# occam-2 language implementation manual

Conor O’Neill

SW-0044-4



APPROVED 21 September, 1990

# Contents


1 Introduction . . . 3
2 Change history . . . 3
2.1 Changes since SW0044-03. . . . 3
2.2 Changes since SW0044-02. . . . 3
2.3 Changes since issue of January 24, 1990. . . . 3
2.4 Changes since issue of January 4, 1990. . . . 4
3 Language differences . . . 4
3.1 Compiler keywords . . . 4
3.2 Syntax permitted at outermost level . . . 5
3.3 INLINE keyword . . . 5
3.4 String escape characters . . . 5
3.5 Vectorspace . . . 6
3.6 PLACE name AT WORKSPACE n . . . 6
3.7 RETYPING channels . . . 8
3.8 Channel constructors . . . 8
4 Implementation restrictions . . . 9
5 Implementation limits . . . 9
6 Implementation defined areas . . . 9
7 Usage and Alias checking . . . 11
7.1 Usage checking . . . 11
7.2 Alias checking . . . 11
7.3 Other checks . . . 12
8 Error behaviour . . . 12
9 Compiler directives . . . 13
9.1 #COMMENT directive . . . 13
9.2 #IMPORT directive . . . 13
9.3 #INCLUDE directive . . . 13
9.4 #OPTION directive . . . 13
9.5 #PRAGMA directive . . . 14
9.5.1 EXTERNAL pragma . . . 14
9.5.2 LINKAGE pragma . . . 15
9.5.3 TRANSLATE pragma . . . 15
9.6 #SC directive . . . 15
9.7 #USE directive . . . 16
10 Predefined routines . . . 17
10.1 Multiple length integer arithmetic functions . . . 17
10.2 Floating point functions . . . 18
10.3 Full IEEE arithmetic functions . . . 19
10.4 Transputer-specific predefined routines . . . 20
10.4.1 General purpose routines . . . 20
10.4.2 Block Move routines . . . 21
10.4.3 Cyclic redundancy checking . . . 22
10.4.4 Bit manipulation routines . . . 23
10.4.5 Floating point support routines . . . 23
10.4.6 Dynamic code loading . . . 24
11 Other Standard libraries . . . 25
11.1 Elementary function library . . . 25
11.2 Value and string conversion procedures . . . 25
12 Inline transputer code insertion . . . 26
12.1 GUY Construct . . . 26
12.2 ASM Construct . . . 27
12.2.1 ASM instructions . . . 28
12.2.2 Pseudo operations . . . 29
12.2.3 Special names . . . 29

12.3 Differences between ASM and GUY . . . . . 30
12.4 Sequential code insertion . . . . . 30
12.5 Full code insertion . . . . . 31
13 The end . . . . . 31

# 1 Introduction

This document describes the language which the occam 2 compiler (**oc**) compiles. This is basically the language as described in “occam 2 Reference Manual”, Prentice-Hall 1988 (ISBN 0-13-629312-3). It should be read in conjunction with the Reference Manual; any differences are as described below.

# 2 Change history

## 2.1 Changes since SW0044-03.

- Tabs inside character literals and strings are not expanded.
- No longer a limit of 2000 names in scope.
- **BYTE** and **WORD ASM** pseudo ops can take a list of args.
- No labels allowed inside **INLINE** routines.
- Improved detail on **RETYPED** channels.
- Inserted detail on **PLACED** channels.
- Channel constructors.
- No hard limits on size of compiler libraries.
- No restrictions on nested ALT levels.
- Added predefines **IEEE32REM** and **IEEE64REM**.

## 2.2 Changes since SW0044-02.

- Hard limit of 200 entrypoints in a library has been removed.
- **UNPACKSN** and **ROUNDSN** not supported on 16-bit processors.
- Better description of **PLACE name AT WORKSPACE n**.
- Constant arrays subscripted by replicators are not constant.

## 2.3 Changes since issue of January 24, 1990.

- Better description of **KERNEL.RUN**.
- Changed maximum number of nested files to 20.
- Better description of ‘complicated’ abbreviations.
- Better explanation of **INLINE**.
- Error behaviour of **SHIFTRIGHT** etc. (again).

- Error behaviour of **`BITREVNBITS`**. (again).

## 2.4 Changes since issue of January 4, 1990.

- Completely revamped **`ASM`** specification.
- Changed definition of **`LOAD.INPUT.CHANNEL.VECTOR`**, etc.
- Addition of LINKAGE pragma.
- Nesting of **`PRI PAR`** is always checked at compile time.
- EXTERNAL pragma doesn't default to 1 workspace slot.
- Named the order of return values from **`UNPACKSN`**.
- Added **`PLACE AT WORKSPACE`** *n*.
- Multiple assignment of fixed length arrays is OK.
- Tabs are every 8 stops.
- **`#SC`** no longer supported.
- **`'*L'`** as well as **`'*l'`**.
- Arrays are aligned to word boundary on definition.
- **`INT16 RETYPES`** aligned to half-word boundary on 32-bit processors.
- Error behaviour of **`SHIFTRIGHT`** etc.
- Error behaviour of **`BITREVNBITS`**.
- TRANSLATE pragma must precede its **`#USE`**.
- Any number of selections allowed in a **`CASE`**.

# 3 Language differences

This section describes the differences between the language as implemented, and that described in "occam 2 Reference Manual", Prentice-Hall 1988.

## 3.1 Compiler keywords

The additions to the list supplied in "occam 2 Reference Manual", Prentice-Hall 1988 are:

**`ASM GUY INLINE STEP VECSPACE WORKSPACE`**

These keywords are *reserved* and as such cannot be used by a programmer as variable names, etc. This is a full list of the keywords reserved to the compiler:

| | | | | | | |
|---|---|---|---|---|---|---|
| **AFTER** | **ALT** | **AND** | **ANY** | **ASM** | **AT** | **BITAND** |
| **BITNOT** | **BITOR** | **BOOL** | **BYTE** | **CASE** | **CHAN** | **ELSE** |
| **FALSE** | **FOR** | **FROM** | **FUNCTION** | **GUY** | **IF** | **IN** |
| **INLINE** | **INT** | **INT16** | **INT32** | **INT64** | **IS** | **MINUS** |
| **MOSTNEG** | **MOSTPOS** | **NOT** | **OF** | **OR** | **PAR** | **PLACE** |
| **PLACED** | **PLUS** | **PORT** | **PRI** | **PROC** | **PROCESSOR** | **PROTOCOL** |
| **REAL32** | **REAL64** | **REM** | **RESULT** | **RETYPES** | **ROUND** | **SEQ** |
| **SIZE** | **SKIP** | **STEP** | **STOP** | **TIMER** | **TIMES** | **TRUE** |
| **TRUNC** | **VAL** | **VALOF** | **VECSPACE** | **WHILE** | **WORKSPACE** | |

## 3.2 Syntax permitted at outermost level

The compiler places restrictions on the syntax which is permitted at the outermost level of a compilation unit; ie. not enclosed by any function or procedure.

- No variable declarations are permitted.
- The file must contain at least one **PROC** or **FUNCTION**; a null source file is illegal.
- No abbreviations containing function calls or **VALOF**s are allowed, even if they are actually constant. For example:

```
VAL x IS (VALOF
            SKIP
            RESULT 99) :  -- This is illegal
VAL m IS max (27, 52) :   -- so is this
```

## 3.3 INLINE keyword

You may add the **INLINE** keyword immediately before the **PROC** or **FUNCTION** keyword of any procedure or function declaration. This will cause its body to be expanded inline in any call, and the declaration will not be compiled as a normal routine. Note that the *declaration* is marked with the keyword, but the *call* is affected. This means that you cannot inline expand procedures and functions which have been declared by a **#USE** directive; to achieve that effect you may put the source of the routine in an include file, marked with the **INLINE** keyword, and include it with an **#INCLUDE** directive.

For example:

```
INT INLINE FUNCTION sum3 (VAL INT x, y, z) IS x + (y + z) :

INLINE PROC seterror ()
  error := TRUE
:
```

There is an implementation restriction that **ASM** labels may not be defined within an **INLINE PROC** or **FUNCTION**.

## 3.4 String escape characters

The compiler accepts the string escape characters as described in Appendix I of "occam 2 Reference Manual", Prentice-Hall 1988. The compiler also accepts **'*l'** or **'*L'** as the first character of a string literal. This is expanded to be the length of the string excluding the character itself. For example, **string1** and **string2** are identical:

```
VAL string1 is "*lFred" :
VAL string1 is "*#04Fred" :
```

The use of `'*l'` is illegal if the string (excluding the `'*l'`) is longer than 255 bytes, and will be reported as an error.

## 3.5 Vectorspace

The occam compiler allocates space for local scalar variables on a falling stack known as the *workspace*. By default, arrays larger than 8 bytes are allocated into another stack known as the *vectorspace*. This optimises use of the workspace, creating more compact and quicker code. It can also make better use of a transputer's on-chip RAM.

This can be overridden in two ways. Firstly, by using either a command line switch, or the **#OPTION** directive (see below), the default can be totally overridden, forcing all variables into the workspace. Secondly, the current 'default' may be overridden on an array-by-array basis by using extra allocations as follows:

```
[100]BYTE a :
PLACE a IN WORKSPACE :  -- forces a to reside in workspace

[100]BYTE b :
PLACE b IN VECSPACE :   -- forces b to reside in vectorspace
```

Only arrays may be placed in vectorspace; scalar variables must reside in workspace. Arrays smaller than 8 bytes may be explicitly placed in vectorspace.

It may be desirable to change the default vectorspace allocation for various reasons. Using vectorspace can actually slow down execution, since an extra parameter is passed to each subroutine which requires it. However, this cost is normally overwhelmed by the reduction in workspace size, and the associated compaction in the number of prefix instructions required to address local variables. In certain circumstances it may be useful to place a commonly used array into workspace, particularly if it is heavily used in array assignment (block moves). Alternatively it may be useful to place most arrays in workspace, but move any large arrays into vectorspace.

## 3.6 PLACE name AT WORKSPACE n

The compiler implements another allocation; **PLACE name AT WORKSPACE** *n*, where *n* is a constant integer. This is used to ensure that a variable is allocated a particular position within a procedure or function's workspace. This is normally required only if various specialised transputer instructions which require specific workspace placings are used within a code insert. For example, **POSTNORMSN** uses workspace location 0; **OUTBYTE**, **OUTWORD** and the disabling **ALT** instructions also use workspace location 0.

On a **T414**, the **POSTNORMSN** instruction can be used to pack a floating point number; it requires an exponent to be previously stored at workspace offset 0. See "Transputer Instruction Set - A compiler writer's guide", Prentice-Hall 1988 for this example.

```
REAL32 FUNCTION pack (VAL INT guard, frac, exp, sign)
  REAL32 result :
```

```
  VALOF
    INT temp :
    PLACE temp AT WORKSPACE 0 :
    SEQ
      temp := exp
      ASM
        LDAB guard, frac
        NORM
        POSTNORMSN
        ROUNDSN
        LDL sign
        OR
        ST result
    RESULT result
:
```

The compiler ensures that at least *n* words of workspace are allocated for that procedure or function, and that no other variables are placed at that address. The compiler will warn if a variable **PLACED AT WORKSPACE** *n* is in scope when its own workspace allocation requires to use that workspace location, or when another is **PLACED** at the same location.

## 3.7 RETYPING channels

Channels may now be **RETYPE**d.

Firstly this allows you to change the protocol on a channel, in order to pass it as a parameter to another routine, for example. This facility should be used with care.

```
PROTOCOL PROT32 IS INT32 :
PROC p (CHAN OF INT32 x)
  x ! 99(INT32)
:
PROC q1 (CHAN OF PROT32 y)
  SEQ
    p (y)                        -- this is illegal
    CHAN OF INT32 z RETYPES y :
    p (z)                        -- this is legal
:
```

Channels may also be **RETYPE**d to and from data items. The data items map onto a pointer to the channel word. This can be used to determine the address of the channel word, or to create an array of channels pointing at particular addresses:

*Note that this ability is extremely non-portable, and its effect may change even with different compiler command line options, (eg the* **ZW** *option), but these options are currently not user documented.*

```
CHAN OF protocol c :
VAL INT x RETYPES c :  -- Must be a VAL RETYPE
...   use x as the address of the channel word
```

Note that this can be achieved more portably by means of the **LOAD.INPUT.CHANNEL** predefines.

The following code demonstrates how to create a channel array whose channels point at arbitrary addresses.

```
[10]INT x :
SEQ
  ...  initialise elements of x to the addresses of the channel words
  [10]CHAN OF protocol c RETYPES x :
  ...  use channel array c
```

## 3.8 Channel constructors

Arrays of channels may be constructed out of a list of other channels. For example.

```
PROC p (CHAN OF protocol a, [2]CHAN OF protocol b)
  [3]CHAN OF protocol c IS [a, b[0], b[1]] :  -- channel constructor
  ALT i = 0 FOR SIZE c
    c[i] ? data
      ...
:
```

# 4 Implementation restrictions

- **FUNCTION**s may not return arrays, not even with fixed sizes.
- Multiple assignment of arrays of unknown size is not permitted.
- Replicated **PAR** count must be constant.
- There must be exactly 2 branches in a **PRI PAR**.
- Replicated **PRI PAR**s are not allowed.
- Table sizes must be known at compile time. For example:

```
PROC p ([]INT a, []INT b)
  VAL [][]INT x IS [a] :   -- this is illegal
  VAL   []INT y IS  b  :   -- this is   legal
  ...
:
```

- Constant arrays which are indexed by replicator variables are not considered to be constants for the purposes of compiler constant folding, even if the start and limit of the replicator are also constant. This restriction does not apply during usage checking.
- **ASM** labels are not permitted inside **INLINE** routines.

# 5 Implementation limits

- Maximum number of nested files is 20.
- Maximum source line length is 255 characters (including leading spaces).
- Maximum filename length is 128 characters.
- Maximum 256 tags allowed in **PROTOCOL**s.
- Maximum number of lexical levels is 254. (Nested **PROC**s and replicated **PAR**s).
- Maximum number of variables in a procedure or function is 512.

# 6 Implementation defined areas

See SW-0064 (“occam 2 Run time model”) for more details of the run-time layout of variables.

- Identifiers.
  There is no limit to the number of significant characters in identifiers, and the case of characters is significant.
- Values given to protocol tags.
  Protocol tags are given **BYTE** values consecutively from zero.

- Implementation of **CASE** statement.
  **CASE** statements are implemented as a combination of explicit tests, binary searches, and jump tables, depending on the relative density of the selection values. The choice has been made to optimise the general case where each selection is equally probable. The compiler does not make any use of the order of the selections as they are written in the source code.

- Implementation of **ALT** statement.
  No assumption can be made about the relative priority of the guards of an **ALT** statement; if priority is required, you must use a **PRI ALT**.

- Implementation of **PRI PAR**.
  The compiler implements two priorities; high and low. These match exactly the transputer's two priorities.

  The compiler does not permit a **PRI PAR** statement to be nested inside the high priority branch of another. This is checked at compile time, even across separately compiled units.

- **PLACED** variables.
  The address used in a **PLACE** allocation is converted to a transputer address by considering the address to be a word offset from **MOSTNEG INT**.

  For example, suppose we require to access a **BYTE** memory mapped peripheral located at machine address **#1234**, on a 32 bit processor.

  ```
  PORT OF BYTE peripheral :
  PLACE peripheral AT (#1234 >< (MOSTNEG INT)) >> 2 :
  peripheral ! 0(BYTE)
  ```

  Individual channels which are placed use the specified address as the channel word. Arrays of channels which are placed map the array of pointers onto the specified address.

  *Note that this is different from the behaviour of the 'old' occam compiler which was supplied with* **IMS D705B** *and equivalent products.*

  To create an array of channels **PLACED** on the transputer's links, you should use the following:

  ```
  CHAN OF protocol link0, link1, link2, link3 :
  PLACE link0 AT 0 :
  PLACE link1 AT 1 :
  PLACE link2 AT 2 :
  PLACE link3 AT 3 :
  [4]CHAN OF protocol out.links IS [link0, link1, link2, link3] :
  ```

- Alignment of data.
  The first element of an array is always aligned to a word boundary. (This obviously does not apply to segments of **BYTE** arrays, etc.)

- **RETYPES**.
  Values accessed through **RETYPES** must be aligned to the natural alignment for that datatype; **BYTE**s and **BOOL**s may be aligned to any byte; **INT16**s on a 32 bit processor must be aligned to a half-word boundary; all other datatypes must be aligned to a word boundary. This will be checked at run-time if it cannot be guaranteed at compile time. Some which should be detected at compile time are actually checked at run time.

  ```
  [20]BYTE array :                         -- This will be word aligned
  INT32 x RETYPES [array FROM 1 FOR 4] : -- Run-time check is inserted
  INT32 y RETYPES [array FROM i FOR 4] : -- Run-time check is inserted
  ```

```
INT32 z RETYPES [array FROM 8 FOR 4] : -- No Run-time check is inserted
```

- Tabs.
  The compiler expands tabs in source files to be every 8th character position. Tabs are permitted within a line as well as at the beginning of a line, but tabs within strings or character constants are not expanded.

- Formal parameter names.
  If a name is used more than once in a single formal parameter list, the *last* definition is used.

# 7 Usage and Alias checking

See Appendix C of "occam 2 Reference Manual", Prentice-Hall 1988for a description of the rules.

The compiler supports two switches which can be used to disable either Usage checking, or both Alias and Usage checking together. These can be set either by command line switches, or by use of the **#OPTION** compiler directive directive.

The following rules apply in addition to those described in the language manual.

## 7.1 Usage checking

A variable which is named on the right hand side of a non-**VAL** abbreviation is considered to be modified, whether or not it actually is.

Arrays as procedure formal parameters are considered to be wholly accessed if any component of the array is accessed, whether or not by a constant subscript. Similarly, free arrays (ie arrays which are accessed non-locally) of any procedure are also considered to be wholly accessed if any component of the array is accessed.

## 7.2 Alias checking

If an array is referenced in the expression of a VAL abbreviation, for example:

**x** in **VAL a IS x[i] :**

then the following rules apply to the use of the array within the scope of the abbreviation:

If the subscript is constant then elements of the array may be assigned to as long as they are only subscripted by constant values different from the abbreviated subscript. Any element of the array may also appear anywhere in the expression of a VAL abbreviation. Any other elements of the array may be non-VAL abbreviated, and run time checking code is generated if subscripts used in the abbreviation are not constant.

If the subscript is not constant then no element of the array may be assigned to unless it is first non-VAL abbreviated. The non-VAL abbreviation will have to generate run time code to check that it does not overlap the VAL abbreviation. The array may be used in the expression of a VAL abbreviation.

Elements of the array may be accessed anywhere within the scope of the abbreviation except where restricted by further abbreviations.

If an array is abbreviated in a non-VAL abbreviation, for example:

**x** in **a IS x[i] :**

then the following rules apply to the use of the array within the scope of the abbreviation:

If the subscript is constant then elements of the array may be read and assigned to as long as they are accessed by constant subscripts different from the abbreviated subscript. Other elements of the array may be abbreviated in further VAL and non-VAL abbreviations, and run time checking code is generated if subscripts used in the abbreviation are not constants.

If the subscript is not constant then the array may not be referenced at all except in abbreviations where run time checking code is needed to check that the abbreviations do not overlap.

### 7.3 Other checks

A channel formal parameter, or a free channel of a procedure, may not be used for both input and output in a procedure. This check cannot be disabled.

## 8 Error behaviour

The compiler supports the occam error modes **HALT** and **STOP**. The **HALT** mode only affects a single transputer at a time; it also requires the transputer to be executing with its *haltonerror* flag set *TRUE*. The **STOP** mode requires the transputer to be executing with its *haltonerror* flag set *FALSE*.

The compiler also supports a **UNIVERSAL** mode, in which the code will behave in **HALT** or **STOP** mode according to the state of the *haltonerror* flag.

The compiler can implement occam **UNDEFINED** (or REDUCED) mode by use of the **#OPTION** directive, or by use of a command line option which disables the insertion of run-time checks (see SW-0062 ("occam 2 Compiler specification")).

# 9 Compiler directives

The occam 2 compiler accepts a variety of compiler directives. These all begin with a hash (#).

## 9.1 #COMMENT directive

**#COMMENT** "*string*"

The **#COMMENT** directive allows comments to be placed into the object code file, which are then readable by the appropriate lister program. The ordering of comments in the object file will be the same as that of the directives in the source, but may bear no relationship to their position in the source code. **#COMMENT** directives are mainly intended for indicating version number, etc, in an object file.

For example:

```
#COMMENT "Company wide library, V2.04, 7 Dec 1989"
#COMMENT "Routine: payroll, V1.23"
```

## 9.2 #IMPORT directive

**#IMPORT** "*filename*" [ *comment* ]

The **#IMPORT** directive is treated in exactly the same way as a **#USE** directive. It is included to indicate to other tools that the imported file has not been generated by the occam toolset, and therefore should not be searched when generating makefile dependencies.

## 9.3 #INCLUDE directive

**#INCLUDE** "*filename*" [ *comment* ]

The contents of the named file is inserted into the program source at the point where the directive occurs, with the same indentation as the directive. The filename must be explicitly supplied; there is no default extension. By convention, occam source files have the suffix **.occ**, and header files consisting only of constant declarations have the suffix **.inc**. The file is searched for using the normal **ISEARCH** rules; see SW-0062 (“occam 2 Compiler specification”).

For example:

```
#INCLUDE "header.inc"
```

## 9.4 #OPTION directive

**#OPTION** "*string*" [ *comment* ]

The **#OPTION** directive allows you to specify compiler command line options within the source text of a compilation unit. The options apply to the whole compilation, and are added to the command line when the compiler is invoked. Only compiler options that relate directly to the source can be specified, namely:

| | |
|---|---|
| A | Disable alias (and usage) checking |
| E | Disable access to the compiler libraries<br>(See SW-0063 (“occam 2 Compiler Library specification”)) |
| G | Allow sequential code inserts (**ASM** and **GUY**) |
| K | Disable the insertion of run-time range checks |
| N | Disable usage checking |
| U | Disable the insertion of any extra run-time error checks |
| V | Disable the use of a separate vector space |
| W | Enable full code inserts (**ASM** and **GUY**) |
| Y | Use instruction i/o rather than library calls |

The options characters are simply listed in the string in upper or lower case. No ‘escape’ character is required or allowed. Spaces are allowed in the string. All other text on the line is ignored. For example:

```
#OPTION "A W V" -- disable alias checking, full code inserts, no vecspace
```

The **#OPTION** directive can only appear in the main file of the compilation unit to which it applies; it cannot be nested in an include file. It must also be the first non-blank or non-comment text in the source file.

## 9.5 #PRAGMA directive

**#PRAGMA** *pragma-name* [ { *values* } ]

If the *pragma-name* is not recognised, the compiler will generate a warning.

### 9.5.1 EXTERNAL pragma

**#PRAGMA EXTERNAL "***declaration***"** [ *comment* ]

This allows access to other language compilations. *declaration* is a PROC or FUNCTION heading, with formal parameters, which indicates the required calling convention for calls to the external routine (see SW-0064 (“occam 2 Run time model”) for details of occam calling conventions). This is followed (within the string) by two numbers in decimal, indicating the number of workspace and vectorspace slots (words) to reserve for that call. The number of workspace slots should not include those needed to set up the parameters for the call. The number of vectorspace slots defaults to 0 if not explicitly provided. Note that if the vectorspace requirement is zero, then no vectorspace pointer parameter will be passed to the routine.

Note that it is important to ensure that enough space is allocated, both for workspace and vectorspace, because the compiler cannot check for overruns.

The syntax of the *declaration* is as follows:

*formal procedure or function heading* = *workspace* [ , *vectorspace* ]

Examples:

```
#PRAGMA EXTERNAL "PROC p1 (VAL INT x, y) = 20"
#PRAGMA EXTERNAL "PROC p2 (VAL INT x, y) = 20, 100"
#PRAGMA EXTERNAL "INT FUNCTION f1 (VAL INT x, y) = 50"
#PRAGMA EXTERNAL "INT FUNCTION f2 (VAL INT x, y) = 50, 0"
```

The procedure or function name is the name by which the external routine is accessed from the occam source. It is also the name which will be used by the linker to access the external language function, though this may be modified by use of the TRANSLATE pragma.

### 9.5.2 LINKAGE pragma

**#PRAGMA LINKAGE** [ **"***section-name***"** ] [ *comment* ]

This is used to enable you to change the order in which code modules are linked together; this may help use faster on-chip RAM.

Normally the compiler creates the object code into a section named **"text%base"**. This pragma causes the compiler to change the name of the section to that supplied by the user in the string. If no string is present, **"pri%text%base"** is used, this section being inserted at the front by the linker in the default case. A user can override the default section ordering, to place the named sections in any required ordering, by supplying linkage commands when invoking the linker (see SW-0041 ("Linker Command File Reference Manual")).

The linkage directive should appear at the start of the source code, immediately following the **#OPTION** directive (if it exists).

### 9.5.3 TRANSLATE pragma

**#PRAGMA TRANSLATE** *identifier* **"***string***"** [ *comment* ]

This is used to enable linkage with routines whose entrypoint names do not correspond to occam syntax for identifier names; both imported names to be called by this compilation unit, and exported names defined in this compilation unit. An entrypoint is a name which is visible to the linker. Thus procedures and functions declared at the outermost level of a compilation unit are entrypoints, whereas nested procedures and functions are not.

Any entrypoint defined in the compilation unit whose name matches *identifier* is translated to *string* when inserted into the object file, and hence can only be referenced as *string* when linking. *string* may not contain the NULL character (**'*#00'**).

Any entrypoints in **#USE**d libraries and other compilation units whose names match *string* can be referred to within the compilation unit as *identifier*. This also applies to identifiers defined by EXTERNAL pragmas. TRANSLATE pragmas must *precede* any reference to their identifier.

For example:

```
#PRAGMA TRANSLATE c.routine "c_routine"
#PRAGMA EXTERNAL "PROC c.routine () = 100"
```

## 9.6 #SC directive

**#SC "***filename***"** [ *comment* ]

This directive used to be available for compatibility with existing TDS code. It is no longer supported, and will generate an error message. The **#USE** directive should be used instead.

## 9.7 #USE directive

**#USE "***filename***"** [ *comment* ]

This imports all the procedure and function declarations from the named file so that they are visible in the scope of the **#USE** directive. The file may be either a simple compilation unit object file (in TCOFF object file format), or a file consisting of several object files concatenated together, or a TCOFF library file.

Only those declarations of occam procedures and functions which have been compiled for a ‘compatible’ processor and error mode are imported. Any ‘incompatible’ declarations are ignored. See SW-0062 (“occam 2 Compiler specification”). Any names in the library which do not conform to occam syntax, and which have not been translated by means of a TRANSLATE pragma will be ignored. Note that this means that a TRANSLATE pragma must precede its related **#USE**.

If no suffix is supplied as part of the filename, the same suffix as the current output file is used. The file is searched for using the normal **ISEARCH** rules. For details of how the output file name is generated, and of **ISEARCH**, see SW-0062 (“occam 2 Compiler specification”).

For example:

```
#USE "module"
#USE "library.lib"
#USE "module.tco"
#USE "module.t2h"
```

# 10 Predefined routines

The compiler supports a number of *Predefined* routines. These are procedure and function definitions which are automatically visible to the programmer, needing no explicit library **#USE** directive to reference them. The names are not *reserved*. Therefore it is perfectly legal to reuse these names. However this should be discouraged.

Many of these predefined routines are compiled inline into sequences of transputer instructions, but others are compiled as calls to standard libraries. See SW-0063 ("occam 2 Compiler Library specification") for details.

## 10.1 Multiple length integer arithmetic functions

The functions listed here and described in Appendix K of "occam 2 Reference Manual", Prentice-Hall 1988, are available as predefined routines. These routines are all compiled inline into sequences of transputer instructions (see SW-0078 ("Code generated for predefined routines")).

```
INT            FUNCTION  LONGADD (VAL INT left, right, carry.in)
INT            FUNCTION  LONGSUM (VAL INT left, right, carry.in)
INT            FUNCTION  LONGSUB (VAL INT left, right, borrow.in)
INT, INT       FUNCTION  LONGDIFF (VAL INT left, right, borrow.in)
INT, INT       FUNCTION  LONGPROD (VAL INT left, right, carry.in)
INT, INT       FUNCTION  LONGDIV (VAL INT dvd.hi, dvd.lo, dvsr)
INT, INT       FUNCTION  SHIFTRIGHT (VAL INT hi.in, lo.in, places)
INT, INT       FUNCTION  SHIFTLEFT (VAL INT hi.in, lo.in, places)
INT, INT, INT  FUNCTION  NORMALISE (VAL INT hi.in, lo.in)
INT            FUNCTION  ASHIFTRIGHT (VAL INT argument, places)
INT            FUNCTION  ASHIFTLEFT (VAL INT argument, places)
INT            FUNCTION  ROTATERIGHT (VAL INT argument, places)
INT            FUNCTION  ROTATELEFT (VAL INT argument, places)
```

**SHIFTRIGHT** and **SHIFTLEFT** return zeroes when the number of places to shift is negative, or is greater than twice the transputer's wordlength, in which case they may take a long time to execute.

**ASHIFTRIGHT**, **ASHIFTLEFT**, **ROTATERIGHT**, and **ROTATETLEFT** are all invalid when the number of places to shift is negative or exceeds the transputer's word length.

## 10.2 Floating point functions

The functions listed here and described in Appendix L of “occam 2 Reference Manual”, Prentice-Hall 1988, are available as predefined routines. Some of these routines are compiled inline into sequences of transputer instructions, others are compiled as calls to standard libraries. This may depend on the processor type (see SW-0063 (“occam 2 Compiler Library specification”) and SW-0078 (“Code generated for predefined routines”)).

```
REAL32                FUNCTION  ABS (VAL REAL32 X)
REAL64                FUNCTION  DABS (VAL REAL64 X)
BOOL                  FUNCTION  ISNAN (VAL REAL32 X)
BOOL                  FUNCTION  DISNAN (VAL REAL64 X)
BOOL                  FUNCTION  NOTFINITE (VAL REAL32 X)
BOOL                  FUNCTION  DNOTFINITE (VAL REAL64 X)
BOOL                  FUNCTION  ORDERED (VAL REAL32 X, Y)
BOOL                  FUNCTION  DORDERED (VAL REAL64 X, Y)
REAL32                FUNCTION  MULBY2 (VAL REAL32 X)
REAL64                FUNCTION  DMULBY2 (VAL REAL64 X)
REAL32                FUNCTION  DIVBY2 (VAL REAL32 X)
REAL64                FUNCTION  DDIVBY2 (VAL REAL64 X)
REAL32                FUNCTION  SQRT (VAL REAL32 X)
REAL64                FUNCTION  DSQRT (VAL REAL64 X)
REAL32                FUNCTION  FPINT (VAL REAL32 X)
REAL64                FUNCTION  DFPINT (VAL REAL64 X)
REAL32                FUNCTION  MINUSX (VAL REAL32 X)
REAL64                FUNCTION  DMINUSX (VAL REAL64 X)
REAL32                FUNCTION  SCALEB (VAL REAL32 X, VAL INT n)
REAL64                FUNCTION  DSCALEB (VAL REAL64 X, VAL INT n)
REAL32                FUNCTION  COPYSIGN (VAL REAL32 X, Y)
REAL64                FUNCTION  DCOPYSIGN (VAL REAL64 X, Y)
REAL32                FUNCTION  NEXTAFTER (VAL REAL32 X, Y)
REAL64                FUNCTION  DNEXTAFTER (VAL REAL64 X, Y)
REAL32                FUNCTION  LOGB (VAL REAL32 X)
REAL64                FUNCTION  DLOGB (VAL REAL64 X)
INT, REAL32           FUNCTION  FLOATING.UNPACK (VAL REAL32 X)
INT, REAL64           FUNCTION  DFLOATING.UNPACK (VAL REAL64 X)
BOOL, INT32, REAL32   FUNCTION  ARGUMENT.REDUCE (VAL REAL32 X, Y, Y.err)
BOOL, INT32, REAL64   FUNCTION  DARGUMENT.REDUCE (VAL REAL64 X, Y, Y.err)
```

## 10.3 Full IEEE arithmetic functions

The functions listed here and described in Appendix M of “occam 2 Reference Manual”, Prentice-Hall 1988, are available as predefined routines. These routines are all compiled as calls to standard libraries (see SW-0063 (“occam 2 Compiler Library specification”)).

```
REAL32         FUNCTION  REAL32OP (VAL REAL32 X, VAL INT Op, VAL REAL32 Y)
REAL64         FUNCTION  REAL64OP (VAL REAL64 X, VAL INT Op, VAL REAL64 Y)
BOOL, REAL32   FUNCTION  IEEE32OP (VAL REAL32 X, VAL INT Rm, Op,
                                    VAL REAL32 Y)
BOOL, REAL64   FUNCTION  IEEE64OP (VAL REAL64 X, VAL INT Rm, Op,
                                    VAL REAL64 Y)
REAL32         FUNCTION  REAL32REM (VAL REAL32 X, VAL REAL32 Y)
REAL64         FUNCTION  REAL64REM (VAL REAL64 X, VAL REAL64 Y)
BOOL           FUNCTION  REAL32EQ (VAL REAL32 X, Y)
BOOL           FUNCTION  REAL64EQ (VAL REAL64 X, Y)
BOOL           FUNCTION  REAL32GT (VAL REAL32 X, Y)
BOOL           FUNCTION  REAL64GT (VAL REAL64 X, Y)
INT            FUNCTION  IEEECOMPARE (VAL REAL32 X, Y)
INT            FUNCTION  DIEEECOMPARE (VAL REAL64 X, Y)
```

The following functions are analogous, but were omitted from “occam 2 Reference Manual”, Prentice-Hall 1988. They calculate **X REM Y** according to the IEEE standard, returning two results, a boolean which is **TRUE** if an error has occured, and the result of the remainder operation. No rounding mode is required since remainder is exact.

```
BOOL, REAL32   FUNCTION  IEEE32REM (VAL REAL32 X, VAL REAL32 Y)
BOOL, REAL64   FUNCTION  IEEE64REM (VAL REAL64 X, VAL REAL64 Y)
```

## 10.4 Transputer-specific predefined routines

The following functions and procedures are supplied as predefined routines to enable use of the more advanced features of the transputer instruction set.

### 10.4.1 General purpose routines

These routines are all compiled inline into sequences of transputer instructions (see SW-0078 ("Code generated for predefined routines")).

- **`PROC CAUSEERROR ()`**
  This inserts instructions into the code to set the transputer error flag. This is then treated in exactly the same way as any other error would be treated in the error mode in which the program is compiled. For example, in HALT mode the whole processor will halt.

- **`PROC RESCHEDULE ()`**
  This causes the current process to be moved to the end of the current priority scheduling queue; in effect, it forces a 'timeslice' (even in high priority).

- **`PROC LOAD.BYTE.VECTOR (INT here, VAL []BYTE bytes)`**
  Assigns the machine address of the byte array **`bytes`** to **`here`**. This can be used in conjunction with **`RETYPES`** to find the address of any variable.

- **`PROC LOAD.INPUT.CHANNEL (INT here, CHAN OF ANY in)`**
  Assigns the machine address of the channel word of **`in`** to **`here`**.

- **`PROC LOAD.INPUT.CHANNEL.VECTOR (INT here, []CHAN OF ANY in)`**
  Assigns the machine address of the base element of the channel array **`in`** (ie. the base of the array of pointers) to **`here`**.

- **`PROC LOAD.OUTPUT.CHANNEL (INT here, CHAN OF ANY out)`**
  Assigns the machine address of the channel word of **`out`** to **`here`**.

- **`PROC LOAD.OUTPUT.CHANNEL.VECTOR (INT here, []CHAN OF ANY out)`**
  Assigns the machine address of the base element of the channel array **`out`** (ie. the base of the array of pointers) to **`here`**.

### 10.4.2 Block Move routines

These routines are all compiled inline into sequences of transputer instructions on processors which support the appropriate instructions, or as calls to standard library routines for other processor types (see SW-0063 ("occam 2 Compiler Library specification") and SW-0078 ("Code generated for predefined routines")).

- **PROC MOVE2D (VAL [][]BYTE source, VAL INT sx, sy,**
  **[][]BYTE dest, VAL INT dx, dy,**
  **VAL INT width, length)**

  This is *equivalent* to:

```
SEQ y = 0 FOR length
  [dest[y+dy] FROM dx FOR width] := [source[y+sy] FROM sx FOR width]
```

  This block moves a block of size **width,length** which starts at byte **source[sy][sx]** to the block starting at byte **dest[dy][dx]**.

- **PROC DRAW2D (VAL [][]BYTE source, VAL INT sx, sy,**
  **[][]BYTE dest, VAL INT dx, dy,**
  **VAL INT width, length)**

  This is *equivalent* to:

```
SEQ line = 0 FOR length
  SEQ point = 0 FOR width
    VAL temp IS source[line+sy][point+sx] :
    IF
      temp <> (0(BYTE))
        dest[line+dy][point+dx] := temp
      TRUE
        SKIP
```

  This does the same as **MOVE2D**, but destination bytes corresponding to zeroes in the source are not modified.

- **PROC CLIP2D (VAL [][]BYTE source, VAL INT sx, sy,**
  **[][]BYTE dest, VAL INT dx, dy,**
  **VAL INT width, length)**

  This is *equivalent* to:

```
SEQ line = 0 FOR length
  SEQ point = 0 FOR width
    VAL temp IS source[line+sy][point+sx] :
    IF
      temp = (0(BYTE))
        dest[line+dy][point+dx] := 0(BYTE)
      TRUE
        SKIP
```

  This does the same as **MOVE2D**, but only zeroes those destination bytes corresponding to zeroes in the source.

### 10.4.3 Cyclic redundancy checking

These routines are all compiled inline into sequences of transputer instructions on processors which support the appropriate instructions, or as calls to standard library routines for other processor types (see SW-0063 ("occam 2 Compiler Library specification") and SW-0078 ("Code generated for predefined routines")).

- **`INT FUNCTION CRCWORD (VAL INT data, CRCIn, generator)`**

  This is *equivalent* to:

```
INT MyData, CRCOut, OldCRC :
VALOF
  SEQ
    MyData, CRCOut := data, CRCIn
    SEQ i = 0 FOR BitsPerWord  -- 16 or 32
      SEQ
        OldCRC := CRCOut
        CRCOut, MyData := SHIFTLEFT (CRCOut, MyData, 1)
        IF
          OldCRC < 0 -- MSB of CRC = 1
            CRCOut := CRCOut >< generator
          TRUE
            SKIP
  RESULT CRCOut
```

  It performs a cyclic redundancy check over the single word **`data`** using the CRC value obtained from the previous call. **`generator`** is the CRC polynomial generator.

- **`INT FUNCTION CRCBYTE (VAL INT data, CRCIn, generator)`**

  This is *equivalent* to:

```
INT MyData, CRCOut, OldCRC :
VALOF
  SEQ
    MyData, CRCOut := data, CRCIn
    SEQ i = 0 FOR 8
      SEQ
        OldCRC := CRCOut
        CRCOut, MyData := SHIFTLEFT (CRCOut, MyData, 1)
        IF
          OldCRC < 0 -- MSB of CRC = 1
            CRCOut := CRCOut >< generator
          TRUE
            SKIP
  RESULT CRCOut
```

  As **`CRCWORD`** but performs the check over a single byte. The byte processed is contained in the *most significant* byte of the word **`data`**.

### 10.4.4 Bit manipulation routines

These routines are all compiled inline into sequences of transputer instructions on processors which support the appropriate instructions, or as calls to standard library routines for other processor types (see SW-0063 ("occam 2 Compiler Library specification") and SW-0078 ("Code generated for predefined routines")).

- **`INT FUNCTION BITCOUNT (VAL INT Word, CountIn)`**
  Counts the number of bits set to 1 in **`Word`**, adds it to **`CountIn`**, and returns the total.

- **`INT FUNCTION BITREVWORD (VAL INT x)`**
  Forms an **`INT`** containing the bit reversal of **`x`**.

- **`INT FUNCTION BITREVNBITS (VAL INT x, n)`**
  Forms an **`INT`** containing the **`n`** least significant bits of **`x`** in reverse order. The upper bits are set to zero. The operation is invalid if **`n`** is negative or greater than the number of bits in a word.

### 10.4.5 Floating point support routines

These routines are all compiled inline into sequences of transputer instructions on processors which support the appropriate instructions, or as calls to standard library routines for other processor types (see SW-0063 ("occam 2 Compiler Library specification") and SW-0078 ("Code generated for predefined routines")).

- **`INT FUNCTION FRACMUL (VAL INT x, y)`**
  Performs a fixed point multiplication of **`x`** and **`y`**, treating each as a binary fraction in the range [-1, 1), and returning their product rounded to the nearest available representation. The value of the fractions represented by the arguments and result can be obtained by multiplying their **`INT`** value by $2^{-31}$ (or $2^{-15}$ on a 16-bit processor). The result can overflow if both **`x`** and **`y`** are -1.

  This predefine is compiled inline into a sequence of transputer instructions on 32-bit processors (see SW-0078 ("Code generated for predefined routines")), or as a call to a standard library routine for 16-bit processors.

- **`INT, INT, INT FUNCTION UNPACKSN (VAL INT X)`**
  This returns three parameters; from left to right they are **`Xfrac`**, **`Xexp`**, and **`Type`**. It unpacks **`X`**, regarded as an IEEE single length real number (ie a **`RETYPED REAL32`**), into **`Xexp`**, the (biased) exponent, and **`Xfrac`**, the fractional part. It also returns an integer defining the **`Type`** of **`X`**, ignoring the sign bit:

  | **`Type`** | Reason |
  |---|---|
  | 0 | **`X`** is zero |
  | 1 | **`X`** is a normalised or denormalised number |
  | 2 | **`X`** is **Inf** |
  | 3 | **`X`** is **NaN** |

  This predefine is compiled inline into a sequence of transputer instructions on 32-bit processors such as the **`IMS T425`**, which do not have a floating point unit, but do have specialised instructions for floating point support (see SW-0078 ("Code generated for predefined routines")). It is compiled as a call to a standard library routine for other 32-bit processors. It is invalid on 16-bit processors, since **`Xfrac`** cannot fit into an **`INT`**.

- **`INT FUNCTION ROUNDSN (VAL INT Yexp, Yfrac, Yguard)`**
  This takes a possibly unnormalised fraction, guard word, and exponent, and returns the IEEE floating point value it represents. It takes care of all the normalisation, postnormalisation, rounding, and packing of the result. The rounding mode used is round to nearest. The exponent should already be biased.

  The function normalises and postnormalises the number represented by **`Yexp`**, **`Yfrac`** and **`Yguard`** into local variables **`Xexp`**, **`Xfrac`** and **`Xguard`**. It then packs the (biased) exponent **`Xexp`** and fraction **`Xfrac`** into the result, rounding using the extra bits in **`Xguard`**. The sign bit is set to 0. If overflow occurs, **Inf** is returned.

  This predefine is compiled inline into a sequence of transputer instructions on 32-bit processors such as the **`IMS T425`**, which do not have a floating point unit, but do have specialised instructions for floating point support (see SW-0078 (“Code generated for predefined routines”)). It is compiled as a call to a standard library routine for other 32-bit processors. It is invalid on 16-bit processors, since **`Xfrac`** cannot fit into an **`INT`**.

### 10.4.6 Dynamic code loading

- 
```
PROC KERNEL.RUN (VAL []BYTE code, VAL INT entry.offset,
                 []INT workspace,
                 VAL INT number.of.parameters)
```

  This is supplied to enable a user to call a piece of transputer code without using an occam call.

  Note that the calling convention for arrays of channels has changed for this compiler; it now passes arrays of channels as arrays of pointers to channels. See SW-0064 (“occam 2 Run time model”) for details of occam calling conventions.

  If the code has been compiled with an INMOS compiler, then the first byte of the **`code`** array *must* be aligned to a word boundary. The code is executed starting at **`code[entry.offset]`**. The user must have already supplied any necessary parameters into the **`workspace`** array. The topmost word is reserved for use by **`KERNEL.RUN`**; the next **`number.of.parameters`** words are assumed to have been set up by the user, and the code is entered with the transputer’s *Workspace pointer* pointing at the next word, running in the same priority as the process which called it. That process will not proceed until the executing code restores the *Workspace pointer* to the value it had upon entry and executes a **`ret`** instruction.

  It is the user’s responsibility to ensure that the **`workspace`** array is large enough to accommodate all the stack usage requirements. If the code requires a vectorspace pointer, the user should set up an extra parameter which points to an area of memory (eg an **`INT`** array) which is large enough to accommodate all the vectorspace requirements. Note that the compiler requires that **`number.of.parameters`** is a compile-time constant, and has value $\geq$ 3.

  The predefine is compiled inline into a sequence of transputer instructions (see SW-0078 (“Code generated for predefined routines”)).

# 11 Other Standard libraries

## 11.1 Elementary function library

The functions **SIN** etc, as listed in Appendix J.4 and described in Appendix N of “occam 2 Reference Manual”, Prentice-Hall 1988, are *not* available as predefined routines. They must be explicitly referenced by **#USE**-ing a library. See SW-0114 (“occam 2 User Library Specification”).

## 11.2 Value and string conversion procedures

The functions **INTTOSTRING** etc, as listed in Appendix J.5 and described in Appendix O of “occam 2 Reference Manual”, Prentice-Hall 1988, are *not* available as predefined routines. They must be explicitly referenced by **#USE**-ing a library. See SW-0114 (“occam 2 User Library Specification”).

# 12 Inline transputer code insertion

The occam compiler supports the insertion of transputer code directly into an occam program. See “Transputer Instruction Set - A compiler writer’s guide”, Prentice-Hall 1988 for details of the transputer instruction set. These facilities must be specifically enabled (see SW-0062 (“occam 2 Compiler specification”)). Two levels are available; ‘Sequential’ instructions are enabled by an option. This allows access to all transputer instructions for a particular processor which cannot normally affect parallel processes and scheduling. ‘Full’ instructions can be enabled by a different option. This allows access to the complete range of instructions supported by that processor.

## 12.1 GUY Construct

For compatibility with existing occam compilers, code inserts are provided by the **GUY** construct. The syntax of this construct is:

| | | |
|---|---|---|
| *process* | = | *guy.construct* |
| *guy.construct* | = | **GUY**<br>{ *guy.line* } |
| *guy.line* | = | *primary* |
| | \| | **[ STEP ]** *secondary* |
| | \| | *labeldef* |
| *primary* | = | *primary.op* **.** *label* |
| | \| | *primary.op constant.expression* |
| | \| | *load.or.store.op guy.expr* |
| *labeldef* | = | **:** *label* |
| *secondary* | = | *secondary.op* |
| *guy.expr* | = | *name* { **[** *constant.expression* **]** } |
| *primary.op* | = | any primary instruction (in upper-case letters) |
| *secondary.op* | = | any secondary instruction (in upper-case letters) |
| *load.or.store.op* | = | **LDL** \| **LDNL** \| **LDLP** \| **STL** \| **STNL** |

*guy.expr* expressions must be scalar variables, or array accesses with constant subscripts (expressions may, of course, be constant). Access to channels is no longer supported within **GUY** code.

There are some problems with the semantics of **GUY** code.

1 Any primary instruction may take a label as operand even though it is meaningless for anything other than a jump or load constant. (The operand is evaluated as the difference between the address of the label and the address of the next instruction.)

2 When given a constant operand, the local / nonlocal load and store instruction pairs (ie. **LDL / LDNL**, **LDLP / LDNLP**, **STL / STNL**) behave as expected and produce the appropriate

instruction with the given operand. However, when given a *guy.expr* operand, both instructions in a pair behave identically and not necessarily at all as expected. **LDL / LDNL** are treated as *load*, **LDLP / LDNLP** are treated as *load pointer*, and **STL / STNL** are treated as *store*. These operations may include chaining back down static links to load a variable (given **LDL** on a non-local variable), or not (given **LDNL** on a local variable, **LDL** will be generated!), or even loading up a pointer in order to store into an array subscription.

The **STEP** form is provided to allow testing of transputer instructions. It can be replaced by use of the **OPR** primary instruction so is obsolete.

## 12.2 ASM Construct

The **ASM** construct provides the ability to insert transputer code sequences into occam programs. Its syntax is similar to (but not identical to) that of the obsolete **GUY** construct which has just been described, except that it is introduced by the **ASM** keyword. However, the *semantics* are much more secure. A primary instruction in **ASM** code will now always generate that primary instruction in the object file; this was not the case with the **GUY** construct.

| | | |
|---|---|---|
| *process* | = | *asm.construct* |
| *asm.construct* | = | **ASM** |
| | | { *asm.line* } |
| *asm.line* | = | *primary.op constant.expression* |
| | \| | *load.or.store.op name* |
| | \| | *branch.op* **:** *label* |
| | \| | *secondary.op* |
| | \| | *pseudo.op* |
| | \| | *labeldef* |
| *labeldef* | = | **:** *label* |
| *primary.op* | = | any primary instruction (in upper-case letters) |
| *load.or.store.op* | = | **LDL** \| **LDNL** \| **LDLP** \| **LDNLP** |
| | \| | **STL** \| **STNL** |
| *branch.op* | = | **J** \| **CJ** \| **CALL** |
| *secondary.op* | = | any secondary instruction (in upper-case letters) |
| *pseudo.op* | = | **LD** *asm.exp* |
| | \| | **LDAB** *asm.exp, asm.exp* |
| | \| | **LDABC** *asm.exp, asm.exp, asm.exp* |
| | \| | **ST** *element* |
| | \| | **STAB** *element, element* |
| | \| | **STABC** *element, element, element* |
| | \| | **BYTE** {, *constant.expression* } |
| | \| | **WORD** {, *constant.expression* } |
| | \| | **ALIGN** |
| | \| | **LDLABELDIFF** **:** *label* – **:** *label* |
| *asm.exp* | = | **ADDRESSOF** *element* |
| | \| | *expression* |

### 12.2.1 ASM instructions

The primary instructions which perform loads and stores are allowed to take a symbolic name as their operand; they evaluate to the primary instruction with an operand *equal to that symbol's offset in workspace*. Note that this means, for example, that **LDL x** where **x** is a non-**VAL** parameter, will return the pointer to **x**. This also means that if **x** is a non-local variable, the operand used will be the variable's offset in the non-local workspace. Primary instructions with symbolic name operands should only be used in special cases; you would normally use the pseudo ops as described below.

The assembler will optimise away primary instructions which are known to be no-ops. These are

```
AJW   0
ADC   0
LDNLP 0
```

**PFIX 0** should be used where a **NOP** byte is required, or the **BYTE** pseudo-op could be used.

Secondary instructions, and the **fpentry** instructions, simply expand out to the correct byte sequence, as expected.

You may only branch to a label defined within the same procedure or function. It is illegal to declare two labels with the same name in the same procedure. It is currently illegal to use labels inside a procedure or function which will be **INLINE**d.

### 12.2.2 Pseudo operations

The *pseudo.op* operations are defined as follows:

| | |
|---|---|
| **LD** | Loads a value into the Areg. May use other stack slots and/or temporaries. |
| **LDAB** | Loads values into the Areg and Breg. The left hand expression ends up in Areg. May use other stack slots and/or temporaries. |
| **LDABC** | Loads values into the Areg, Breg and Creg. The leftmost expression ends up in Areg. May use temporaries. |
| **ST** | Stores the value from the Areg. May use other stack slots and/or temporaries. |
| **STAB** | Stores values from the Areg and Breg. The left hand element receives Areg. May use other stack slots and/or temporaries. |
| **STABC** | Stores values into the Areg, Breg and Creg. The leftmost element receives Areg. May use temporaries. |
| **BYTE** | Inserts the following constant **BYTE** value(s) into the code. The expression may be either a single **BYTE**, or a **BYTE** table or string, or a comma separated list of such items. |
| **WORD** | Inserts the following constant **INT** value(s) into the code. The expression may be either a single integer, or an integer table, or a comma separated list of such items. |
| **LDLABELDIFF** | Calculates the difference, *n*, between two labels and inserts a **LDC** *n*. |
| **ALIGN** | Inserts zero or more **PFIX 0** instructions until aligned to a word boundary. *Currently not implemented.* |

Expressions used in *load* pseudo-ops must be word sized or smaller. To load a floating point value, use a **LD** to load its address, then a **FPLDNLSN** or **FPLDNLDB** as required. Elements used in *store* pseudo-ops must be word sized (or smaller?).

### 12.2.3 Special names

The following special names are available as constants inside **ASM** expressions.

**.WSSIZE** Evaluates to the size of the current procedure's workspace. This will be the workspace offset of the return address, except within a replicated **PAR**, where it will be the size of that replication's workspace requirement.

**.VSPTR** Evaluates to the workspace offset of the vectorspace pointer. If inside a replicated **PAR**, it points to the vectorspace pointer for that branch only. A compile time error is generated if there is no vectorspace pointer because no vectors have been created.

**.STATIC** Evaluates to the workspace offset of the static link. If inside a replicated **PAR**, it points to the static link for that branch only. A compile time error is generated if there is no static link.

For example, to determine the return address of a procedure, you would use: **LDL .WSSIZE**

There is no checking of 'suitability', hence **J .WSSIZE**, etc, is legal.

## 12.3 Differences between ASM and GUY

A summary of the differences between **ASM** and **GUY** follows:

- Symbolic access to primaries.
  The instructions **LDL, LDNL, LDLP, LDNLP, STL, STNL** now have a different behaviour. Whereas in **GUY**, each behaved as simply a **LD** *x*, **LD ADDRESSOF** *x*, or **ST** *x*, in **ASM** they will evaluate to the primary instruction followed by the offset of that variable in workspace.

  A sequence of loads or stores in **GUY** should be changed to one of the **ASM** pseudo operations.

- References to labels.
  References to labels in **GUY** code are preceded by a dot. This must be changed to a colon (**:**) in **ASM**.

  In some cases, where in **GUY** you would **LDC .label**, you can now do this better with in **ASM** with **LDLABELDIFF**.

- Channel accesses.
  Symbolic access to channels is not permitted in **GUY** code although it was previously. (This has been disallowed because the internal representation of channels has changed).

  In **ASM**, **LDL chan** will return a pointer to the channel word.

## 12.4 Sequential code insertion

Here follows the total list of the transputer instructions which are permitted when 'sequential' code insertion is enabled. You may only use those instructions which exist on the target processor.

The **ASM** pseudo-operations are also permitted when sequential code insertion is enabled.

ADC, ADD, AND, BCNT, BITCNT, BITREVNBITS, BITREVWORD, BSUB, CCNT1, CFLERR, CJ, CRCBYTE, CRCWORD, CSNGL, CSUB0, CWORD, DIFF, DIV, DUP, EQC FMUL, FPADD, FPB32TOR64, FPCHKERR, FPDIV, FPDUP, FPEQ, FPGT, FPI32TOR32, FPI32TOR64, FPINT, FPLDNLADDDB, FPLDNLADDSN, FPLDNLDB, FPLDNLDBI, FPLDNLMULDB, FPLDNLMULSN, FPLDNLSN, FPLDNLSNI, FPLDZERODB, FPLDZEROSN, FPMUL, FPNAN, FPNOTFINITE, FPORDERED, FPREMFIRST, FPREMSTEP, FPREV, FPRTOI32, FPSTNLDB, FPSTNLI32, FPSTNLSN, FPSUB, FPTESTERR, FPUABS, FPUCHKI32, FPUCHKI64, FPUCLRERR, FPUDIVBY2, FPUEXPDEC32, FPUEXPINC32, FPUMULBY2,

FPUNOROUND, FPUR32TOR64, FPUR64TOR32, FPURM, FPURN, FPURP, FPURZ, FPUSETERR, FPUSQRTFIRST, FPUSQRTLAST, FPUSQRTSTEP, GT, J, LADD, LB, LDC LDDEVID, LDIFF, LDINF, LDIV, LDL, LDLP, LDMEMSTARTVAL, LDNL, LDNLP, LDPI, LDPRI, LDTIMER, LMUL, LSHL, LSHR, LSUB, LSUM, MINT, MOVE, MOVE2DALL, MOVE2DINIT, MOVE2DNONZERO, MOVE2DZERO, MUL, NORM, NOT, OR, POP, POSTNORMSN, PROD, REM, REV, ROUNDSN, SB, SETERR, SHL, SHR, STL, STNL, STTIMER, SUB, SUM, TESTERR, TESTHALTERR, TESTPRANAL, UNPACKSN, WCNT, WSUB, WSUBDB, XDBLE, XOR, XWORD

### 12.5 Full code insertion

You can use any instruction listed in the compiler writer's guide, or listed in a datasheet for that processor.

## 13 The end