
`treenode *ConnectArcOf`	`S_CONNECT`	Arc of connection
`treenode *MapSourceOf`	`S_MAP`	Logical name
`treenode *MapDestOf`	`S_MAP`	Physical name
`treenode *MapPriOf`	`S_MAP`	Priority (**`NODE`**s only)

6.9 constexpnode

`constexpnode`s are used for scalar constants. They hold the value (up to 64 bits) plus a representation of how the constant was generated, so that the original constant could be re-generated.

The only possible tag value for a **`constexpnode`** is **`S_CONSTEXP`**.

Macros valid on a **`constexpnode`** are:

Macro	Description
`treenode *CExpOf`	Original expression tree
`BIT32 HiValOf`	Upper 32 bits of value
`BIT32 LoValOf`	Lower 32 bits of value
`treenode *CENextOf`	Used by backend for constant tables
`INT32 CEOffsetOf`	Used by backend for constant tables

6.10 consttablenode

consttablenodes are used for array constants. They hold a pointer to a name table entry which holds the data, plus a representation of the original expression. Strings are also represented using **consttablenode**s.

The only possible tag values for a **consttablenode** are **S_CONSTCONSTRUCTOR** and **S_STRING**.

Macros valid on a **consttablenode** are:

Macro	Description
wordnode *CTValOf	Name table entry containing byte string
treenode *CTExpOf	Original expression tree
treenode *CTNextOf	Used by backend for constant tables
int CTLabelOf	Used by backend for constant tables

6.11 declnode

declnodes denote any occam declaration. These include variable declarations, abbreviations, procedure declarations, etc. (In fact, anything which ends in a colon). The fields of the node contain the name, (actually held as a pointer to the symbol table entry (**namenode**)), (or a list of names in the case of a multiple variable declaration), the 'value' of the name (eg the right hand side of an abbreviation, or a procedure body), and the trailing process (ie the process which follows the colon). By a quirk of history, the trailing process is denoted as the 'body', which can cause confusion! The 'type' of the name being declared is held in the symbol table entry. This includes formal parameter lists for procedures and functions.

The possible tag values for a **declnode** are:

Tag name	Description
S_ABBR	[specifier] name **IS** element **:**
S_CONFIG	Configurer **CONFIG** construct
S_DECL	Variable declaration
S_LABELDEF	Assembler label declaration
S_LFUNCDEF	'Long' (multi-line) **FUNCTION** declaration
S_NETWORK	Configurer **NETWORK** construct
S_MAPPING	Configurer **MAPPING** construct
S_PLACE	**PLACE** name **AT** expression **:**
S_PLACEON	Configurer **PLACE** namelist **ON** physical **:**
S_PROCDEF	**PROC** declaration
S_RETYPE	specifier name **RETYPES** element **:**
S_SFUNCDEF	'Short' (single-line) **FUNCTION** declaration
S_SPROTDEF	Sequential **PROTOCOL** declaration
S_TPROTDEF	Tagged **PROTOCOL** declaration
S_VALABBR	**VAL** [specifier] name **IS** element **:**
S_VALRETYPE	**VAL** specifier name **RETYPES** element **:**
S_VSPLACE	**PLACE** name **IN VECSPACE :**
S_WSPLACE	**PLACE** name **IN WORKSPACE :**
	or **PLACE** name **AT WORKSPACE** expression **:**

Macros valid on a **declnode** are:

Macro	Description
treenode *DNameOf	**namenode** of declared name
treenode *DValOf	'Value' of name; eg. **PROC** body, or rhs of abbreviation
treenode *DBodyOf	Process which is the scope of the declaration

S_DECL nodes may declare a list of names. This can only occur where multiple *variables* are declared in a single declaration; there is no other way to declare multiple symbols in a single declaration. In this case, the **DNameOf** field contains a list of **namenode**s. Note that these **namenode**s will share a common **NTypeOf** type tree.

6.12 dopnode

dopnodes denote any dyadic operator. They are also used for counted array protocols, because they have two parts, also for **ASM** statements! They hold two expression fields, plus another denoting the type of the operator.

The possible tag values for a **dopnode** are:

Tag name	occam	Compiler Description
S_ADD	**+**	
S_AFTER	**AFTER**	
S_AND	**AND**	
S_BITAND	**/**	
S_BITOR	**\\/**	
S_CCNT1		Inserted for 'Check count from 1' (range checks)
S_COLON2	**::**	Used for counted array protocols
S_CSUB0		Inserted for 'Check subscript from 0' (range checks)
S_DIV	**/**	
S_EQ	**=**	
S_EVAL		Eval lhs then rhs; used for overlapchecks
S_FOR		**FROM x FOR y**; used for overlapchecks
S_GE	**>=**	
S_GR	**>**	
S_GUYCODE		Assembler instruction
S_GUYSTEP		Assembler instruction with microcode stepping
S_LE	**<=**	
S_LS	**<**	
S_LSHIFT	**<<**	
S_MINUS	**MINUS**	
S_MULT	*****	
S_NE	**<>**	
S_OR	**OR**	
S_OVERLAPCHECK		Overlapcheck node
S_PLUS	**PLUS**	
S_REM	\	
S_RSHIFT	**>>**	
S_SUBTRACT	**-**	
S_TIMES	**TIMES**	
S_XOR	**><**	

Macros valid on a **dopnode** are:

Macro	Description
`treenode *LeftOpOf`	Left operand tree
`treenode *RightOpOf`	Right operand tree
`int DOpTypeOf`	Type of operand (note that all are scalar operations)

If the node is a `S_GUYCODE` or `S_GUYSTEP`, then the 'left' operand is the name table entry containing the instruction string, and the 'right' operand is the instruction operand. The 'type' field is initialised by the type checker to the instruction opcode.

6.13 hiddenparamnode

`hiddenparamnode`s are used to denote extra parameters which are inserted into a `PROC` or `FUNCTION` parameter list by the compiler. These include the results of `FUNCTION`s which do not fit onto the integer stack, and parameters passed to 'fill in' open array dimensions. They are all inserted in the backend of the compiler.

The possible tag values for a `hiddenparamnode` are:

Tag name	Description
`S_FNACTUALRESULT`	Actual result parameter passed by reference
`S_FNFORMALRESULT`	Formal result parameter passed by reference
`S_HIDDEN_PARAM`	Open array dimension
`S_PARAM_VSP`	Hidden parameter for procedure vectorspace pointer

Macros valid on a `hiddenparamnode` are:

Macro	Description
`treenode *HExpOf`	Expression or type to which it applies
`BIT32 DimensionOf`	Dimension which hidden param represents
	or Vectorspace offset for an actual parameter `S_PARAM_VSP`

Important note: Both `S_FNACTUALRESULT` and `S_FNFORMALRESULT` nodes break the tree structure; ie their `HExpOf` field is actually a pointer to another part of the tree. Care must be taken here!

6.14 instancenode

`instancenode`s are used to denote procedure or function instances. They include a list of actual parameters. The list is augmented by the mapper to include hidden parameters if necessary.

The only possible tag values for an `instancenode` are `S_FINSTANCE` and `S_PINSTANCE`.

Macros valid on a `instancenode` are:

Macro	Description
`treenode *INameOf`	`namenode` of instanced routine
`treenode *IParamListOf`	Actual parameter list
`int ILoadSeqOf`	Used by backend

6.15 leafnode

leafnodes are used to denote any construct which needs to hold no extra information; examples are the types **INT**, **INT16**, etc, also the processes **SKIP**, **STOP**.

The possible tag values for a **leafnode** are:

Tag name	Description
S_ANY	Protocol for **CHAN OF ANY**
S_ARC	Configurer **ARC**
S_BOOL	
S_BYTE	
S_DUMMYEXP	Dummy expression for compiler-generated **S_CONSTEXP**
S_EDGE	Configurer **EDGE**
S_END	Marker for end of outermost list of declarations
S_ELSE	
S_FALSE	
S_NODE	Configurer **NODE**
S_INT	
S_INT16	
S_INT32	
S_INT64	
S_LABEL	Type of an assembler label
S_PARAM_STATICLINK	Hidden parameter for procedure static link
S_REAL32	
S_REAL64	
S_SKIP	
S_STOP	
S_TIMER	
S_TRUE	
S_UNDECLARED	Used while type checking in case of errors

There are no other access macros for a **leafnode**.

Important note: Currently all dummy expressions are simply pointers to a globally used **S_DUMMYEXP leafnode**. Hence they break the tree structure.

6.16 listnode

listnodes are used for a generic list structure. They consist of a **ThisItem** field, and a **NextItem** field. They are also used for a special case: the type of a function has two lists; one containing a list of return types, the other containing a list of formal parameters. The two lists are separated by a special listnode.

The only possible tag values for a **listnode** are **S_LIST** and **S_FNTYPE**.

Macros valid on a **listnode** are:

Macro	Description
treenode *ThisItem	This item on the list
treenode *NextItem	Next item on the list
treenode *FnTypeListOf	**FUNCTION** return type list
treenode *FnParamsOf	**FUNCTION** formal parameter list

The macro **int EmptyList(treenode *)** can be used to determine when a list is empty.

6.17 litnode

litnodes are used to denote constant literals from the source code. They simply hold an entry in the name table which contains the string denoting that literal.

The possible tag values for a **litnode** are:

Tag name	Example	Note
S_BYTELIT	**27(BYTE)**	
S_INTLIT	**27(INT)**	
S_INT16LIT	**27(INT16)**	
S_INT32LIT	**27(INT32)**	
S_INT64LIT	**27(INT64)**	
S_REAL32LIT	**27.0(REAL32)**	
S_REAL64LIT	**27.0(REAL64)**	
S_UBYTELIT	**'a'**	converted to **S_BYTELIT**
S_UINTLIT	**27**	converted to **S_INTLIT**
S_UREALLIT	**27.0**	converted to **S_REALnnLIT**

Macros valid on a **litnode** are:

Macro	Description
wordnode *StringPtrOf	Name table entry containing literal string

These nodes are all constant folded into **S_CONSTEXP** nodes by the type checking phase.

6.18 mopnode

mopnodes are used to denote monadic operators. They have a field describing the expression, and another denoting the type of the operator. Table constructors are also denoted by a **mopnode**, where the expression is actually a list of expressions.

The possible tag values for a **mopnode** are:

Tag name	occam	Compiler Description
`S_ADDRESSOF`		Used in Assembler, and by backend
`S_BITNOT`	`~`	Bitwise negation
`S_CONSTRUCTOR`		Eg. `[x + y, z, 27]`
`S_ELSIZE`		Equvalent to `SIZE`, inserted by backend
`S_EXACT`	type conversion	Eg. `INT16 int`
`S_MOSTPOS`	`MOSTPOS`	`MOSTPOS INT`, etc
`S_MOSTNEG`	`MOSTNEG`	`MOSTNEG INT`, etc
`S_NEG`	`-`	Unary negation
`S_NOT`	`NOT`	Boolean negation
`S_ROUND`	type conversion	Eg. `REAL32 ROUND int`
`S_SEGSTART`		Start of segment, inserted by backend
`S_SIZE`	`SIZE`	
`S_TRUNC`	type conversion	Eg. `REAL32 TRUNC int`
`S_UMINUS`	`MINUS`	Unary `MINUS`

Macros valid on a `mopnode` are:

Macro	Description
`treenode *OpOf`	Expression tree
`int MOpTypeOf`	Type of operand (note that all are scalar operations)

Important note: Both `S_ELSIZE` and `S_SEGSTART` nodes break the tree structure; ie their `OpOf` field is actually a pointer to another part of the tree. Care must be taken here!

6.19 namenode

`namenode`s constitute the symbol table and are described in another section.

6.20 processornode

`processornode`s are only used by the configurer.

The only possible tag value for a `processornode` is `S_PROCESSOR`.

Macros valid on a `processornode` are:

Macro	Description
`treenode *ProcessorExpOf`	
`treenode *ProcessorTypeOf`	
`treenode *ProcessorBodyOf`	
`treenode *ProcessorScopeOf`	

6.21 replcnode

`replcnode`s are used to denote replicated constructors, such as `PAR i = ...` etc. They have fields to denote the symbol table entry of the replicator variable, and the start and length expressions, plus the body of the replicator.

The possible tag values for a **replcnode** are:

Tag name	Description
S_PLACEDREPLPAR	**PLACED PAR i = ...** (configurer)
S_PRIREPLALT	**PRI ALT i = ...**
S_PRIREPLPAR	**PRI PAR i = ...**
S_REPLALT	**ALT i = ...**
S_REPLDO	**DO i = ...** (configurer)
S_REPLIF	**IF i = ...**
S_REPLPAR	**PAR i = ...**
S_REPLSEQ	**SEQ i = ...**

Macros valid on a **replcnode** are:

Macro	Description
treenode *ReplCNameOf	**namenode** of replicator variable
treenode *ReplCStartExpOf	Start expression tree
treenode *ReplCLengthExpOf	Length expression tree
treenode *ReplCBodyOf	Replicated process body
treenode *ReplCTempOf	Used by backend for **ALT**

6.22 segmentnode

segmentnodes are used to denote segments of arrays. They hold the base of the segment, plus a start and length expression. In the tree transformation phase these are manipulated in a similar way to **arraysubnode**s.

The only possible tag values for a **segmentnode** are **S_SEGMENT** and **S_SEGMENTITEM**. The latter indicates that the node has been processed by the tree transformation phase.

Macros valid on a **segmentnode** are:

Macro	Description
treenode *SNameOf	Expression denoting base of segment
treenode *SStartExpOf	Start expression tree
treenode *SLengthExpOf	Length expression tree
treenode *SCheckExpOf	**S_SEGMENTITEM** only: range checking
treenode *SSubscriptExpOf	**S_SEGMENTITEM** only: variable expression
treenode *SLengthOf	**S_SEGMENTITEM** only: length expression, if not constant
int SOffsetOf	**S_SEGMENTITEM** only: constant folded offset

6.23 spacenode

spacenodes are inserted by the mapper into **PAR** and replicated **PAR** constructs. They hold details of the workspace requirements for each branch of the **PAR**.

The only possible tag value for a **spacenode** is **S_SPACENODE**.

Macros valid on a **spacenode** are:

Macro	Description
treenode *SpBodyOf	Body of **PAR**
INT32 SpMaxwspOf	‘Above workspace’ space required
INT32 SpDatasizeOf	Total space including ‘below workspace’
INT32 SpVSUsageOf	Vectorspace requirements
INT32 SpNestedVSOf	Vectorspace requirements for nested calls
BIT32 SpCPOffsetOf	Constant table offset in replicated **PAR**
treenode *SpNamechainOf	Names declared in body (?)

6.24 valofnode

valofnodes are used to denote a **VALOF**, which has a process and a result list. The body of a function will always be a **valofnode**.

The only possible tag value for a **valofnode** is **S_VALOF**.

Macros valid on a **valofnode** are:

Macro	Description
treenode *VLBodyOf	Body of **VALOF** process
treenode *VLResultListOf	**RESULT** list

6.25 variantnode

variantnodes are used to denote a variant of a case input process. They hold a list of input items (beginning with a protocol tag), and a process to be performed.

The only possible tag value for a **variantnode** is **S_VARIANT**.

Macros valid on a **variantnode** are:

Macro	Description
treenode *VRBodyOf	Body of variant
treenode *VRTaggedListOf	List of input expressions

6.26 wordnode

See another section about the name table.

7 Name table

The name table is a data structure used for storing names, keywords, and various other strings, without duplication. When adding a name to the name table, if it is already there, a pointer to the original is returned. If not, a new name table entry is created, and its pointer is returned. This means that pointers may be tested for equality to determine string equality.

Each name in the table also has a tag attributed with it. Keywords are initialised so that their strings have the correct tag associated. This is used by the lexer as a quick way of looking up keywords. Any new names are given the tag **S_NAME**. Miscellaneous strings are also stored in the name table; in this case the associated tag is ignored.

Name table entries are represented as **wordnode**s. The have the same tag layout as tree nodes so that they can be distinguished by using **TagOf()**. **wordnode**s do not have 'location' information, as they are not associated with a particular filename and line number. **wordnode**s are not found after the type checking phase.

The only possible tag values for a **wordnode** (when found in the tree) are **S_NAME** and **S_ASMNAME**. The latter is used by specific names which are only permitted inside **ASM** constructs.

Macros valid on a **wordnode** are:

Macro	Description
int WTagOf	Tag (can also use **TagOf()**
char *WNameOf	Pointer to character string
int WLengthOf	Length of character string
wordnode *WNextOf	Used internally by name table

8 Symbol table

The symbol table never exists in its own right, but is distributed into lots of **namenode**s which hang off the tree. Each **declnode** (declaration node) of the tree holds a pointer to the **namenode** of the corresponding declaration. These **namenode**s all have tags beginning **N_**, but otherwise are of the same form as other tree nodes. Temporaries are also **namenode**s. Their tags begin with **T_**.

The possible tag values for a **namenode** are:

Tag name	Description
N_ABBR	Abbreviation
N_CONFIG	Configurer - named **CONFIG** construct
N_DECL	Variable declaration
N_FIELD	Configurer - field (attribute) of a record
N_INLINEFUNCDEF	**INLINE FUNCTION**
N_INLINEPROCDEF	**INLINE PROC**
N_LABELDEF	**ASM** or **GUY** label
N_LFUNCDEF	‘Long’ **FUNCTION**
N_LIBFUNCDEF	Library **FUNCTION**
N_LIBPROCDEF	Library **PROC**
N_MAPPING	Configurer - named **MAPPING** construct
N_NETWORK	Configurer - named **NETWORK** construct
N_PARAM	Formal parameter
N_PREDEFFUNCTION	Predefined **FUNCTION**
N_PREDEFPROC	Predefined **PROC**
N_PROCDEF	Procedure
N_REPL	Replicator variable
N_RETYPE	**RETYPE**
N_SCFUNCDEF	**SC FUNCTION** (obsolete)
N_SCPROCDEF	**SC PROC** (obsolete)
N_SFUNCDEF	‘Short’ **FUNCTION**
N_SPROTDEF	Sequential **PROTOCOL**
N_STDLIBFUNCDEF	‘Standard library’ **FUNCTION**
N_STDLIBPROCDEF	‘Standard library’ **PROC**
N_TAGDEF	**PROTOCOL** tag
N_TPROTDEF	Tagged **PROTOCOL**
N_VALABBR	**VAL** abbreviation
N_VALPARAM	**VAL** formal parameter
N_VALRETYPE	**VAL RETYPE**
T_PREEVALTEMP	Temporary which has already been evaluated
T_REGTEMP	Register temporary - used for **FUNCTION** results
T_RESERVEDWS	Reserved low workspace - used for **ALT** and param slots
T_TEMP	Temporary

8.1 Generic symbol table fields

All symbols have the following subsidiary fields, with differing levels of relevance to different types of symbols: Note that each of these macros actually takes a **wordnode**; but since all node types are merged into a union **treenode**, the ‘prototypes’ indicate this. These access macros are held in **nameshdr.h**.

- **`wordnode *NNameOf(treenode *)`**
  This returns a name table entry for this symbol.

- **`treenode *NTypeOf(treenode *)`**
  This returns the 'type tree' for the symbol. For simple variables this might simply be a **`leafnode`** containing **`S_INT`** etc. If an array, it would be an **`S_ARRAY`** node. For procedures, this will be a list of symbols corresponding to the formal parameters. For functions, this will be a **`S_FNTYPE`** node which holds two fields: **`FnTypeListOf()`** which is a list of return types, and **`FnParamsOf()`** which is the formal parameter list.

  Procedure and function formal parameter lists can be accessed by the routines **`treenode *NParamListOf(treenode *nptr)`** and **`void SetNParamList(treenode *nptr, treenode *list)`**. These determine whether the routine is a function or procedure, and access the formal parameter list, either via the **`S_FNTYPE`** node, or directly.

- **`treenode *NDeclOf(treenode *)`**
  This returns a pointer back to the **`declnode`** in the tree which defines this symbol. This allows you to determine the 'value' of the symbol given the **`namenode`**.

- **`int NLexLevelOf(treenode *)`**
  This returns the lexical level of the symbol's definition. Symbols defined at the outermost compilation level have lexical level 0. Each nested procedure and function body, and each replicated **`PAR`**, introduces a new lexical level.

- **`int NNestedPriParOf(treenode *)`**
  This is set to **`TRUE`** if the procedure contains a **`PRI PAR`**. Used in the type checker.

- **`int NUsedOf(treenode *)`**
  This is set to **`TRUE`** if the symbol is used. This is used to warn about unused variables in the type checker.

- **`int NScopeOf(treenode *)`**
  This returns an integer indicating how many other names were on the scope stack when the symbol was declared. Thus any symbol **`x`** nested inside a symbol **`y`** will have **`NScopeOf(x) > NScopeOf(y)`**.

- **`treenode *NChecker(treenode *)`**
  This is a 'spare' field which is used by the alias and usage checker.

8.2 Variables

Data items such as variables, channels, etc, hold other more specific data:

- **`BIT32 NVOffsetOf(treenode *)`**
  This returns the workspace location of this variable. This is initialised by the mapper.

- **`int NModeOf(treenode *)`**
  This returns a value indicating the 'memory access' type. The possible values are:

Mode	Description
NM_DEFAULT	Not yet decided.
NM_WORKSPACE	Item is placed in current stack frame.
NM_VECSPACE	Current stack frame contains a pointer to vectorspace.
NM_PLACED	**PLACED** at an absolute address.
NM_POINTER	Accessed via a pointer (eg. formal param or abbreviation).
NM_WSPLACED	**PLACED** at a specific workspace address.

The 'memo' function **int isinvectorspace()** can be applied to a variable's **namenode**. If the mode is still **NM_DEFAULT**, it will decide and convert it to either **NM_WORKSPACE** or **NM_VECSPACE**.

Variables are placed into vectorspace if vectorspace is enabled and 1) They are an array; and 2) either they use more than 8 bytes, or they are of a type which is smaller than a word.

- **treenode *NVNextOf(treenode *)**
  This is used to associate many variables together; for example if we have the abbreviation **x IS y :**, then **x** and **y** are chained together and will be allocated the same workspace slot.

- **BIT32 NVVarNumOf(treenode *)**
  This is used by the mapper to give temporaries an identifying number. Not used for other symbols (?).

- **treenode *NVNextTemp(treenode *)**
  This is used by the mapper to chain temporaries together (while they're not live?). Not used for other symbols (?).

- **treenode *NVAllocNextOf(treenode *)**
  This is used by the mapper to chain together a list of all variables which have already been allocated; these are then checked for clashes with each successive variable.

- **BIT32 NVVSOffsetOf(treenode *)**
  Only applies to simple variables (ie. those with tag **N_DECL**). This returns the vectorspace offset for this variable if it is mapped into vectorspace.

- **BIT32 NVUseCountOf(treenode *)**
  This returns the estimated dynamic usage count for this variable. Calculated by the mapper.

- **int NReplKnownOf(treenode *)**
  Only applies to replicator variables (ie. those with tag **N_REPL**). This is used by the Usage checker when expanding out replicator variables. If this is **TRUE**, then **NReplValueOf()** is valid; see below. This field is not available in the backend.

- **BIT32 NReplValueOf(treenode *)**
  Only applies to replicator variables (ie. those with tag **N_REPL**). This is used by the Usage checker when expanding out replicator variables. If **NReplKnownOf()** is **TRUE**, then this is valid; see above. Returns the 'current' value of the replicator. This field is not available in the backend.

- **int NChanMarkOf(treenode *)**
  This field only applies to scalar channels (**TagOf(NTypeOf(n)) == S_CHAN**). If this returns **TRUE**, then the channel is used from assembler, and certain optimisations may not be performed. Initialised by the mapper.

- **BIT32 NVRSizeOf(treenode *)**
  This field only applies when **TagOf(n) == T_RESERVEDWS**. This returns the number of words

which are reserved.

8.3 Protocol tags

- **`BIT32 NTValueOf(treenode *)`**
  This returns the value of that protocol tag.

8.4 Procedure and Function symbols

Procedures and Functions hold other more specific data:

- **`int NPLabelOf(treenode *)`**
  This returns the label number of the entrypiont of the routine. It is initialised as the code for that routine is generated (which will be before any references).

- **`BIT32 NPMaxwspOf(treenode *)`**
  This returns the maximum workspace requirement for this routine (in words).

- **`BIT32 NPDatasizeOf(treenode *)`**
  This returns the 'below workspace' requirements for that routine; ie. the number of workspace slots which must be reserved for nested procedures and descheduling, etc.

- **`BIT32 NPVSUsageOf(treenode *)`**
  This returns the number of vectorspace words required for this routine. If it is zero, no vectorspace pointer parameter is used.

- **`NPParamsOf(treenode *)`**
  This returns the number of parameters required by this routine.

- **`NPSLUsageOf(treenode *)`**
  This returns the highest (lowest?) lexical level of non-local variables which are used by this routine.

'Normal' routines, (tag values **`N_PROCDEF`**, **`N_LFUNCDEF`**, or **`N_SFUNCDEF`**), ie. those actually being compiled, also hold the following:

- **`treenode *NPConstTablesOf(treenode *)`**
  This returns a pointer to a list of `constexpnode`s and `consttablenode`s which are accessed by this routine. It is used to generate the constant table.

- **`BIT32 NPCPOffsetOf(treenode *)`**
  This returns the workspace offset of the constant table pointer; this is initialised by the mapper.

Separately compiled routines, (tag values **`N_LIBPROCDEF`**, **`N_LIBFUNCDEF`**, **`N_SCPROCDEF`**, **`N_SCFUNCDEF`**, **`N_STDLIBPROCDEF`**, **`N_STDLIBFUNCDEF`**), which have been **`#USE`**d from a library, also hold the following:

- **`void *NLExternalOf(treenode *)`**
  This returns a pointer to another structure, which is used within `desc1.c` to access data about the origin file, etc.

- **`NLEntryNextOf(treenode *)`**
  This is used to chain together all the external routines which are actually used in this compilation.

- **`NLEntryOffsetOf(treenode *)`**
  This returns the address of the code area for the linker to patch with the real call to the routine.

Predefined routines, (tag values **`N_PREDEFPROC`** or **`N_PREDEFFUNCTON`**), also hold the following:

- **`int NModeOf(treenode *)`**
  This returns a number indicating which predefined routine. These values are found in **`predefhd.h`**.