

## **Contents**

Welcome to T. This is an easy to learn, user friendly, high level, computer programming language. T is more like natural English than most other computer languages and this makes a T program both easy to write and easy to understand.

Help contains a tutorial on the T computer language, and operating instructions for the interpreter.

Main menu commands

Getting started

Working with data

Some input and output

Looping and jumping

Using subprograms

Language reference

Source code

## Getting started

This topic introduces you to the usage of the T interpreter. It shows you how to enter, debug, and run a program.

### writing a T program

Every T program is a sequence of declarations and statements that begins and ends within a program module. The following is a complete program:

```
program  
  put "Hello!"  
end program
```

It is made up of key words, literals, special symbols, and standard subprograms. In the example above, **put** is a key word, " is a special symbol, and Hello! is a literal string.

A T program is modular. The program module defines both the start and end of a program. All executable statements are contained within this module or within subprogram modules. Subprograms, procedures and functions, are used to create a program from small manageable pieces. T allows you to define procedure modules and function modules as needed for your programs.

The T programming language supports a variety of data types. You may declare named variables and named constants; you may define your own data type using a type definition. T has only two numerical data types; integers and floating point numbers. It has a boolean type, a character type, and a string type. It supports an array type, a record type and a union type which you the programmer define. Global variable declarations and data type definitions must be located outside of the program and any subprogram modules. Declarations are limited to the scope in which they are defined. This means that a variable named `number` declared as a global is not the same as the variable `number` declared within a subprogram.

### comments and white space

Comments, together with white space (spaces between symbols and blank lines) make a program easier to read and understand. This is important if you want to show your program to someone else or use it again yourself at some future time. T is a free form language. As long as the words and symbols are in the correct order, a program will run correctly. It is up to you to make a program easy to read and understand. Some programmers find that it is helpful to add comments and to name data and subroutines in such a way that each step of the debugger is easily understood.

### comment symbol

The character `%` indicates that all following text to the end of the line is a comment and not part of the program

### first program

This is an example of a simple program which you can use to try out the T interpreter:

```
const x := 2
const y := 4

program

    var sum : int

    sum := x + y
    put x, " +", y, " =", sum
```

### **end program**

Enter this program, and using the instructions which follow. The first step is to open a new file using the File-New command.

### source files

Enter the sample program into the editor. After you have finished use the File-Saveas command to save your program. Type a name for your program; how about `first.t`? Note the `.t`; this is the file name extension used by the editor to identify a program's source files. Press <Enter> when you've finished. At the top of the edit window, the `no_name#.t` should disappear and be replaced with the name you typed.

### project file

Next you must create a project file. Using the editor, create a file containing the name of the file just created. Save this file with a file name in the form `projname.prt` using the File-Saveas command. The project file should contain a list of program files which makes up a program. This feature allows you to create multiple source file programs.

### running a T program

Load the project file with the Project-Load command; this will enable the commands which allow you to run and debug your programs. Now use the Project-Run command or the <F9> function key to run the program. The editor will start the interpreter in its run mode. The interpreter parses all the files listed in the project file and runs the program. Text output is directed to a `projname.out` file which you can edit and save.

### debugging a T program

Load a project using the Project-Load command and use the Project-Step command or the <F7> function key to start the interpreter in its debugging mode. If you entered correct code

you should see one of your files with a bar highlighting the first executable line.

If you entered incorrect code you will see a list of errors displayed in a dialog box. Using the mouse to select error messages will show you where the errors are. Each line containing an error message shows the file and location of the error using the following format:

```
filename.ext [line:column] description
```

Let's assume that either you entered the program correctly or you corrected any errors and started the interpreter again in the debugging mode. Press the <F7> function key. The highlight will jump to the next line containing a statement. Keep doing this until the interpreter reaches the end of the program. Each line containing an executable statement was highlighted.

The other debug command, Project-Step or function key <F8>, allows you to step over a function or procedure you defined in your program rather than tracing into it. Use of this command may save you some time in debugging a large program.

### language features for debugging

Three features to make it easier for you to debug your programs.

The assert statement has the form:

```
assert boolean expression
```

If the *boolean expression* is **false** during program execution the program is halted. This program fragment would terminate a program because of invalid data:

```
get x           % from console  
assert x > 0.0  % if false, halt  
put sqrt( x )  % do if true
```

The break statement has the form:

```
break
```

It unconditionally interrupts processing and displays the trace line at the corresponding line of the source file. You may resume processing by using the Run, Trace, or Step command.

The watch procedure allows you to observe variables while debugging a program. It is written as a statement in a program with the form:

```
watch( expression )
```

When the interpreter is in the debug mode, the value of the *expression* is displayed on the screen.

## **File menu commands**

### **New**

Keyboard command: Alt+F N

Hot key: Ctrl+N

Opens a new document window with a default title and makes it the active window.

### **Open...**

Keyboard command: Alt+F O

Hot key: Ctrl+O

Allows you to select and open an existing file. The just opened file will be made the active window.

### **Save**

Keyboard command: Alt+F S

Hot key: Ctrl+S

Saves the file in the currently active window to disk. The file remains open so you can continue working on it.

### **Save As...**

Keyboard command: Alt+F A

Allows you to name a new file or save an existing file under a new name or to a different directory. The original file is not changed. The file remains open so you can continue working on it.

### **Print...**

Keyboard command: Alt+F P

Hot key: Ctrl+P

Allows you to print the file in the active window.

### **Exit**

Keyboard command: Alt+F X

Closes open files and quits the T interpreter application. You can save open files before quitting.

## **Main menu commands**

The T interpreter functions within a multiple document editor. A set of menu commands allows you to control the editor as well as the interpreter.

### **subtopics:**

[File menu commands](#)

[Edit menu commands](#)

[Search menu commands](#)

[Project menu commands](#)

[Window menu commands](#)

[Help menu commands](#)

## Looping and jumping

The T programming language provides several statements that control the sequence of program execution. Each of these control statements must be used entirely within the program module or a subprogram module.

### exit statements

An exit statement has the form:

```
exit [when boolean expression]
```

and is allowed only within a loop statement or a for statement. The exit statement causes program execution to jump to the first statement after the nearest enclosing loop or for statement. If the optional key word **when** is present, the jump is conditional and occurs only if the *boolean expression* is **true**.

### continue statements

A continue statement has the form:

```
continue [when boolean expression]
```

and, as above, is allowed only within a loop statement or a for statement. The continue statement causes program execution to jump to the first statement in the nearest enclosing loop or for statement. If the optional key word **when** is present, the jump is conditional and occurs only if the *boolean expression* is true.

### loop statements

The loop control statement has the form:

```
loop  
    declarations and statements  
end loop
```

Program execution jumps to the first statement in the loop body on reaching end loop. Note that, by itself, a loop statement is infinite; that is, it will continue indefinitely unless stopped by some other statement. An exit statement terminates the nearest enclosing loop. Declarations made within a loop are visible only within the loop body. An example:

### **program**

```
var number : int := 0  
  
loop
```

```
    incr number
    exit when number > 4
    continue when number = 2
    put number
```

```
end loop
```

```
end program
```

### for statements

The for control statement is written as:

```
    for [decreasing] name := begin...end do
        declarations and statements
    end for
```

The range following the " := " defines the beginning and ending values of the count variable name. The count limits begin and end must be integer expressions. The loop's statement list is executed once for each valid value of the count variable which is incremented by one or, if **decreasing** is included, decremented by one before repeating the statement list. As above, an exit statement can be used to terminate the loop. Declarations made within the for statement are not visible outside of the statement. An example:

### **program**

```
    var number : int := 0
    var i : int

    for i := 1...5 do

        decr number
        continue when number = -3
        put number

    end for
```

```
end program
```

### if statements

An if control statement has the form:

```
    if boolean expression then
        declarations and statements
    {elsif boolean expression then
```



```

        declarations and statements }
[else
    declarations and statements ]
end if

```

The *boolean expression* for each branch is evaluated until one of them is true. The statements in the branch are executed until a closing **elsif**, **else**, or **end if** is reached. If no *boolean expression* is true then the statements following **else**, if present, are executed. The program resumes at the first statement after **end if**. An example:

**program**

```

prompt "Enter test score:"
loop

    var mark : int

    get mark
    exit when mark < 0
    if mark > 100 then
        put "Invalid"
    elsif mark >= 93 then
        put 'A'
    elsif mark >= 85 then
        put 'B'
    elsif mark >= 78 then
        put 'C'
    elsif mark >= 70 then
        put 'D'
    else
        put 'F'
    end if

end loop

```

**end program**

case statements

A case control statement has the form:

```

case expression of
    value constant{, constant} :
        declarations and statements
    {value constant{, constant} :
        declarations and statements }
[value :

```

```
        declarations and statements ]  
end case
```

The *expression* and each *constant* must be of matching type of either integer, character, string, or an enumerated type. Declarations made within a branch are not visible outside the branch. The *expression* is evaluated and compared with each constant of each branch until one of them is true. The statements in the branch are executed until another **value** or **end case** is reached. If no match is found then the statements following an optional **value** without a *constant* are executed. The program resumes at the first statement after **end case**. An example:

**program**

```
var word : string  
  
put "enter a word from:"  
put "the rain in spain"  
prompt "enter a word:"  
loop  
  
    get word  
  
    case word of  
        value "the", "rain", "in":  
            put "ok"  
        value "spain":  
            put "done"  
            exit  
        value:  
            put "not ok"  
    end case  
  
end loop  
  
end program
```

goto statements

The goto statement causes an unconditional jump from one point in a list of statements to a named location. Jumps must be entirely within a program or subprogram module. In order to use a goto statement, a name of the location to jump to must be declared using the form:

```
label name :    % no type!
```

The goto statement can then be coded as:

```
goto name      % from here
```

```
·  
·  
·  
name :           % to here
```

This statement can be used to simplify your code by enabling jumps out of deeply nested logic or by creating jumps to a single point of return from a subprogram. The goto statement can also be used to make your program difficult to understand.

**union**        keyword

usage

**union**

*item*{, *item*} : *type specification*

    {*item*{, *item*} : *type specification*}

**end union**

remarks

Keyword is used to declare a union of data items. To access elements of a union, use the item selector "." between a variable name and the *item*.

see also

Working with data

break        keyword

usage

break

remarks

Interrupts program execution and displays the corresponding line in the source file.

see also

Getting started

watch

**decr**            keyword

usage

**decr** *name*

remarks

Used to decrease the value of *name* by 1; *name* must be the identifier of a variable integer.

see also

**incr**

Working with data

## Working with data

The T programming language supports several kinds of data; literal constants, named constants and named variables. Constants and variables must be declared before they are used. This is done with a declaration statement. You may use any of the standard data types:

```
int
real
boolean
char
string
```

or a data type you define in your program using one of these declaration key words:

```
enum
array
record
union
```

### literals

A literal integer is written as a sequence of digits. A + or - operator can optionally precede the first digit:

```
123
-46
```

A literal real number, that is, one written into your source code, begins and ends with a digit and must contain a decimal point. A + or - can precede the first digit. The following forms are valid:

```
-9.954
7.43e-4
```

These forms of real numbers are invalid:

```
.97
9.
```

A literal string is a sequence of characters between a pair of quotation marks:

```
"The rain in Spain falls mainly on the plain."
```

A literal character is a single character between a pair of apostrophes:

```
't'
```

## identifiers

Every constant and variable you declare must be identified with a *name*. The T computer programming language is case sensitive, a variable named "sum" is not the same variable as one named "Sum". The maximum length of a *name* is 64 characters. A *name* can be made from letters, digits, and the underscore character "\_" but must start with a letter.

## variable declarations

The declaration of a variable uses the key word **var** and has the following form:

```
var name { , name } : type specification [ := expression ]
```

Each *name* in the list is declared with the same *type specification* and is optionally initialized to the same *expression* value.

## constant declarations

The declaration of a constant uses the key word **const** and has this form:

```
const name : type specification := expression
```

The syntax of a constant declaration is similar to that of a variable declaration; however, only one *name* at a time is declared. A constant must be initialized when it is declared.

## type declarations

A type declaration creates a *name* for a data type which you may use elsewhere in a program to declare a variable or a constant with name as the type specification. The declaration of a data type takes this form:

```
type name : type specification
```

in which *type specification* can be one of the standard types. For example this declares a data type named `index`:

```
type index : int
```

## expressions

Expressions are used as arguments in many program statements; they are used in assignment statements, decision statements, and as arguments in subprogram calls. An *expression* returns a numerical value, a boolean value, an enumerated value, a character, or a string. They do not return entire arrays, records, or unions. An *expression* can be one of:

- a. *name*



- b. *literal constant*
- c. *expression operator expression*
- d. *operator expression*
- e. *( expression )*

Form (a) must represent a value from one of the standard data types or an enumeration. The name may represent a constant, an initialized variable, or a function. Form (b) can represent any of the standard data types. Forms (c), (d), and (e) allow evaluation of complex arithmetic and boolean expressions.

### assignments

Assignment statements have the form:

$$\textit{name} := \textit{expression}$$

The *name* on the left hand side of  $:=$  must be for a variable of standard type or a standard type item of a user defined data type. The *expression* must be compatible with *name*, i.e., both sides of the symbol  $:=$  must have the identical data type except when an integer is assigned to a real number variable.

The assignment statement is used to assign a new value to a variable. An assignment statement closely resembles an equation:

$$\text{sum} := x + y$$

In a computer program, this means that the value of the *expression*  $x + y$  is to be assigned to the memory location identified by `sum` which is its name. The assignment operator is the symbol  $:=$ . It causes the memory location identified to the left of it to be assigned the value of the *expression* to the right.

An assignment statement is not an equality. Consider a statement used frequently in repetitive computer operations:

$$x := x + 1$$

What happens to the value of `x` when this statement executes?

### numerical data

Only integers and real numbers are available in the T language. A constant number is declared as follows:

```
const i : int := 0
const pi : real := 3.14159
```

A variable number does not need to be initialized when declared; but can be:

```
var s : real
var i, j, k : int

% both are initialized
var a, b : real := 1.0
```

The following operators may be used in numerical expressions:

|            |                                |
|------------|--------------------------------|
| +          | integer or real addition       |
| -          | integer or real subtraction    |
| *          | integer or real multiplication |
| /          | real division (result is real) |
| <b>div</b> | integer quotient               |
| <b>mod</b> | integer remainder              |
| ^          | integer or real exponentiation |

In numerical expressions, the order of operations is from left to right for all but exponentiation. Exponentiation has the highest precedence; next is the group: **\*** / **mod div** and last is the group: **+** **-**. Operations within enclosing parentheses occur before operations outside.

For example, a numerical expression would be evaluated as follows:

```
4 + 9 div 2 * ( 9 - 11 mod 3 ^ 2 )
4 + 9 div 2 * ( 9 - 11 mod 9 )
4 + 9 div 2 * ( 9 - 2 )
4 + 9 div 2 * 7
4 + 4 * 7
4 + 28
32
```

A numerical expression reduces to either a real number or to an integer. An integer value may be assigned to a real variable; however, a real value may not be assigned to a variable declared as an integer. This is to prevent loss of information.

Integers may be increased or decreased by 1 with the increment and decrement operators. They only operate on integer variables. For example:

```
var i, j : int := 0

incr i    % increment i by one
decr j    % decrement j by one
```

boolean data

A Boolean variable is limited to the range of **true** or **false**. The keywords **true** and **false** are boolean constants. The following declarations are valid:

```
var flag : boolean
var result, done : boolean := false
```

The following Boolean operators are available in the T interpreter:

|             |              |
|-------------|--------------|
| <b>and</b>  | logical and  |
| <b>nand</b> | not and      |
| <b>or</b>   | or           |
| <b>nor</b>  | not or       |
| <b>xor</b>  | exclusive or |
| <b>not</b>  | invert       |

The operator **not** is a unary operator and has higher precedence than the operators **and** and **nand** which have higher precedence than **or**, **nor**, and **xor**.

Comparison operators accept integer, real, character, or string operands and return **true** if the comparison is satisfied, otherwise they return **false**:

|    |                          |
|----|--------------------------|
| =  | equal to                 |
| ~= | not equal to             |
| >  | greater than             |
| >= | greater than or equal to |
| <  | less than                |
| <= | less than or equal to    |

A comparison of two data items is a boolean factor and may be used as an operand in a *boolean expression*. A boolean value may be assigned only to a boolean variable. Boolean variables are often used in logical statements which control program execution. The following shows a boolean assignment:

```
singular := det = 0.0
```

### string data

Strings are a sequence of text characters. A string may be up to 255 characters long. The end of a string is marked by a null byte. The interpreter appends this marker automatically in many of its functions. If a program you write inserts individual characters into a string, you could inadvertently overwrite the end character with unpredictable results.

String expressions may use the concatenation operator **&** to concatenate a sequence of strings by joining the end string on the left of operator to the beginning of the string to the right.

A string expression may be assigned only to a string variable. The following program uses string assignments:

```
const wmsg : string := "Welcome to T, "  
var message : string  
var name : string  
  
program  
  
    prompt "Hi, what's your name? "  
    get name  
    message := wmsg & name & "!"  
    put message  
  
end program
```

The functions `intstr`, `realstr`, `erealstr`, and `frealstr` convert numbers into formatted strings and may be used in string expressions. Note that characters may not be concatenated into strings.

#### character data

Characters are individual text characters. They can be declared as follows:

```
var input : char  
const one : char := '1'
```

You can assign several non-text characters to strings and to character data by using a preceding backslash character:

|                     |                                |
|---------------------|--------------------------------|
| <code>\"</code>     | embedded quote                 |
| <code>\'</code>     | embedded apostrophe            |
| <code>\\</code>     | embedded backslash             |
| <code>\b, \B</code> | back space                     |
| <code>\f, \F</code> | form feed                      |
| <code>\n, \N</code> | new line                       |
| <code>\t, \T</code> | tab                            |
| <code>\0</code>     | null (end of string character) |

A character may be assigned only to a character variable. For example, this program fragment:

```
var msg : string  
  
msg[ 0 ] := 'H'  
msg[ 1 ] := 'i'  
msg[ 2 ] := '\0'
```

initializes the variable `string msg`. Note that the string is terminated by a null character. An individual character in a string may be accessed using an indexed form of the string variable name. The following statements are valid:

```
% get first character
input := name[ 0 ]

% set fifth character
msg[ 4 ] := 't'
```

The standard function `ord` accepts a character and returns an integer. Its inverse is the function `chr` which converts an integer into a character.

### enumerated data

An enumeration *type specification* is declared using the key word **enum** with the syntax:

```
type name : enum[ item{, item} ]
```

The items are valued sequentially and increasing. Example:

```
type color : enum[ red, yellow, green ]
var light : color := color.green
```

Note that enumerated items are identified using the dot operator.

```
name.item
```

### arrays of data

An array type specification is declared using the key words **array** and **of** with the syntax:

```
array[ index{, index} ] of type specification
```

Where each *index* must be an *integer expression*. Array indices are zero based. Example, for:

```
var A : array[ 2, 2 ] of real
```

valid identifiers for `A` are:

```
A[ 0, 0 ]  A[ 0, 1 ]
A[ 1, 0 ]  A[ 1, 1 ]
```

### records of data

A record type specification is declared using the key words **record** and **end** with the syntax:

```
record
    item{, item} : type specification
    {item{, item}} : type specification
end record
```

A record *item* is identified using the dot operator:

*name.item*

where *name* is the identifier of a constant or a variable. Each *item* has a distinct memory location. Example:

```
var pt : record
        x, y, z : real
end record

r := sqrt( pt.x^2 + pt.y^2 + pt.z^2 )
```

### unions of data

A union type specification is declared using the key words **union** and **end** with the syntax:

```
union
    item{, item} : type specification
    {item{, item}} : type specification
end union
```

Unlike a record declaration, the items in a union occupy the same memory location. Your program must keep track of the current type of data stored in a union. Unpredictable results can occur if you access data in a union incorrectly. Like a record, a union item is also identified using the dot operator:

*name.item*

### precedence of operators

The order of precedence determines which operations occur first in an expression; the highest is first, the lowest last. The order of precedence for all operators from highest to lowest is:

```
^
+   -   (as unary operators)
*   /   div mod
```

+       -       &  
=       ~=      <      <=      >      >=  
**not**  
**and**   **nand**  
**or**    **nor**   **xor**

**prompt**        keyword

usage

**prompt** *string expression*

remarks

Keyword is used to set the prompt string in the get dialog box which is used when entering data from the console.

example

**program**

```
var i : int

prompt "enter i: "
get i
put "i = ", i, ", i^2 = ", i * i
```

**end program**

see also

get

Some input and output



## **Source code**

The T Interpreter was developed using the C programming language and uses the Windows 3.1 Application Programming Interface. If you purchase the source code from the copyright owner, the author below, you will have a right to use, or modify the source files for the T interpreter in any way you find useful, provided that you agree that the copyright owner, the author, has no warranty, obligations or liability for any of the source files for the T interpreter.

To order the source code please send \$150 US to the author:

Stephen R. Schmitt  
962 Depot Road  
Boxborough  
MA 01719

specify either 3.5 inch or 5.25 inch floppy diskette. These will be forwarded to you within 60 days. The disks will be replaced for free if defective.

## Using subprograms

It is almost always necessary to use subprogram modules so that your programs are easy to understand and maintain. There are two distinct types of subprogram modules. A procedure is a statement by itself. A function returns a value for use in expression evaluation.

### subprogram calls

A call to a subprogram has the form:

*name* [ (*argument* { , *argument* } ) ]

Program execution jumps to the subprogram declaration. The call passes each *argument* to the subprogram. Upon completion of the statement list in a subprogram's body, program execution returns to the point immediately after the call.

An example:

```
x := square( 7 )
```

### subprogram arguments

The *arguments* used in a subprogram call must be compatible with the *parameters* defined in a subprogram declaration. Arguments are passed to a subprogram either by value or by reference. Arguments passed by value cannot be changed by the subprogram. This means that a variable used as an argument will have the same value before and after the subprogram call it was used in. When an argument is passed by reference, the address of the argument is given to the subprogram. In this case, a variable used as an argument may have a different value before and after the subprogram call.

All standard data types can be passed by value. However, data structures, arrays, records, and unions, cannot be passed by value to a subprogram. For example, if you need to perform an operation on an array, you can pass the address of the entire array to a subprogram by reference.

A *parameter* list is a list of variable declarations used in the subprogram. It has this form:

[ **var** ] *name* { , *name* } : *type specification*

The key word **var** is used in a subprogram header to declare that each *name* in a *parameter* list is passed by reference. Its omission means that each *name* in a *parameter* list is passed by value.

### return statements

Procedures may optionally contain a return statement of the form:

## **return**

Functions, however, must contain at least one return statement having the form:

**return** *expression*

The *expression's* type must be compatible with the function's return type. The action of a return statement is always immediate. A subprogram may contain more than one return statement.

## procedure declarations

The declaration of a procedure takes the following form:

```
procedure name [ (parameter{, parameter}) ]  
    declarations and statements  
end procedure
```

Declarations of variables or constants within the procedure body are only visible within the procedure.

A procedure is a program statement. Program execution will resume at the next statement after a procedure call. Program execution returns from a procedure upon reaching the end of the procedure's statement list or by the action of a return statement anywhere in the body of the procedure.

An example of a procedure declaration:

```
procedure put_square( value : real )  
    put value*value  
end procedure
```

## function declarations

The declaration of a function is similar to that of a procedure:

```
function name [ (parameter{, parameter}) ] : type specification  
    declarations and statements  
end function
```

The differences are that a return type must be specified after the list of parameters as shown above and that a function must return a value using a return statement.

Declarations of variables or constants within the function body are only visible within the function.

Functions are used in expressions. Program execution returns to the point in the expression after a function call. Program execution returns from a function upon reaching a return

statement anywhere in the body of the function.

An example of a function declaration:

```
function square( value : real ) : real  
    return value^2  
end function
```

The T language includes the following standard functions and procedures to help you write useful programs:

### mathematical functions

|                  |   |
|------------------|---|
| <u>arccos</u>    | arc cosine                                  |
| <u>arcsin</u>    | arc sine                                    |
| <u>arctan</u>    | arc tangent                                 |
| <u>arctanxy</u>  | arc tangent of Cartesian coordinates        |
| <u>ceil</u>      | real to integer above                       |
| <u>cos</u>       | cosine                                      |
| <u>cosh</u>      | hyperbolic cosine                           |
| <u>exp</u>       | power of natural logarithm base $e$         |
| <u>floor</u>     | real to integer below                       |
| <u>getexp</u>    | exponent base 10 of argument                |
| <u>ln</u>        | natural (base $e$ ) logarithm               |
| <u>log10</u>     | base 10 logarithm                           |
| <u>rand</u>      | real random number in range 0.0 to 1.0      |
| <u>randint</u>   | integer random number in range of arguments |
| <u>randomize</u> | changes seed of random number generator     |
| <u>randseed</u>  | set random seed                             |
| <u>round</u>     | real to nearest integer                     |
| <u>setexp</u>    | set exponent base 10 to a new value         |
| <u>sign</u>      | integer sign (+/-1) of real                 |
| <u>sin</u>       | sine  |
| <u>sinh</u>      | hyperbolic sine                             |
| <u>sqrt</u>      | square root                                 |
| <u>tan</u>       | tangent                                     |
| <u>tanh</u>      | hyperbolic tangent                          |

### string and character functions

Functions in this group perform operations on strings and characters.

|                 |                                       |
|-----------------|---------------------------------------|
| <u>chr</u>      | integer to character                  |
| <u>erealstr</u> | real to string, exponent format       |
| <u>frealstr</u> | real to string, floating point format |

|                |                                 |
|----------------|---------------------------------|
| <u>index</u>   | location of sub string          |
| <u>intstr</u>  | integer to string               |
| <u>length</u>  | length of string                |
| <u>ord</u>     | character to integer            |
| <u>realstr</u> | real to string, default formats |
| <u>repeat</u>  | repeated sub strings            |
| <u>strint</u>  | string to integer               |
| <u>strreal</u> | string to real number           |

#### file system access

These functions provide access to hard and floppy disk files.

|              |   |
|--------------|---|
| <u>close</u> | closes an open disk file                    |
| <u>eof</u>   | indicates when the end of a file is reached |
| <u>open</u>  | opens a disk file                           |

## Language reference

This topic contains descriptions of key words, special symbols, standard functions, and standard procedures used in the T programming language.

### conventions

Bracketed [*item*] items are optional. Items in braces {*item*} are optional and may be repeated. Italicized *items* are elements of code determined by the programmer. A bar | means that either the word on the right or the word on the left is applicable.

### tables

special symbols

limits

### definitions

|                          |                   |
|--------------------------|-------------------|
| <b><u>and</u></b>        | keyword           |
| <u>arccos</u>            | standard function |
| <u>arcsin</u>            | standard function |
| <u>arctan</u>            | standard function |
| <u>arctanxy</u>          | standard function |
| <b><u>array</u></b>      | keyword           |
| <b><u>assert</u></b>     | keyword           |
| <b><u>boolean</u></b>    | keyword           |
| <b><u>break</u></b>      | keyword           |
| <b><u>case</u></b>       | keyword           |
| <u>ceil</u>              | standard function |
| <b><u>char</u></b>       | keyword           |
| <u>chr</u>               | standard function |
| <u>close</u>             | standard function |
| <b><u>const</u></b>      | keyword           |
| <b><u>continue</u></b>   | keyword           |
| <u>cos</u>               | standard function |
| <u>cosh</u>              | standard function |
| <b><u>decr</u></b>       | keyword           |
| <b><u>decreasing</u></b> | keyword           |
| <b><u>div</u></b>        | keyword           |
| <b><u>do</u></b>         | keyword           |
| <b><u>else</u></b>       | keyword           |
| <b><u>elsif</u></b>      | keyword           |
| <b><u>end</u></b>        | keyword           |
| <b><u>enum</u></b>       | keyword           |
| <u>eof</u>               | standard function |
| <u>erealstr</u>          | standard function |
| <b><u>exit</u></b>       | keyword           |

|                         |                    |
|-------------------------|--------------------|
| <u>exp</u>              | standard function  |
| <b><u>false</u></b>     | keyword            |
| <u>floor</u>            | standard function  |
| <b><u>for</u></b>       | keyword            |
| <u>frealstr</u>         | standard function  |
| <b><u>function</u></b>  | keyword            |
| <b><u>get</u></b>       | keyword            |
| <u>getexp</u>           | standard function  |
| <b><u>goto</u></b>      | keyword            |
| <b><u>if</u></b>        | keyword            |
| <b><u>incr</u></b>      | keyword            |
| <u>index</u>            | standard function  |
| <b><u>int</u></b>       | keyword            |
| <u>intstr</u>           | standard function  |
| <b><u>label</u></b>     | keyword            |
| <u>length</u>           | standard function  |
| <u>ln</u>               | standard function  |
| <u>log10</u>            | standard function  |
| <b><u>loop</u></b>      | keyword            |
| <b><u>mod</u></b>       | keyword            |
| <b><u>nand</u></b>      | keyword            |
| <b><u>nor</u></b>       | keyword            |
| <b><u>not</u></b>       | keyword            |
| <b><u>of</u></b>        | keyword            |
| <u>open</u>             | standard function  |
| <b><u>or</u></b>        | keyword            |
| <u>ord</u>              | standard function  |
| <b><u>procedure</u></b> | keyword            |
| <b><u>program</u></b>   | keyword            |
| <b><u>prompt</u></b>    | keyword            |
| <b><u>put</u></b>       | keyword            |
| <u>rand</u>             | standard function  |
| <u>randint</u>          | standard function  |
| <u>randomize</u>        | standard procedure |
| <u>randseed</u>         | standard procedure |
| <b><u>real</u></b>      | keyword            |
| <u>realstr</u>          | standard function  |
| <b><u>record</u></b>    | keyword            |
| <u>repeat</u>           | standard function  |
| <b><u>return</u></b>    | keyword            |
| <u>round</u>            | standard function  |
| <u>setexp</u>           | standard function  |
| <u>sign</u>             | standard function  |
| <u>sin</u>              | standard function  |
| <u>sinh</u>             | standard function  |
| <u>sqrt</u>             | standard function  |
| <b><u>string</u></b>    | keyword            |

|                |                    |
|----------------|--------------------|
| <u>strint</u>  | standard function  |
| <u>strreal</u> | standard function  |
| <u>tan</u>     | standard function  |
| <u>tanh</u>    | standard function  |
| <u>then</u>    | keyword            |
| <u>true</u>    | keyword            |
| <u>type</u>    | keyword            |
| <u>union</u>   | keyword            |
| <u>value</u>   | keyword            |
| <u>var</u>     | keyword            |
| <u>watch</u>   | standard procedure |
| <u>when</u>    | keyword            |
| <u>xor</u>     | keyword            |



**incr**            keyword

usage

**incr** *name*

remarks

Used to increase the value of *name* by 1; *name* must be the identifier of a variable integer.

see also

**decr**

Working with data

## Some input and output

Input and output is provided by means of put and get statements to the output window and to disk files.

### put statements

The complete definition of the put statement is:

```
put [ :stream, ] put item{ , put item } [ . . . ]
```

It is used for output of text data to files or the video display of your console. The value of *stream* must match an integer value obtained with the standard function `open`. If *stream* is omitted, the output is sent to the console for video display. A *put item* has the form:

```
expression [ :width [ :fraction width [ :exponent width ] ] ]
```

The *expression* can be of any standard type except boolean. The value of *width* is the total number of characters in the put item. Strings are left justified; numbers are right justified. The *fraction width* and *exponent width* options are for writing a number in a real number format. If a specified format is too small, the actual format width is increased to accommodate the item.

The optional ellipses ". . ." symbol inhibits adding a new line after the last put item. Some examples:

```
put "hi":8  
put 0.001:12:4:2  
put 99:4  
  
const pi : real := 3.1415926535  
  
put pi  
put pi:12  
put pi:16:8:2
```

### prompt statements

The prompt statement may be used to set a global prompt message. Once set the message is displayed each time a get statement is used for console input. It has the form:

```
prompt string expression
```

### get statements

The complete definition of a get statement is:

```
get [:stream,] get item{, get item}
```

It is used for input of text data from files or the console. The value of *stream* must match an integer value obtained with the standard function `open`. If *stream* is omitted, the input obtained by keyboard entry. A *get item* is one of:

- a. *name*
- b. *name* : \*
- c. *name* : *width*

Form (a) is used for token input; the root type of the *get item*'s identifier can be integer, real, or string. This form skips white space until an initial character indicates the start of a token. Form (b) is used for line input and reads up to an end of line symbol. Form (c) reads *width* characters. The identifiers in forms (b) and (c) can only be string type. Some examples:

```
get your_name  
get characters : 8
```

### file access functions

The standard functions `open`, `close`, and `eof` provide access to files on disk. Files may be opened to read from or to write to. The following program fragment shows how these standard functions may be used in a program:

```
var file : int  
var filename : string := "a_file.txt"  
  
file := open( filename, "r" )  
if file = 0 then  
    put "file not found: ", filename  
else  
    loop  
        exit when eof( file )  
        get :file, buffer : *  
        put buffer  
        buffer := ""  
    end loop  
    if close( file ) = 0 then  
        put "file close error"  
    end if  
end if
```

`arccos`      standard function

usage

`arccos( expression : real ) : real`

remarks

Function returns the real arc cosine of *expression* in units of radians. The value of *expression* must be in the range -1.0 to +1.0 or a run-time error will occur.

example

```
% return arc secant
function arcsec( x : real ) : real

    var r : real

    if x >= 1.0 then
        r := arccos( 1 / x )
    elsif x <= -1.0 then
        r := -arccos( 1 / x )
    else
        r := 0.0
    end if

    return r

end function
```

see also

arcsin

arctan

arctanxy

Using subprograms

arcsin        standard function

usage

arcsin( *expression* : **real** ) : **real**

remarks

Function returns the real arc sine of *expression* in units of radians. The value of *expression* must be in the range -1.0 to +1.0 or a run-time error will occur.

example

```
const Pi : real := 2 * arcsin( 1 )
```

```
% return arc cosecant
```

```
function arccsc( x : real ) : real
```

```
    var r : real
```

```
    if x >= 1.0 then
```

```
        r := arcsin( 1 / x )
```

```
    elsif x <= -1.0 then
```

```
        r := -Pi - arcsin( 1 / x )
```

```
    else
```

```
        r := 0.0
```

```
    end if
```

```
    return r
```

```
end function
```

see also

arccos

arctan

arctanxy

Using subprograms

## **Window menu commands**

Window management commands for the multiple document editor.

### **Cascade**

Keyboard command: Alt+W C

Hot key: Shift+F5

Arranges the open source files into a cascade.

### **Tile horizontal**

Keyboard command: Alt+W H

Hot key: Shift+F4

Arranges the open source files into horizontal tiles if space permits.

### **Tile vertical**

Keyboard command: Alt+W T

Arranges the open source files into vertical tiles if space permits.

### **Arrange icons**

Keyboard command: Alt+W I

Arranges the icons of open source files into regularly spaced rows.

### **Switch**

Keyboard command: Alt+W S

Hot key: Ctrl+F6

Switches focus from one open source file to another open source file.

### **Close all**

Keyboard command: Alt+W A

Closes all of the open source files.

**and**            keyword

usage

*boolean expression* **and** *boolean expression*

remarks

Operator returns a boolean value:

| x | y | x <b>and</b> y |
|---|---|----------------|
|---|---|----------------|

|              |              |              |
|--------------|--------------|--------------|
| <b>false</b> | <b>false</b> | <b>false</b> |
|--------------|--------------|--------------|

|              |             |              |
|--------------|-------------|--------------|
| <b>false</b> | <b>true</b> | <b>false</b> |
|--------------|-------------|--------------|

|             |              |              |
|-------------|--------------|--------------|
| <b>true</b> | <b>false</b> | <b>false</b> |
|-------------|--------------|--------------|

|             |             |             |
|-------------|-------------|-------------|
| <b>true</b> | <b>true</b> | <b>true</b> |
|-------------|-------------|-------------|

see also

Working with data

arctan        standard function

usage

arctan( *expression* : **real** ) : **real**

remarks

Function returns the real arc tangent of *expression* in units of radians in the range of  $-\pi/2$  to  $\pi/2$ .

example

```
const Pi : real := 2 * arcsin( 1 )
```

```
% calculate hyperbolic <-> circular parameter
```

```
function gudermannian( x : real ) : real
```

```
    var r : real
```

```
    r := 2 * arctan( exp( x ) ) - Pi / 2
```

```
    return r
```

```
end function
```

see also

arctanxy

arcsin

arccos

Using subprograms



arctanxy     standard function

usage

```
arctanxy( x : real, y : real ) : real
```

remarks

Function returns the real arc tangent of  $y/x$  in units of radians in the range of  $-\pi$  to  $\pi$ . If both  $x$  and  $y$  are 0.0 a run-time error will occur.

example

```
const Pi : real := 2 * arcsin( 1 )
```

```
% return heading in degrees
```

```
function heading( e, n : real ) : real
```

```
    var hdg : real
```

```
    hdg := 90 - 180 * arctanxy( e, n ) / Pi
```

```
    if hdg < 0.0 then
```

```
        hdg := hdg + 360
```

```
    end if
```

```
    return hdg
```

```
end function
```

see also

arctan

arcsin

arccos

Using subprograms

**array**        keyword

usage

**array**[ *size*{, *size* } ] **of** *type specification*

remarks

Keyword is used for specifying a data type as an array of *type specification*. Array indices, *size*, must be constant integer expressions.

see also

limits

Working with data

**assert**      keyword

usage

**assert** *boolean expression*

remarks

Keyword is used to conditionally continue execution of a program.  
If *boolean expression* is false the program halts.

see also

Getting started

**boolean**     keyword

usage

**var** *name* : **boolean**

remarks

Standard data type specifier. Boolean data can have a value of either **true** or **false**.

see also

Working with data

**case**            keyword

usage

**case** *expression of*

**value** *constant*{, *constant*} :  
        *declarations and statements*

{**value** *constant*{, *constant*} :  
    *declarations and statements*}

[**value** :  
    *declarations and statements*]

**end case**

remarks

The *expression* and each *constant* must be of matching types of **int**, **string**, **char**, or **enum**. One **value** not having a *constant* may be placed at the end of the sequence of case values as a default branch.

see also

Looping and jumping

`ceil`            [standard function](#)

[usage](#)

```
ceil( expression : real ) : int
```

[remarks](#)

Function returns the smallest integer greater than or equal to *expression*.

[example](#)

```
% find absolute ceiling of number
function abs_ceil( x : real ) : int

    var r : int

    if x >= 0.0 then
        r := ceil( x )
    else
        r := floor( x )
    end if

    return r

end function
```

[see also](#)

[floor](#)

[round](#)

[sign](#)

[Using subprograms](#)

**char**            keyword

usage

**const** *name* : **char** := '*literal character*'

**var** *name* : **char**

remarks

Standard data type specifier for characters.

see also

Working with data

chr            standard function

usage

chr( *expression* : **int** ) : **char**

remarks

Function returns a character corresponding to the integer value of *expression*.

example

**procedure** list\_characters

```
    var i, j, n : int

    for i := 2...7 do
        for j := 0...15 do
            n := i * 16 + j
            put n, " - ", chr( n )
        end for
    end for
```

**end procedure**

see also

ord

Using subprograms



close            standard function

usage

```
close( stream : int ) : int
```

remarks

Function closes the file associated with *stream*. Returns *stream* on success or else 0.

example

```
% copy text files
function copy( d : string,
               s : string ) : boolean

    var df, sf : int
    var line : string

    sf := open( s, "r" )
    df := open( d, "w" )

    if sf = 0 or
        df = 0 then
        return false
    end if

    loop
        exit when eof( sf )
        get : sf, line : *
        put : df, line
    end loop

    if close( sf ) = 0 or
        close( df ) = 0 then
        put "file close error"
        return false
    else
        return true
    end if

end function
```

see also

eof

open

Some input and output

Using subprograms

**const**      keyword

usage

**const** *name* : *type specification* := *constant expression*

remarks

Keyword is used to declare a constant. The *constant expression* may not include any names of variables.

see also

Working with data

**continue**    keyword

usage

**continue** [ **when** *boolean expression* ]

remarks

Used to jump to the start of the nearest enclosing **for** or **loop** statement. Jump is immediate unless the optional **when** condition is included.

see also

Looping and jumping

cos            standard function

usage

cos( *expression* : **real** ) : **real**

remarks

Function returns the cosine of *expression*. The value of *expression* is assumed to be in units of radians.

example

```
% return secant
function sec( x : real ) : real

    return 1 / cos( x )

end function
```

see also

sin

tan

Using subprograms

cosh            standard function

usage

```
cosh( expression : real ) : real
```

remarks

Function returns the hyperbolic cosine of *expression*. The value of *expression* is assumed to be in units of radians.

example

```
% return hyperbolic secant  
function sech( x : real ) : real
```

```
    return 1 / cosh( x )
```

```
end function
```

see also

sinh

tanh

Using subprograms

**decreasing** keyword

usage

**for decreasing** *name* := *begin...end* **do**

remarks

The keyword indicates that the **for** loop counter decrements by one on each repeat of the loop.

see also

**for**

Looping and jumping

**div**            keyword

usage

*integer expression* **div** *integer expression*

remarks

Operator returns the quotient for integer division. The result type is integer.

see also

**mod**

Working with data



**do**            keyword

usage

**for** *name* := *begin...end* **do**

see also

**for**

Looping and jumping

**elsif**      keyword

usage

**elsif** *boolean expression* **then**  
    *declarations and statements*

see also

**if**  
Looping and jumping

**else**            keyword

usage

**else**  
      *declarations and statements*  
**end if**

see also

**if**  
Looping and jumping

**end**            keyword

usage

**end loop**  
**end for**  
**end if**  
**end case**  
**end function**  
**end program**  
**end procedure**  
**end record**  
**end union**

remarks

Used to mark the end of logic statements, data structure definitions, and subprograms.

see also

Getting started  
Working with data  
Looping and jumping  
Using subprograms

eof            standard function

usage

eof( *stream* : int ) : boolean

remarks

Function returns **true** if the end of the file corresponding to *stream* has been reached. The value of *stream* is normally obtained using the "open" function.

example

```
% copy text files
function copy( d : string,
               s : string ) : boolean

    var df, sf : int
    var line : string

    sf := open( s, "r" )
    df := open( d, "w" )

    if sf = 0 or
        df = 0 then
        return false
    end if

    loop
        exit when eof( sf )
        get : sf, line : *
        put : df, line
    end loop

    if close( sf ) = 0 or
        close( df ) = 0 then
        put "file close error"
        return false
    else
        return true
    end if

end function
```

see also

close

open

Some input and output

Using subprograms

**enum**            keyword

usage

**type** *name* : **enum**[ *item* {, *item* } ]

remarks

Used to define an enumerated data type. The value of each *item* increases to the right. Values are accessed using the form:

*name.item*

see also

Working with data

`erealstr`     standard function

usage

```
erealstr( expression : real,  
          format width : int,  
          fraction width : int,  
          exponent width : int ) : string
```

remarks

Function returns a string of the form:

`{blank}[-]digit.{digit} e sign digit{digit}`

corresponding to *expression*. Widths are increased automatically if necessary.

example

```
const Pi : real := 2 * arcsin( 1 )
```

```
procedure put_area( r : real )
```

```
    var a : real  
    var line : string  
  
    a := Pi * r^2  
    line := "area = " &  
            erealstr( a, 24, 12, 3 )  
    put line
```

```
end procedure
```

see also

frealstr

realstr

intstr

Using subprograms



**exit**            keyword

usage

**exit** [ **when** *boolean expression* ]

remarks

Used to exit from the nearest enclosing **for** or **loop** statement.  
Exit is immediate unless the optional **when** condition is included.

see also

Looping and jumping

exp            standard function

usage

exp( *expression* : **real** ) : **real**

remarks

Function returns the natural logarithm base  $e$  raised to the power of *expression*.

example

```
% return probability of Poisson pdf
function poisson( x : int, m : real ) : real

    var f : int := 1
    var r : real

    assert x >= 0
    assert m > 0

    r := m^x * exp( -m )

    loop
        exit when x = 0
        f := f * x
        decr x
    end loop

    r := r / f

    return r

end function
```

see also

ln

Using subprograms

**false**      keyword

usage

*name* := **false**

remarks

Boolean constant; opposite of **true**.

see also

Working with data

floor            standard function

usage

floor( *expression* : **real** ) : **int**

remarks

Function returns the largest integer less than or equal to *expression*.

example

```
% find absolute floor of number
function abs_floor( x : real ) : int

    var r : int

    if x >= 0.0 then
        r := floor( x )
    else
        r := ceil( x )
    end if

    return r

end function
```

see also

ceil

round

sign

Using subprograms

**for**            keyword

usage

```
for [decreasing] name := begin...end do  
    declarations and statements  
end for
```

remarks

The for statement repeats the list of *declarations and statements* for each value in the range *begin...end*. The identifier *name* must be declared as an integer outside the loop. The value of *name* is incremented, or decremented if the optional keyword **decreasing** is used, before repeating the loop. The **continue** and **exit** statements can be used for control within the loop. Declarations made within the loop are not visible outside the loop.

see also

Looping and jumping

frealstr     standard function

usage

```
frealstr( expression : real,  
          format width : int,  
          fraction width : int ) : string
```

remarks

Function returns a string of the form:

*{blank}[-]digit{digit}.{digit}*

corresponding to *expression*. Blanks are added as needed to right justify the string. Widths are increased automatically if necessary.

example

```
const Pi : real := 2 * arcsin( 1 )  
  
procedure put_circumference( r : real )  
  
    var c : real  
    var line : string  
  
    c := 2 * Pi * r  
    line := "circumference = " &  
            frealstr( c, 24, 12 )  
    put line  
  
end procedure
```

see also

erealstr

realstr

intstr

Using subprograms

**function**    keyword

usage

**function** *name* [(*param*{, *param*)}] : *type specification*  
    *declarations and statements*  
**end function**

in which *param* is:

    [**var**] *name*{, *name*} : *type specification*

remarks

A function must return a value using a **return** statement. Declarations within the function definition are only visible within the function. The use of **var** in a parameter list means that the parameter is to be passed to the function by reference rather than by value.

see also

Using subprograms

**get**            keyword

usage

**get** [:*stream*,] *get item*{, *get item*}

in which a *get item* is one of:

- a. *name*
- b. *name* : \*
- c. *name* : *width*

remarks

Each *get item* read sequentially from a file identified by *stream*. If *stream* is omitted, input is from your console's keyboard.

The *name* of *get item* must correspond to a declared variable. Form (a) can be an integer, real number, or a string. Form (b) reads input until an end of line character is found, *name* must be of a string. Form (c) reads *width* characters and *name* must also be of a string.

see also

put

open

close

Some input and output



getexp        standard function

usage

getexp( *expression* : **real** ) : **int**

remarks

Function returns the exponent, base 10, of *expression*. If *expression* equals 0.0, zero is returned.

example

```
type bignum : record
      m : real        % mantissa
      x : int         % exponent
end record
```

```
% print a big number
```

```
procedure put_bignum( var s : bignum )
```

```
    put s.m, " x 10^", s.x
```

```
end procedure
```

```
% divide two big numbers
```

```
% dest <- dest / srce
```

```
procedure divide( var d, s : bignum )
```

```
    var dx : int
```

```
    d.m := d.m / s.m
```

```
    d.x := d.x - s.x
```

```
    dx := getexp( d.m )
```

```
    if dx ~= 0 then
```

```
        d.x := d.x + dx
```

```
        d.m := setexp( d.m, 0 )
```

```
    end if
```

```
end procedure
```

see also

setexp

Using subprograms

**goto**            keyword

usage

**goto** *label name*

remarks

This keyword causes an immediate jump to the location of *label name*. Program execution may not jump from one subprogram to another.

see also

**label**

Looping and jumping

**if**            keyword

usage

```
if boolean expression then  
    declarations and statements  
{elsif boolean expression then  
    declarations and statements }  
[else  
    declarations and statements]  
end if
```

remarks

The *declarations and statements* are executed in the first branch in which the *boolean expression* is true. Optional **elsif** branches must be placed ahead of the single optional **else** branch. Declarations within each branch are not visible outside the branch.

see also

[Looping and jumping](#)

`index`            [standard function](#)

[usage](#)

```
index( string, pattern : string ) : int
```

[remarks](#)

Function returns the value of the location of the first occurrence of *pattern* in *string*. If no match is found, a negative number is returned.

[example](#)

**program**

```
var s : string := "The rain in Spain"  
var i : int  
  
i := index( s, "Spain" )  
put i
```

**end program**

[see also](#)

[length](#)

[Using subprograms](#)

**int**            keyword

usage

**var** *name* : **int**

**const** *name* : **int** := *integer expression*

remarks

Standard data type specifier for integer data.

see also

limits

Working with data

intstr        standard function

usage

intstr( *expression*, *format width* : int ) : string

remarks

Function returns a string of form:

{*blank*}[*-*]*digit*{*digit*}

corresponding to *expression*. Blanks are added as needed to right justify the string. The actual width is increased automatically if *format width* is too small.

example

**procedure** fibonacci\_numbers

**var** s : string

**var** f0, f1, f2 : int

    f0 := 1

    f1 := 1

    s := intstr( f0, 4 ) & intstr( f1, 4 )

**loop**

**exit when** f2 > 100

        f2 := f1 + f0

        s := s & intstr( f2, 4 )

        f0 := f1

        f1 := f2

**end loop**

**put** s

**end procedure**

see also

erealstr

frealstr

realstr

Using subprograms





**label**        keyword

usage

**label** *name* :

remarks

This keyword is used to declare a marker for a **goto** statement.

see also

**goto**

Looping and jumping

length        standard function

usage

length( *expression* : **string** ) : **int**

remarks

Function returns the actual number of characters in *expression*.

example

**program**

```
var s : string := "The rain in Spain"  
var i : int
```

```
  i := length( s )  
  put i
```

**end program**

see also

index

Using subprograms

ln                    standard function

usage

`ln( expression : real ) : real`

remarks

Function returns the natural logarithm of *expression* which must have a value greater than zero or a run-time error will occur.

example

```
% inverse hyperbolic sine
function inv_sinh( x : real ) : real

    var r : real

    r := ln( x + sqrt( x*x + 1 ) )

    return r

end function
```

see also

exp

log10

Using subprograms

log10        standard function

usage

log10( *expression* : **real** ) : **real**

remarks

Function returns the base 10 logarithm of *expression* which must have a value greater than zero or a run-time error will occur.

example

```
% logarithm with error handler
function log_base_10( x : real ) : real

    var r : real := 0.0

    if x > 0 then
        r := log10( x )
    end if

    return r

end function
```

see also

ln  
Using subprograms

**loop**            keyword

usage

**loop**

*declarations and statements*

**end loop**

remarks

This keyword marks the beginning and end of an infinite loop. Declarations within the loop are not visible outside the loop. Statements in the loop are executed until terminated by an **exit** statement. A **continue** statement may also be used for control within the loop.

see also

Looping and jumping

## **Project menu commands**

These commands are for operating the T interpreter.

### **Run**

Keyboard command: Alt+P R

Hot key: F9

Run the current project. If this command is selected after stepping or tracing, your program will run to completion.

### **Step over**

Keyboard command: Alt+P S

Hot key: F8

Allows you to step through a program without entering subprograms. Closed source files will be opened automatically as needed.

### **Trace into**

Keyboard command: Alt+P T

Hot key: F7

Allows you to step through a program and jump into subprograms. Closed source files will be opened automatically as needed.

### **Halt**

Keyboard command: Alt+P H

Allows you to halt a program which you are stepping or tracing through.

### **Load project...**

Keyboard command: Alt+P L

Loads the file containing the list of source files which make up your program. This will enable run, step, or trace operations.

### **Close project**

Keyboard command: Alt+P C

This command will remove the current project and disable run, step, and trace operations.

**mod**            keyword

usage

*integer expression* **mod** *integer expression*

remarks

Operator returns the remainder for integer division. The result is an integer.

see also

**div**

Working with data

**nand**            keyword

usage

*boolean expression* **nand** *boolean expression*

remarks

Operator returns a boolean value:

| x | y | x <b>nand</b> y |
|---|---|-----------------|
|---|---|-----------------|

|              |              |              |
|--------------|--------------|--------------|
| <b>false</b> | <b>false</b> | <b>true</b>  |
| <b>false</b> | <b>true</b>  | <b>true</b>  |
| <b>true</b>  | <b>false</b> | <b>true</b>  |
| <b>true</b>  | <b>true</b>  | <b>false</b> |

see also

Working with data



**nor**            keyword

usage

*boolean expression* **nor** *boolean expression*

remarks

Operator returns a boolean value:

| x | y | x <b>nor</b> y |
|---|---|----------------|
|---|---|----------------|

|              |              |             |
|--------------|--------------|-------------|
| <b>false</b> | <b>false</b> | <b>true</b> |
|--------------|--------------|-------------|

|              |             |              |
|--------------|-------------|--------------|
| <b>false</b> | <b>true</b> | <b>false</b> |
|--------------|-------------|--------------|

|             |              |              |
|-------------|--------------|--------------|
| <b>true</b> | <b>false</b> | <b>false</b> |
|-------------|--------------|--------------|

|             |             |              |
|-------------|-------------|--------------|
| <b>true</b> | <b>true</b> | <b>false</b> |
|-------------|-------------|--------------|

see also

Working with data

**not**            keyword

usage

**not** *boolean expression*

remarks

Operator returns a boolean value:

x                **not** x

**false**        **true**  
**true**         **false**

see also

Working with data

**of**            keyword

usage

**array**[ *size*{, *size*} ] **of** *type specification*  
**case** *expression of*

see also

**array**

**case**

Working with data

Looping and jumping

open            standard function

usage

```
open( filename, mode : string ) : int
```

remarks

Function opens a file for reading or writing and returns the file's stream number. The *mode* is either of:

"r" for sequentially reading from, or  
"w" for sequentially writing to.

If the file cannot be opened, zero is returned.

example

```
% copy text files
function copy( d : string,
               s : string ) : boolean

    var df, sf : int
    var line : string

    sf := open( s, "r" )
    df := open( d, "w" )

    if sf = 0 or
        df = 0 then
        return false
    end if

    loop
        exit when eof( sf )
        get : sf, line : *
        put : df, line
    end loop

    if close( sf ) = 0 or
        close( df ) = 0 then
        put "file close error"
        return false
    else
        return true
    end if
```

**end function**

see also

close

eof

Some input and output

Using subprograms

**or**            keyword

usage

*boolean expression* **or** *boolean expression*

remarks

Operator returns a boolean value:

| x | y | x <b>or</b> y |
|---|---|---------------|
|---|---|---------------|

|              |              |              |
|--------------|--------------|--------------|
| <b>false</b> | <b>false</b> | <b>false</b> |
|--------------|--------------|--------------|

|              |             |             |
|--------------|-------------|-------------|
| <b>false</b> | <b>true</b> | <b>true</b> |
|--------------|-------------|-------------|

|             |              |             |
|-------------|--------------|-------------|
| <b>true</b> | <b>false</b> | <b>true</b> |
|-------------|--------------|-------------|

|             |             |             |
|-------------|-------------|-------------|
| <b>true</b> | <b>true</b> | <b>true</b> |
|-------------|-------------|-------------|

see also

Working with data

ord            standard function

usage

ord( *expression* : **char** ) : **int**

remarks

Function accepts a character and returns its corresponding integer value.

example

% compare two strings

**function** strcmp( s1, s2 : **string** ) : **int**

**var** i : **int** := 0

**var** d : **int**

**loop**

        d := ord( s1[i] ) - ord( s2[i] )

**exit when** d ~= 0

**exit when** s1[i] = '\0'

**exit when** s2[i] = '\0'

**exit when** i >= 255

**incr** i

**end loop**

**return** d

**end function**

see also

chr

Using subprograms

**procedure** keyword

usage

**procedure** *name* [(*param*{, *param*})]

*declarations and statements*

**end procedure**

in which *param* is:

[**var**] *name*{, *name*} : *type specification*

remarks

A procedure may return after reaching the end of the list of its statements or when a **return** statement is reached. Declarations within the procedure definition are only visible within it. The use of **var** in a parameter list means that the parameter is to be passed by reference.

see also

Using subprograms



**program**      keyword

usage

**program**

*declarations and statements*

**end program**

remarks

The program statement defines the start and end of every program. Statements can call functions or procedures which are subprograms. Declarations are only visible within the program statement.

see also

Getting started

## limits

|                                    |               |
|------------------------------------|---------------|
| maximum value of an integer        | +2147483647   |
| minimum value of an integer        | -2147483648   |
| maximum magnitude of a real number | 1.797693e+308 |
| minimum magnitude of a real number | 2.225074e-308 |
| maximum value of base 10 exponent  | +308          |
| minimum value of base 10 exponent  | -307          |
| maximum string length in bytes     | 255           |
| maximum array size in bytes        | 32767         |

see also

Working with data

**put**            keyword

usage

**put** [:*stream*,] *put item*{, *put item*}[...]

in which a *put item* is:

*expression* [ :*format width* [ :*fraction width* [ :*exponent width*] ] ]

remarks

Each *put item* is written sequentially to a file identified by *stream*. If *stream* is omitted, output is to the text output window on your console's video display. A new line is started at the end of the list of *put items* unless the ellipsis symbol "... " is appended.

A global file pointer is set when *stream* is included in the **put** statement. If a *put item* uses a function call, the function should not use a different *stream* than the **put** statement.

see also

**get**

close

open

Some input and output

## **Search menu commands**

### **Find...**

Keyboard command: Alt+S F

Searches for character strings in the active file. Search is case sensitive. You can search forward or backward from the insertion point.

### **Replace...**

Keyboard command: Alt+S R

Searches for character strings in the active file and replaces each occurrence with a new string. Search is case sensitive. You can search forward or backward from the insertion point.

### **Next find**

Keyboard command: Alt+S N

Hot key: F3

Repeats the last search or search and replace operation without opening the Find dialog box.

rand            standard function

usage

rand : **real**

remarks

Function returns the next value of a sequence of pseudo random real numbers approximating a uniform distribution within the range 0.0 to 1.0.

example

% generate a normal random variable

**function** normal( mu, sig : **real** ) : **real**

**var** r, x : **real**

    r := sig \* sqrt( -2 \* ln( rand ) )

    x := r \* sin( 2 \* 3.14159 \* rand ) + mu

**return** x

**end function**

see also

randint

randomize

randseed

Using subprograms

randint      standard function

usage

randint( *low*, *high* : int ) : int

remarks

Function returns the next value of a sequence of pseudo random integers approximating a uniform distribution in the range *low* to *high*.

example

```
type pick : record
    b1, b2, b3, b4 : int
end record
```

```
procedure lotto( var d : pick )
```

```
    d.b1 := randint( 1, 16 )
```

```
    loop
```

```
        d.b2 := randint( 1, 16 )
```

```
        exit when d.b1 ~= d.b2
```

```
    end loop
```

```
    loop
```

```
        d.b3 := randint( 1, 16 )
```

```
        exit when d.b1 ~= d.b3 and
                d.b2 ~= d.b3
```

```
    end loop
```

```
    loop
```

```
        d.b4 := randint( 1, 16 )
```

```
        exit when d.b1 ~= d.b4 and
                d.b2 ~= d.b4 and
                d.b3 ~= d.b4
```

```
    end loop
```

```
end procedure
```

see also

rand

randomize

randseed

## Using subprograms

randomize    standard procedure

usage

randomize

remarks

Procedure sets the pseudo random seed used by functions "rand" and "randint" to a machine generated random value.

example

```
procedure start_rng( n : int )
```

```
    if n /= 0 then  
        randseed( n )  
    else  
        randomize  
    end if
```

```
end procedure
```

see also

randseed

Using subprograms



randseed     standard procedure

usage

```
randseed( new seed : int )
```

remarks

Procedure resets the pseudo random seed used by functions "rand" and "randint" to *new seed*.

example

```
procedure start_rng( n : int )
```

```
    if n ~= 0 then  
        randseed( n )  
    else  
        randomize  
    end if
```

```
end procedure
```

see also

randomize

Using subprograms

**real**            keyword

usage

**var** *name* : **real**

**const** *name* : **real** := *expression*

remarks

Standard data type specifier for real numbers.

see also

limits

Working with data

`realstr`      standard function

usage

```
realstr( expression : real,  
         format width : int ) : string
```

remarks

Function returns a string of the form:

`{blank}[-]digit{digit}.{digit}`

or of the form:

`{blank}[-]digit.{digit} e sign digit{digit}`

depending on the magnitude of *expression*. Blanks are added as needed to right justify the string. If *format width* is too small, the width is increased automatically.

example

```
const Pi : real := 2 * arcsin( 1 )
```

```
procedure put_volume( r : real )
```

```
    var v : real  
    var line : string
```

```
    v := ( 4 / 3 ) * Pi * r^3  
    line := "volume = " &  
            realstr( v, 24 )
```

```
    put line
```

```
end procedure
```

see also

[erealstr](#)

[frealstr](#)

[intstr](#)

[Using subprograms](#)

**record**      keyword

usage

**record**

*item*{, *item*} : *type specification*

    {*item*{, *item*} : *type specification*}

**end record**

remarks

Keyword is used to declare a record data type. To access elements of a record type, use the item selector "." between a variable name and the *item*.

see also

Working with data

repeat        standard function

usage

```
repeat( string : string,  
      expression : int ) : string
```

remarks

Function returns *expression* copies of *string* joined together into a single string.

example

```
procedure plot_sine( w : real )  
  
    var r, t : int  
    var s : string  
  
    for t := 0...40 do  
  
        r := round( 24 * sin( w * t ) )  
        r := r + 24  
        s := repeat( " ", r ) & "*" put s  
  
    end for  
  
end procedure
```

see also

Using subprograms

**return**      keyword

usage

**return** [*expression*]

remarks

Keyword causes a return from a function or procedure. A function must return a value. The type of *expression* must be compatible with a function's return type.

see also

Using subprograms

`round`      standard function

usage

```
round( expression : real ) : int
```

remarks

Function returns the integer nearest to *expression*.

example

```
% convert a real number into dollar-cents
```

```
function real_to_money( x : real ) : real
```

```
    var m : real
```

```
    m := 0.01 * round( 100 * x )
```

```
    return m
```

```
end function
```

see also

ceil

floor

sign

Using subprograms

setexp        standard function

usage

```
setexp( expression : real, exp : int ) : real
```

remarks

Function returns the value of *expression* with its exponent, base 10, changed to *exp*. If *expression* equals 0.0, zero is returned.

example

```
type bignum : record
    m : real      % mantissa
    x : int       % exponent
end record

% convert a real number into a big number
procedure convert( var d : bignum, s : real )

    d.x := getexp( s )
    d.m := setexp( s, 0 )

end procedure

% multiply two big numbers
% dest <- dest * srce
procedure multiply( var d, s : bignum )

    var dx : int

    d.m := d.m * s.m
    d.x := d.x + s.x

    dx := getexp( d.m )

    if dx ~= 0 then

        d.x := d.x + dx
        d.m := setexp( d.m, 0 )

    end if

end procedure
```

getexp



limits

Using subprograms

## **Help menu commands**

Commands for on-line help system.

### **Help index**

Keyboard command: Alt+H H

Opens the T interpreter's on-line help system at the table of contents.

### **Lookup**

Keyboard command: Alt+H L

Hot key: F1

Opens the T interpreter's on-line help system to a help topic about the word at the cursor location in the currently active window. If no related topic exists, the table of contents is displayed.

### **Using help**

Keyboard command: Alt+H U

Opens the Windows help on help facility.

### **About...**

Keyboard command: Alt+H A

Opens a dialog box which provides version and copyright information on the T interpreter.

sign            standard function

usage

sign( *expression* : **real** ) : **int**

remarks

Function returns the sign of *expression* as an integer -1 or +1.

example

% real absolute value

**function** rabs( arg : **real** ) : **real**

**return** sign( arg ) \* arg

**end function**

see also

ceil

floor

round

Using subprograms

`sin`            standard function

usage

`sin( expression : real ) : real`

remarks

Function returns the sine of *expression*. The value of *expression* is assumed to be in units of radians.

example

```
% cosecant
function csc( x : real ) : real

    var s : real

    s := sin( x )

    if s ~= 0.0 then
        x := 1 / s
    end if

    return x

end function
```

see also

cos

tan

Using subprograms

`sinh`            standard function

usage

```
sinh( expression : real ) : real
```

remarks

Function returns the hyperbolic sine of *expression*. The value of *expression* is assumed to be in units of radians.

example

```
% hyperbolic cosecant
function cosech( x : real ) : real

    var s : real

    s := sinh( x )

    if s ~= 0.0 then
        x := 1 / s
    end if

    return x

end function
```

see also

cosh

tanh

Using subprograms

sqrt            standard function

usage

sqrt( *expression* : **real** ) : **real**

remarks

Function returns the square root of *expression*. The value of *expression* must be non-negative or a run-time error will occur.

example

```
% roots of a*x^2 + b*x + c = 0
function roots( a, b, c : real,
                var x1, x2 : real ) : int

    var r, s : real

    s := b^2 - 4 * a * c
    if s < 0.0 then
        return 0
    end if

    r := sqrt( s )
    x1 := ( -b + r ) / ( 2 * a )
    x2 := ( -b - r ) / ( 2 * a )

    return 1

end function
```

see also

Using subprograms

**string**      keyword

usage

**string**

remarks

Standard type specifier for strings which are sequences of characters terminated by a null character.

see also

limits

Working with data

strint        standard function

usage

```
strint( expression : string ) : int
```

remarks

Function returns the integer equivalent to *expression*.

example

```
function get_number : int  
  
    var s : string  
  
    prompt "enter an integer:"  
    get s  
    return strint( s )  
  
end function
```

see also

strreal

Using subprograms



strreal     standard function

usage

strreal( *expression* : **string** ) : **real**

remarks

Function returns the real number equivalent of *expression*.

example

```
procedure put_money( d : string )
```

```
    var m : real
```

```
    m := strreal( d )
```

```
    put "$", m
```

```
end procedure
```

see also

strint

Using subprograms

tan            standard function

usage

tan( *expression* : **real** ) : **real**

remarks

Function returns the tangent of *expression*. The value of *expression* is assumed to be in units of radians.

example

```
% tan of 2*arg
function tan_2( arg : real ) : real

    var s, r : real

    s := tan( arg )
    r := 2 * s / ( 1 - s * s )

    return r

end function
```

see also

sin

cos

Using subprograms

tanh            standard function

usage

tanh( *expression* : **real** ) : **real**

remarks

Function returns the hyperbolic tangent of *expression*. The value of *expression* is assumed to be in units of radians.

example

```
% tanh of 2*arg
function tanh_2( arg : real ) : real

    var s, r : real

    s := tanh( arg )
    r := 2 * s / ( 1 + s * s )

    return r

end function
```

see also

cosh

sinh

Using subprograms

**then**        keyword

usage

**if** *boolean expression* **then**

see also

**if**

Looping and jumping

**true**            keyword

usage

*name* := **true**

remarks

Boolean constant; opposite of **false**.

see also

Working with data

**type**            keyword

usage

**type** *name* : *type specification*

remarks

Declares a named type for the *type specification*. Frequently, the *type specification* is one a user defines using an **array**, **record**, **union**, or **enum** declaration.

see also

Working with data

**value**        keyword

usage

**value** *constant*{, *constant*} :  
    *declarations and statements*

remarks

This keyword marks a block of *declarations and statements* to jump to in a case statement.

see also

**case**

Looping and jumping

**var**            keyword

usage

**var** *name*{, *name*} : *type specification* [ := *expression* ]

remarks

Keyword must precede each variable declaration and is also used to declare that a parameter in a subprogram's parameter list is passed by reference.

see also

**const**

**function**

**procedure**

Working with data

Using subprograms



watch            standard procedure

usage

watch( *expression* )

remarks

Displays the current value of *expression* on the debug screen when in debug mode.

see also

Getting started

Using subprograms

break

**when**            keyword

usage

**exit when** *boolean expression*

**continue when** *boolean expression*

remarks

Keyword is used to set a conditional jump in a for or loop statement.

see also

**for**

**loop**

Looping and jumping

**xor**            keyword

usage

*boolean expression* **xor** *boolean expression*

remarks

Operator returns a boolean value:

| x | y | x <b>xor</b> y |
|---|---|----------------|
|---|---|----------------|

|              |              |              |
|--------------|--------------|--------------|
| <b>false</b> | <b>false</b> | <b>false</b> |
|--------------|--------------|--------------|

|              |             |             |
|--------------|-------------|-------------|
| <b>false</b> | <b>true</b> | <b>true</b> |
|--------------|-------------|-------------|

|             |              |             |
|-------------|--------------|-------------|
| <b>true</b> | <b>false</b> | <b>true</b> |
|-------------|--------------|-------------|

|             |             |              |
|-------------|-------------|--------------|
| <b>true</b> | <b>true</b> | <b>false</b> |
|-------------|-------------|--------------|

see also

Working with data

## special symbols

These are special symbols used in the T programming language:

:= + - \* / ^ & : ,

. ... = ~= < <= > >=

( ) [ ] \ % " '

see also

Working with data

Looping and jumping

## **Edit menu commands**

### **Undo**

Keyboard command: Alt+E U

Hot keys: Ctrl+Z, Alt+Backspace

Restores a text line to its state prior to any editing of it. If restoration is not possible, Undo appears dimmed on the Edit menu.

### **Cut**

Keyboard command: Alt+E T

Hot keys: Ctrl+X, Shift+Delete

Deletes text from a document and places it onto the Clipboard, replacing the previous Clipboard contents.

### **Copy**

Keyboard command: Alt+E C

Hot keys: Ctrl+C, Ctrl+Insert

Copies text from a document onto the Clipboard, leaving the original intact and replacing the previous Clipboard contents.

### **Paste**

Keyboard command: Alt+E P

Hot keys: Ctrl+V, Shift+Insert

Pastes a copy of the Clipboard contents at the insertion point or replaces selected text in a document.

### **Delete**

Keyboard command: Alt+E L

Hot key: Ctrl+Delete

Deletes selected text from a document, but does not place the text onto the Clipboard. This operation cannot be undone.

### **Select All**

Keyboard command: Alt+E S

Selects all the text in a document at once. You can copy the selected text onto the Clipboard, delete it, or perform other editing actions.

### **Auto Indent**

Keyboard command: Alt+E A

Toggles the automatic indenting feature. When checked, the text entry point for a new line will be immediately below the first character on the line above.



