AVEVA Marine

Basic Features

User Guide

VANTAGE Marine Basic Features

Contents **Page**

1 Getting Started

1.1 Limitation of scope

This document concentrats on such functions that are originate from the Tribon System, mainly Hull and Drafting.

1.2 Starting Programs

After a successful installation most user programs can be started from the menu. The default location for this menu is below **Program** in the **Start** menu but this can vary depending on your operating system language and your installation.

1.3 Using Help

On-line help is available in two ways:

- Pressing the **F1** key.
- Selecting **Help Topics** from the **Help** menu.

Both actions above will display the help window.

1.4 Error Handling

Errors from programs can be of different kinds and originate from different sources:

1. Run-time errors from the operating system.
2. Errors from the basic procedure package.
3. Error messages from programs.

This document describes the principles of handling these categories of errors in a program. Variations may occur in different programs, especially for errors in category 3.

1.4.1 Run-time Errors

Run-time errors are, to a certain extent, taken care of by the error handling system and will be translated into category 2 or 3 errors. Many of them can, however, not be predicted and will therefore be presented to the system user in an ordinary way, i. e. on the display or in the log. The interpretation of such an error message is of course depending on the kind of operating system used. Most operating systems can give run-time errors with different degrees of severity (warnings, errors, etc.) and the actions to be taken by the user will be accordingly. The operating system vendor supplies documentation on errors of this category.

These errors may be *normal*, i. e. they report situations like "No more disc space", etc. In this case the error reason can easily be removed and the program restarted, if necessary. Other kinds of errors like "Floating zero divide" etc. must be reported to the systems maintenance personnel.

1.4.2 Basic Errors

The basic data and geometry handling procedures and also most of the application procedures have a uniform way of handling errors. Sources for these errors may be category 1 errors or abnormal situations within the procedures themselves. A special stack technique is used so that the program flow can be followed immediately before and after the error situation. In application programs, however, the ideal situation is that the error codes are translated into category 3 errors before being presented to the user.

In certain applications, the error codes may, however, reach the user. This is especially true in Hull.

1.4.3 Program Error Messages

In a user program the errors from categories 1 and 2 are taken care of and either translated into self-explanatory verbal error messages or presented directly to the end user. The errors will be shown on the display or in an error log. The program also controls the continuation after an error has occurred.

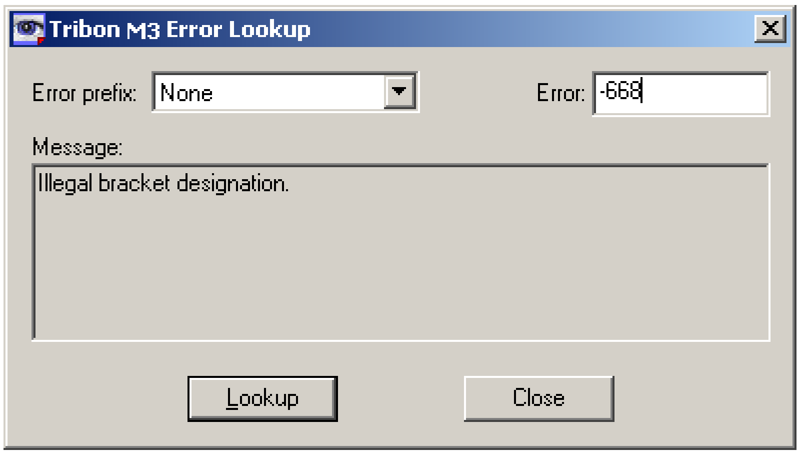
1.4.4 Error Codes in General

The system consists of a number of different applications specific to subsystems and they are all based on a set of common basic packages for handling data bases and graphics. This common base can be considered as a special subsystem.

Hull is different from the other application systems in the sense that some parts of it are *inherited* from other versions of the system. These parts have a separate error handling with error codes that partly coincide with those of the basic system.

1.4.5 Error Lookup

To find the meaning of an error from any of the applications, use the program **Error Lookup** in the **Start** menu.



Error prefix: None

Error: -668

Message:

Illegal bracket designation.

Figure 1:1. Error Lookup Program.

When started, the program prompts for an error prefix and an error code. Some programs (e. g. Hull) show errors in a slightly different way than the rest of the system. It is therefore important that the correct error prefix is selected before the error code is entered.

The error prefixes that are important and that differs from ordinary errors are:

```
"FFIX="
"QERROR="
"SSP ERROR="
"SYSTEM ERROR="
"WA201="
"WAFIX="
"WFFIX="
```

If the error code is prefixed with any of the above, the prefix must be selected in the box otherwise it is likely that the error message is incorrect.

When the error prefix is selected, supply an error code and press return. The system will present the text if available and prompt for a new error code. For *Error Lookup* to work it is necessary to have the name SBB_ERRMESS defined to the file sb_error_codes.txt that resides on SB_SYSTEM.

1.4.6 Hull Error Code Types

The error codes from Hull Programs are of different types:

1. Error codes either with their origin in the basic layer or from the application layer. These can be found with the Error Lookup program.
2. Program specific errors. These are always in the interval 1000-1100. An explanation of the error codes can always be found together with the documentation of the program reporting the error.

1.5 Geometry Handling

The basic geometrical element is the segment. A segment can be a line segment or a circular arc segment. A segment can be represented in R2-space or in R3-space.

A line segment is represented by its initial and terminal points, thus giving the segment a direction.

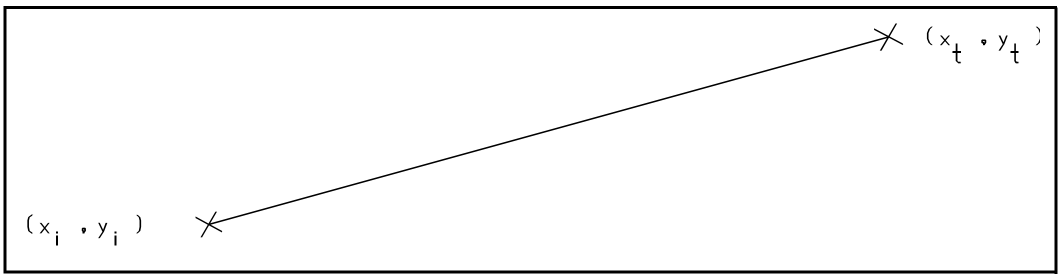


Figure 1:2. A Line Segment.

In R3, a third component z is added.

A circular arc segment is represented as the line but has also an amplitude vector (xh, yh).

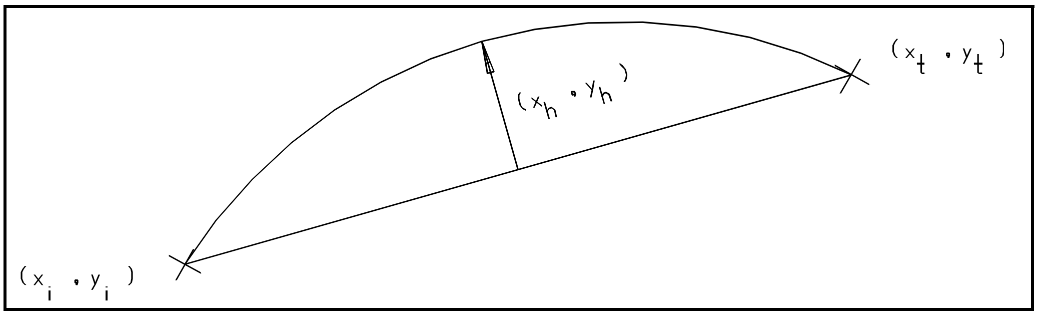


Figure 1:3. A circular Arc Segment.

In some cases in R2, the amplitude vector is replaced with a single number representing the length of the amplitude vector, i. e. the amplitude.

Continuous chains of segments can be combined to form contours.

1.6 The Interpretative Language (TIL)

This document describes the general syntax for the Interpretative Language (TIL). Different subsets of this language are used in different application programs. The documentation of these programs contains descriptions of the language as applied to each specific program. This document should, therefore, be regarded not as a guide for daily use but rather as a reference manual.

The first part of the document consists of basic definitions and the rest of the document contains a description of the general layout of a program.

1.6.1 Basic Definitions

- **Character Set**

All language constructions may be represented with a basic graphic character set, which is subdivided as follows:

- **Letters**

`<upper_case_letter>`	`::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z`
`<upper_case_letter_extended>`	`::= <upper_case_letter> |Å|Ä|Ö`
`<lower_case_letter>`	`::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|`
`<letter>`	`::= <upper_case_letter>|<lower_case_letter>`
`<letter_extended>`	`::= <upper_case_letter_extended>|<lower_case_letter_extended>`

Any lower case letter is equivalent to the corresponding upper case letter, except within character_strings.

- **Digits**

```
<digits>    ::= 0|1|2|3|4|5|6|7|8|9
```

- **Special Characters**

```
<special characters>    ::= #|$|%|&|'|(|)|*|+|,|-|.|/|:|;|<|=|?|@|^|_|~
```

- **Control Characters**

  Horizontal tabulate is equivalent to a space. Otherwise, no control characters may occur. If they do so, they are ignored.

• Lexical Units and Spacing Conventions

A program is a sequence of lexical units. The partition of the sequence into lines and the spacing between lexical units do not affect the meaning of the program. The lexical units are identifiers (including reserved words), numeric constants, character strings, delimiters and comments. Each lexical unit must fit in on one line.

- **Delimiters**

  A delimiter is either one of the following special characters:

```
<single_delimiter>       ::= &|(|)|*|+|,|-|/| |;|=|:|%
```
or one of the following compound symbols:
```
<component_delimiter>    ::= ** | := | :: | <= | >= | ==
```

Adjacent lexical units may be separated by spaces or by passage to a new line. An identifier or a numeric constant must be separated in this way from an adjacent identifier or numeric constant. Spaces must not occur within lexical units, except for character strings and comments.

Note: that the sign ! and single quote are not delimiters. They are part of other lexical units.

- **Identifiers**

  Identifiers are used as names.

```
<identifier> ::=    <letter_extended>|$|#{<letter_extended>|<digit>|$|#|_}
```

The length of an identifier should be in the range 1 through 32. **Note** that identifiers differing only in the use of the corresponding upper and lower case letters are considered to be the same.

- **Numeric Constants**

  There are two classes of numeric constants: integer constants and real constants.

```
<integer>           ::= <digit> {<digit>}
<exponent>          ::= E [+] <integer> | E - <integer>
<decimal_number>    ::= <integer>[.<integer>][<exponent>]
```

Real constants are distinguished by the presence of a decimal point.

Imperial units can be used see *Imperial Units Syntax Description.*

- **Boolean Constants**

The evaluation of a condition deliver a result of this type.

```
<boolean>     ::= TRUE | FALSE
```

- **Character Strings**

A character string is a sequence of zero or more characters prefixed and terminated by the string bracket character.

```
<character_string>  ::= '{<character>}'
```

In order to represent arbitrary strings of characters, any included string bracket character must be written twice. Concatenation must be used to represent strings longer than one line.

- **Comments**

A comment starts with an exclamation mark (!) and is terminated at the end of the line. It may only appear following a lexical unit or at the beginning of the program. Comments have no effect on the meaning of the program.

- **Reserved Words**

The identifiers listed below are called reserved words and are reserved for special significance in the language. Identifiers must not be reserved words. All characters in a reserved word are significant.

`AND, FALSE, NOT, OR, TRUE and XOR`

1.6.2 General Layout of a Program

- **Program**

An input program is divided into statements. A program has the following form:

```
<program>    ::= {<comment>} {<statement>}
```

- **Statements**

Each statement begins with a statement keyword that identifies the statement type and is terminated by the statement delimiter ; (semicolon). A statement has the following general form:

```
<statement>   ::= [<statement_keyword>]
                  {,<argument>}{,<argument>}
                  {/<attribute>};
```

If the statement keyword is omitted, then a default statement keyword is used.

- **Statement Keyword**

  A statement keyword can be abbreviated as long as the truncated statement keyword is unique or contains 3 characters. Keywords consisting of less than 3 characters cannot be abbreviated.

<statement_keyword>	::= <identifier>

- **Attributes**

`<attribute>`	`::= <attribute_keyword> [=(<argument>{,<argument>})]`

If the attribute keyword is followed by only one argument, and by no attributes belonging to the previous one(s), then the left and the right parentheses may be omitted.

- **Attribute Keywords**

  An attribute keyword can be abbreviated as long as the truncated attribute keyword is unique or contains 3 characters. Keywords consisting of less than 3 characters cannot be abbreviated.

<attribute_keywords>	::= <identifier>

- **Argument**

`<argument>`	`::= <position_independent_argument> |`
	`<position_dependent_argument>`

Position independent arguments can be given in any order. The order among position dependent arguments in the argument list is significant.

`<position_independant_argument>`	`::= <argument name>`
`= <expression>`	
`<position_dependant_argument>`	`::= <expression>`
`<argument_name>`	`::= <identifier>`

- **Expressions**

  An expression is a formula that defines the computation of a value. The syntax of a general expression is explained below:

`<range_expression>`	`:= <expression> [:<expression>[::<expression>]]`

`<expression>`	`:=<relation> {AND <relation>} |<relation> {OR <relation>} |`
	`<relation> {XOR <relation>}`

`<relation>`	`::=<simple_expression> [<relational_operator><simple_expression>]`
`<relational_operator>`	`::= == | /= | < | <= | > | >=`
`<simple_expression>`	`:= [unary_operator] <term> {adding_operator} <term>`
`<adding_operator>`	`:= + | - | &`
`<unary_operator>`	`:= + | - | NOT`
`<term>`	`:=<factor> {<multiplying_operator> <factor>}`

`<multiplying_operator>`	`:= * | %`
`<factor>`	`:= <primary> [** <factor>]`

```
<primary>:=       <decimal>               | <integer>
                  | <variable>            | <internal_function>
                  | (<expression>)        | <string>
                  | <boolean>
```

`<internal_function>`	`:= <function_name>(<expression>)`
`<variable>`	`::= <local_variable>`

Each primary has a value and a type. The type of an expression depends only on the type of its constituents and on the operators applied. The rules defining the allowed operand types and the corresponding result types are given below.

- **Logical Operators**

The logical operators are applicable to boolean values and returns boolean values.

Operator	Operation
AND	conjunction
OR	inclusive disjunction
XOR	exclusive disjunction

- **Relational Operators**

The relational operators should have operands of the same type and returns boolean values. Integer constants are converted to decimal constants before the test, if a decimal constant is included in the relation.

Operator	Operation
==	equality
/=	inequality
< <= > >=	test for ordering

- **Adding Operators**

  The adding operators should have operands of the same type and returns a result of the same type as the operands. Integer constants are converted to decimal constants before the operation, if a decimal constant is included in the term.

Operator	Operation	Operand type	Result type
`+`	addition	numeric	same numeric
`–`	subtraction	numeric	same numeric
`&`	catenation	string	string

- **Multiplying Operators**

  The multiplying operators should have operands of the same type and returns a result of the same type as the operands. Integer constants are converted to decimal constants before the operation, if a decimal constant is included in the factor.

Operator	Operation	Operand type	Result type
`*`	multiplication	numeric	same numeric
`%`	division	numeric	same numeric

- **Unary Operators**

  The unary operators are applied to a single operand and returns a result of the same type.

Operator	Operation	Operand type	Result type
`–`	negation	numeric	same numeric
`+`	identify	numeric	same numeric
`NOT`	logical negation	boolean	boolean

2 Imperial Units Syntax Description

2.1 Policy for Use of Imperial Units

Metric units are used:

- as database format
- as system information

Metric or optionally imperial units are used:

- as designer's input on screen or in batch programs
- as output on screen or list

Imperial units will be used if the environment variable for the unit is set to IMP. The default setting for imperial units can be overridden in work station programs if corresponding keyword in default file SBD_DEF1 is set. (See *User's Guide for Drafting)*.

For more information regarding imperial units being implemented for a specific program, see the documentation for the program.

2.2 Syntax Description

The value has to begin with a numeric sign, i.e. 6I is allowed I6 is not allowed. Decimals are always allowed.

The type of unit is always written in a short form and is described in the following chapters. The setting for the inch short form is common for many types of units. It's default value is I but it can be changed to C. This is controlled by the environment variable SB_IMPERIAL_INCH_SIGN which may have the value C or I. The environment variable affects:

- Linear Measures and Coordinates
- Area
- Volume

2.2.1 Linear Measures and Coordinates

The environment variables to get imperial units are SB_LINEAR_MEASURE_UNIT and SB_COORDINATE_UNIT.

The format will be as follows 3F516 i.e. 3 feet 5 inches and 6 fractions. The environment variable SB_LINEAR_MEASURE_FRACT can be set to 8THS, 16THS or 64THS and determines if the fractions are 8, 16 or 64 parts of an inch. Default is 16THS. It will be used if the environment variable not is set. When the measure or co-ordinate is displayed on list or on drawing the F will be exchanges to *and I to.* It is highly recommended to have the environment value SB_LINEAR_MEASURE_FRACT to the same as SB_COORDINATE_FRACT.

If the environment variable SB_LINEAR_MEASURE_NOT is set to 2F_LIM measures will be presented as inches and 16-ths if the measure is less than 2 feet. It is also possible to give the environment variable SB_LINEAR_MEASURE_NOT the value 6F_LIM, the system will then present measures less than 6 feet as inches and fractions.

The environment variable SB_LINEAR_MEASURE_SUPP can be set to the values 8THS, 16THS, 64THS and INCH. If the value is 8THS, 16THS or 64THS the fractions are suppressed if the measure is more that 1 inch. If the value is INCH the inch signs are suppressed if the measure is more than 1 foot. The measure is rounded. If the environment variable is set to INCH the value of SB_LINEAR_MEASURE_NOT will be ignored.

The measures will be presented with fractions if the environment variable SB_LINEAR_MEASURE_FRACT is set to 8THS, 16THS or 64THS and if the output field has enough space.

The default outlook of fraktions is shown in the table below. An alternative is available by setting the environment variable SB_IMPERIAL_OUTPUT_ALT to 1 (default is 0). The sign representing inch will then be moved last and a dash separates foot from inch. Example 3'02"7/8 will look like 3'-2 2/7" and 1'00"3/16 will look like 1-3/16".

Example of fractions with 16 parts of an inch:

Normal presentation:	**With fractions:**
1"00	1"
1"01	1"1/16
1"02	1"1/8
1"03	1"3/16
1"04	1"1/4
1"05	1"5/16
1"06	1"3/8
1"07	1"7/16
1"08	1"1/2
1"09	1"9/16
1"10	1"5/8
1"11	1"11/16
1"12	1"3/4
1"13	1"13/16
1"14	1"7/8
1"15	1"15/16

The environment variables SB_COORDINATE_NOT, SB_COORDINATE_SUPP and SB_COORDINATE_FRACT has got the same function as corresponding environment variable for measures.

The output will automatically be changed to feet with decimals if the length of the value is more than the field size.

2.2.2 Area

The environment variable to get imperial units is SB_AREA_UNIT.

The format will be as follows 10FF, 12II, 5M i.e 10 square feet, 12 square inches and 5 miles. When presenting the area square feet is automatically chosen if the value is more than 1FF, and miles is automatically chosen if the area is less than 1II. Miles is a measure which represents the diameter, in thousands of an inch, of a circle. Miles will in practice only be used for cable cross section areas since all other areas are to large.

2.2.3 Volume

The environment variable to get imperial units is SB_VOLUME_UNIT.

The format will be as follows 10FFF, 12III i.e 10 cubic feet and 12 cubic inches. When presenting the volume cubic feet is automatically chosen if the value is more than 1FFF, otherwise cubic inches is chosen.

2.2.4 Weight

The environment variable to get imperial units is SB_WEIGHT_UNIT.

The format will be as follows 15LB or 35T i.e 15 pounds or 35 long ton. When presenting the weight, long ton is automatically chosen if the value is more than 1 long ton, otherwise pounds are chosen.

SB_WEIGHT_TON

The type of ton that is supposed to be used is assigned to this environment variable. The valid types are LONG and SHORT.

1 long ton (default) is 2240 lb or 1 016,047 kg

1 short ton is 2000 lb or 907,1847 kg

SB_WEIGHT_NOT

The number of pounds, when the system will switch from presenting the unit as pounds to ton, is assigned to the environment variable. Default is 2240.