12d A File Format

Version 15

May 2022

12D SOLUTIONS PTY LTD

ACN 101 351 991

PO Box 351 Narrabeen NSW Australia 2101

Australia Telephone (02) 9970 7117 Fax (02) 9970 7118

International Telephone 61 2 9970 7117 Fax 61 2 9970 7118

email   support@12d.com   web page   www.12d.com

12d A File Format

This document is the 12d A File Fromat taken from the Reference Manual for the software product 12d Model.

Disclaimer

12d Model is supplied without any express or implied warranties whatsoever.

No warranty of fitness for a particular purpose is offered.

No liabilities in respect of engineering details and quantities produced by 12d Model are accepted.

Every effort has been taken to ensure that the advice given in this manual and the program 12d Model is correct, however, no warranty is expressed or implied by 12d Solutions Pty Ltd.

Copyright

Table of Contents

Preface

Introduction

12d Model is an interactive graphics program designed to process survey data, quickly build terrain, conceptual and detail design models.

Data is easily read in, triangulated and contoured to build an initial terrain model. Roads, platforms, channels or other design features can be added interactively and a merged model containing the initial terrain and the new design features formed to produce conceptual design models.

All Models can be examined in plan, section or perspective views. The number and type of views displayed on the screen is totally user defined.

By using a mouse and flexible on-screen menus, 12d Model is easy to use and requires a minimum of training.

To allow the interchanging of data between different survey and civil design packages, 12d Solutions maintain and have publish a text format, called ***12da*** (short for ***12d Archive***) for all the data stored in 12d Model. The *12d A* format is documented as an Appendix in the *12d Model Reference* manual.

This document is the *12d A File Format* Appendix from the ***12d Model Reference*** manual.

1 12d Archive File Format

The ***12d Archive*** file format (called **12d ascii i**n ***12d Model 10*** and earlier) is a text file definition from *12d Solutions* which is used for reading and writing out string data from ***12d Model***. 12d Archive files normally end in '**.**12da' and are often referred to as 12da files.

Unlike the earlier 12d Ascii files in ***12d Model 9***, from ***12d Model 10*** onwards the 12d Archive file is a **Unicode** file.

This document is for the ***12d Archive file*** format used in ***12d Model 15***.

For General Comments about 12da, see 1.1 General Comments about a 12da File

For the 12da definitions:

Attributes	1.3 Attributes
Commands	1.4 Commands
Each string type	1.5 12da Definition for each String Type
Tin	1.4.7 Tin
Super Tin	1.4.8 Super Tin
Trimesh	1.4.9 Trimesh

For the definitions of transitions and spirals available in ***12d Model***, see 1.2 Available Transition Types.

For documentation on the ***12d XML*** file format, see 1 12d XML File Format.

1.1 General Comments about a 12da File

Unicode - 12d Archive file is a Unicode file.

//

Anything written on a line after // is ignored. This is used to place comments in the file.

Blank Lines

Unless they are part of a text string, blank lines are ignored.

Spaces

Unless enclosed in quotes ("), more than one consecutive space or tab is treated as one space. Except when it is the delimiter after a //, an end of line (<enter>) is also considered a space.

Spaces and special characters in text strings

Any text string that includes spaces and any characters other than a to z, A to Z or 0 to 9 (alphanumeric), must be enclosed in double quotes. In text strings, double quotes " and backslash \ must be preceded by a \. For example, \" and \\ define a " and a \ respectively in a text string.

Names of Models, Tins and Super Tins

Models, tins, styles (linestyles), textstyles and colours can include the characters **a** to **z**, **A** to **Z**, **0** to **9** (alphanumeric characters) and **space**. Leading and trailing spaces are ignored. The names can be up to 255 characters in length.

The names for models, tins and super tins **can not** be blank.

The names for models, tins and super tins can contain upper and lower alpha characters which are stored, but for comparisons, the model names, tin names and super tin names are case insensitive. For example the model name "Fred" will be stored as "Fred" but "FRED" is considered to be the *same* model name as "Fred".

Within a project, each model name must be unique amongst all the model names in the project.

For tins and super tins, the name of a tin or a super tin must be unique amongst the combined list of tin names and super tin names.

Object Tree Names for Models and Tins

12d Model supports hierarchical (tree) names for models, tins and super tins, and the forward slash (**/**) is used to separate the different levels of the tree.

As for model and tins names themselves, each level name can include the characters **a** to **z**, **A** to **Z**, **0** to **9** (alphanumeric characters) and **space**. Leading and trailing spaces are ignored. The level names can be up to 255 characters in length.

For example, an object tree model name can be:

Stage 1/Water/Drainage

String names

String names can include the characters a to z, A to Z, 0 to 9 (alphanumeric characters), space, decimal point (.), plus (+), minus (-), comma (,), open and closed round brackets and equals (=). Leading and trailing spaces are ignored. String names can be up to 255 characters in length. If the string name includes anything other than alphanumeric characters, then the name must be enclosed in double quotes (").

String names can contain upper and lower alpha characters which are retained but case is ignored when selecting by string name. That is, the string name **Fred** will be stored as **Fred** but

FRED is not considered to be a different string name.

String names do not have to be unique and can be blank.

Attribute Names

Attribute names can include the characters a to z, A to Z, 0 to 9 (alphanumeric characters) and space. Leading and trailing spaces are ignored. The names can be up to 255 characters in length.

Attributes names **can not** be blank.

Attribute names are case sensitive. That is, the attribute name "Fred" is different to "FRED".

Names of Linestyles (Styles), Textstyles and Colours

Linestyles (styles)), textstyles and colours can include the characters **a** to **z**, **A** to **Z**, **0** to **9** (alphanumeric characters) and **space**. Leading and trailing spaces are ignored. The names can be up to 255 characters in length.

The names for linestyles, textstyles or colours **can not** be blank.

The names for linestyles, textstyles and colours can contain upper and lower alpha characters which are stored, but for comparisons, the linestyle names, textstyle names and colour names are case insensitive. For example the linestyle name "Bypass" will be stored as "Bypass" but "BYPASS" is considered to be the *same* linestyle name as "Bypass".

Within a project, each colour name must be unique amongst all the colour names in the project, each linestyle name must be unique amongst all the linestyle names in the project, and each textstyle name must be unique amongst all the textstyle names.

Times

The format of times such as **time_created** or **time_update** is:

*DD***-***MMM***-***YYYY hh***:***mm***:***ss*

and

DD in the day of the month

MMM is the first three character of the month (Jan for January, etc.)

YYYY in the year

hh in the hour in the 24-hour clock

mm in the number of minutes

ss in the number of seconds

For example, 10-Jun-2020 23:22:55

Note that the time format for 12da is different from the one of 12dxml.

Archive Version and Export Comments

When writing a 12d Archive file (12da/12daz/12dxml/12xmlz), comments about the project, settings used to control the contents of the file are written at the header of the file.

Beginning with V14 C2k, one critical comment is

archive_version "aa.bb.cc.ddd"

The first 3 parts (aa.bb.cc) describe the version of the database written.

When any change is made to the underlying database of 12d is made, the (aa.bb.cc) is updated to reflect a change.

The final part (d) is used when there is a change in the archive definition, that is not related to a database change.

Typically, this number will almost always be 0.

The archive_version is related to what you see in the title area of the program. Using V14 C2k as the example:

aa the release version of 12d (being 14)

bb the major number of 12d (2 for C2)

cc the minor number of 12d (11 for K)

As a general rule

aa the release version of 12d (being 14, 15 etc)

bb the major number of 12d (being 0 for Alpha/Beta, 1 for C1, 2 for C2, etc)

cc the minor number of 12d (being 1 for A, 2 for B, 11 for K, etc)

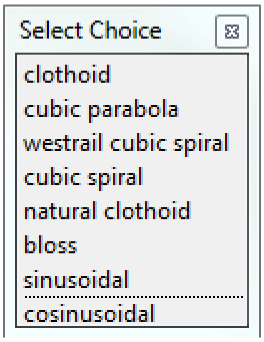
12da example:

```
// archive_version "15.00.00.473"
// decimal_places 8
// null -999
// do_null true
// output_times true
// output_ids false
// output_point_ids true
// output_attribute_ids true
// output_super_string_uids true
// output_sa_parts true
// output_drawables true
// output_new_pipes true
// dereference false
// project false
// output_project_description false
// output_compact_clouds true
// output_full_tin true
// output_model_paths false
// output_hex_floats false
// output_tin_hex_floats true
// model_attributes_mode 0
```

Continue to 1.2 Available Transition Types or return to 1 12d Archive File Format.

1.2 Available Transition Types

The transition that are available inside ***12d Model*** are called:



and these are defined as:

clothoid is the **clothoid spiral approximation** used by Australian road authorities and Queensland Rail

cubic parabola (or **NSW cubic parabola**) is a special transition curve used by NSW Railways. It is not a spiral.

westrail cubic spiral (or **westrail-cubic)** is a clothoid spiral approximation used by Westrail (WA railways).

cubic spiral (or **spiral**) is a low level spiral approximation. Mainly only used in surveying textbooks.

natural clothoid (or LandXML clothoid) is the full Euler clothoid spiral. This is not currently used by any Authority in Australia or New Zealand.

bloss is a Bloss curve. Not a spiral.

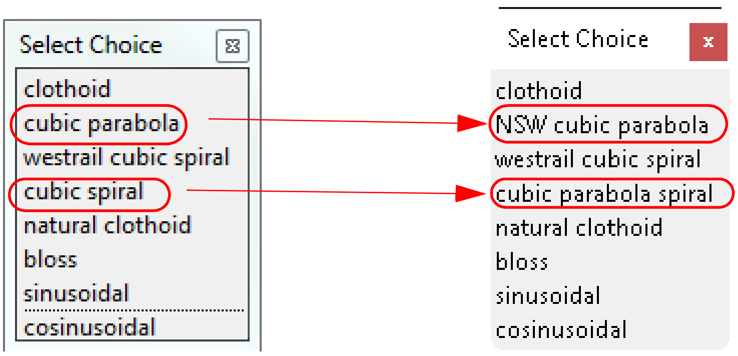
sinusoidal is a sinusoidal curve. Not a spiral.

cosinusoidal is a cosinusoidal curve. Not a spiral.

Although these are the names stored internally inside ***12d Model*** and match the standard ones used in Australia, unfortunately there is no universal definition of what names match which transitions.

So to make it clearer, especially because of the confusion about the term "cubic parabola", in some ***12d Model*** options the pop-up displays different names. This is especially true for options using a transition mapping file (trans_map file) to map the transition names used inside ***12d Model*** to those used in another software package.

In the alternate transition pop-up, "**cubic parabola"** is displayed as "**NSW cubic parabola**" and "**cubic spiral**" is displayed as "**cubic parabola spiral**" to help users realise that the word "cubic parabola" is confusing and could refer to the NSW Rail cubic transition and what is sometimes called the "cubic parabola" approximation to the clothoid spiral.



Continue to 1.3 Attributes or return to 1 12d Archive File Format.

1.3 Attributes

Many *12d* ***Model*** objects (models and elements such as individual strings and tins) can have an unlimited number of named **attributes** of type integer (numbers), real and text.

The attributes for an object are given in an ***attributes block*** which consists of the keyword **attributes** followed by the definitions of the *individual attributes* enclosed in start and end curly braces { and }. That is, an ***attributes_block*** is

attributes {

attribute_1

attribute_2

...

attribute_n

}

where the attribute definitions for the individual attributes *attribute_i* consists of

attribute_type ***attribute_name*** ***attribute_value***

where

attribute_type	is integer, real or text
attribute_name	is the unique attribute name for the object.

If the attribute name includes spaces then the text of the name must be enclosed in double quote character (")

and

attribute_value	is the appropriate value of the integer, real or a text.

Within an object, the attribute names are case sensitive and must be unique. That is, for attribute names, upper and lower case alphabet characters are considered to be different characters.

If the *text* for a text attribute includes spaces then the text must be enclosed in double quote characters ("). It the text is blank, it is given as "".

An example of and ***attribute block*** defining four attributes named "pole id", "street", "pole height" and "pole wires" is:

```
attributes {
  text        "pole id"        "QMR-37"
  text         street          "477 Boundary St"
  real        "pole height"     5.25
  integer     "pole wires"      3
}
```

Continue to 1.4 Commands or return to 1 12d Archive File Format.

1.4 Commands

Commands consist of a ***keyword*** followed by a space and then a ***value*** (a keyword and its value is often referred to as a ***keyword pair***). A *value* must always exist.

keyword   *value*   // a keyword pair

There can be more than on command keyword pair per line as long as each keyword pair is separated by a space. In fact, the *keyword* can be on one line and the *value* on the next line.

Although the names of commands are only shown in lower case in these notes, commands are case insensitive and all combinations of case are recognised as the same command. That is **model**, **MODEL** and **ModeL** are all recognised as the command **model**.

For the definition of the commands in the 12da file *see*:

1.4.1 Model
1.4.2 Colour
1.4.3 Style
1.4.4 Breakline
1.4.5 Null
1.4.6 String
1.4.7 Tin
1.4.8 Super Tin
1.4.9 Trimesh

Or return to 1 12d Archive File Format.

1.4.1 Model

There are two formats for the **model** command:

(a) model command when there are no attributes for the model

model *model_name*

All elements (strings, tins, plot frames *etc*) following until the next **model** keyword are placed in the model *model_name*. This can be overridden for an element by a **model** command inside the element definition.

The default model name used for elements when no model name has been specified is ***data***.

(b) model command when there are model attributes

If the model includes attributes, the following form of the *model* command must be used.

model {

name *model_name*

attributes_block

}

where the *attributes_block* is defined in 1.3 Attributes.

For example:

```
model {
  name        "telegraph poles"
  attributes {
    text            "pole id"          "QMR-37"
    text            "street"           "477 Boundary St"
    real            "pole height"       5.25
    integer         "pole wires"        3
  }
}
```

Continue to 1.4.2 Colour or return to 1.4 Commands or 1 12d Archive File Format.

1.4.2 Colour

The format of the **colour** command is:

colour *colour_name*

When reading a 12da file, there is a ***current colour,*** which has the default value of ***red***, and when a **colour** command is read, the ***current colour*** is set to *colour_name.*

When strings are read in a 12da file, they are given the *current colour*.

This can be overridden for a string by a *string colour command* inside the string command defining that string. For the definition of the string command, see 1.4.6 String.

Continue to 1.4.3 Style or return to 1.4 Commands or 1 12d Archive File Format.

1.4.3 Style

The format of the **style** command is:

style *linestyle_name*

When reading a 12da file, there is a ***current linestyle,*** which has the default value of ***1***, and when a **style** command is read, the ***current linestyle*** is set to *linestyle_name.*

When strings are read in a 12da file, they are given the *current linestyle*.

This can be overridden for a string by a *string style command* inside the string command defining that string. For the definition of the string command, see 1.4.6 String.

Continue to 1.4.4 Breakline or return to 1.4 Commands or 1 12d Archive File Format.

1.4.4 Breakline

The format of the **breakline** command is:

breakline *breakline_type*

where *breakline_type* is *point* or *line*.

When reading a 12da file, there is a ***current breakline type,*** which has the default value of ***point***, and when a **breakline** command is read, the ***current breakline type*** is set to *breakline_type.*

When strings are read in a 12da file, they are given the *current breakline type*.

This can be overridden for a string by a *string breakline command* inside the string command defining that string. For the definition of the string command, see 1.4.6 String.

Continue to 1.4.5 Null or return to 1.4 Commands or 1 12d Archive File Format.

1.4.5 Null

The format of the **null** command is:

null *null_value*

When reading a 12da file, there is a ***current null value,*** which has the default value of ***-999***, and when a **null** command is read, the ***current null value*** is set to *null_value.*

When strings are read in a 12da file and the string has z-values equal to *null_value*, then the z-value is replaced by the ***12d Model*** null value.

This can be overridden for a string by a *null_value command* inside the string command defining that string. For the definition of the string command, see 1.4.6 String.

Continue to 1.4.6 String or return to 1.4 Commands or 1 12d Archive File Format.

1.4.6 String

The format of the **string** command is:

string *string_type* {

attributes_block

string_command_1

string_command_2

...

string_command_n

}

The ***string_type*** is compulsory and must be followed by all the string information enclosed in curly braces { and }.

So if a ***string type,*** or possibly information inside the string is not recognised, the 12da reader has a chance of being able to jump over the string by looking for the end curly brace }.

Inside the braces are **string commands** as keyword pairs defining information for the string.

There can be more than one *string command* keyword pair per line as long as each keyword pair is separated by a space. In fact, the *keyword* can be on one line and the *value* on the next line.

Any unrecognised *string commands* are ignored.

The *string command keyword pairs* include **model**, **colour**, **style** and **breakline,** which are all *optional* inside the string definition. However if any of them exist inside a string definition, then the *string command keyword* overrides the current value for **model**, **colour**, **style** or **breakline** *commands* but the override is only for that particular string.

Not all string types can have an *attributes_block*.

For some string types (*e.g.* super string) there is more data required than just the *string command* keyword pairs.

This extra data is contained is blocks consisting of a *keyword* followed by the required information enclosed in the curly braces **{** and **}**. For example attributes for all *string types* and (x,y,z) data for a super string.

For all string types, if there is not enough recognised information to define the string, the string is ignored.

For the definition for each *string type* and the allowed *string commands* and extra data that is required for that *string type, see* 1.5 12da Definition for each String Type.

Note: if the string does not have any attributes then the ***attributes_block*** can be left out entirely (see 1.3 Attributes for the definition of *attributes_block*).

Continue to 1.4.7 Tin or return to 1.4 Commands or 1 12d Archive File Format.

1.4.7 Tin

Tins (***t***riangulated ***i***rregular ***n***etworks) can be written out and read in from a 12da.

Each tin has text name, ***tin_name***, of up to two hundred alphanumeric characters and spaces and although the tin names are stored with upper or lower case alphabet characters, for comparisons of the tin names, the names are considered to be case insensitive.

Within a project, the name of a tin or a super tin must be unique amongst the combined list of tin names and super tin names.

There are two formats for a tin - one that lists **all the triangles**, including the nulled (invisible) triangles in the tin, and the other that only lists the **visible triangles** that make up the tin.

See

1.4.7.1 All Triangles in the Tin - Visible and Invisible
1.4.7.2 Visible Triangles Only

1.4.7.1 All Triangles in the Tin - Visible and Invisible

This format writes out all the triangles in the tin, including the invisible and construction triangles.

This format take more disk space but cannot be misinterpreted because it includes all the points, triangles and all the neighbouring triangles for each edge of a triangle.

It is also the best method for writing out **large tins** as it is much faster to read in and create a tin.

The keyword for the full format for a **tin** element is **full_tin** and it is defined by:

```
full_tin {
  name    tin_name        // MANDATORY name of the tin when created in 12d Model

  time_created text               // optional - time tin first created
  time_updated text               // optional - time tin last modified


// Attributes Block:


// The attributes style, faces, null_length, null_angle, null_combined_value
// and null_combined_angle are special attributes that has extra information used by
// 12d Model to create the tin. These special attributes should not be deleted.
//
// The attributes in this block and the Attributes block itself are optional.
// When a tin is read into 12d Model from a 12da file, the style is used
// as the Tin style.

  attributes {
```

```
text "style"              text        // name of line style for the tin
integer "faces            0/1         // 0 non triangle data, 1 triangle data
real "null_length"        value       // values for null by angle/length
real "null_angle"         value       // angle in radians
real "null_combined_length"   value
real "null_combined_angle"    value       // angle in radians

//                any other attributes
}                                            // end of attributes block
```

// **Points Block**

//

//This gives the coordinates of the points that will be vertices of the triangles

// in the tin, including the first four points that are construction points

// that are on the four corners of a rectangle that surrounds the actual data.

//

// The points are implicitly numbered by the order in the list (starting at point 1).

//

// The Points Block is **MANDATORY**

```
points {                              // x y z for each point in the tin
   x-value   y-value   -value         // point 1
       "         "        "           // point 2
      "          "         "
}                                     // end of points block
```

// **Triangles Block**

//

// This gives the triangles that make up the tin.

// Each triangle is given as a triplet of the point numbers that make up

// the triangle vertices (the point numbers are the implicit position of the points

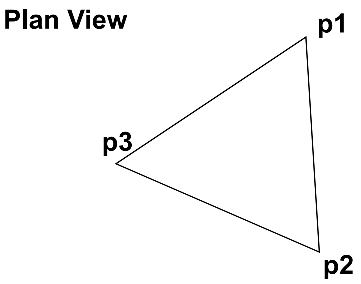
// given in the Points Block.

// The order of the triangles is unimportant but the **order of the points in the**

// **triangle is important**.

// The vertices of each triangle must be listed in a **clockwise** order when looking at the tin

// from above.



// The Triangles Block is **MANDATORY**

```
triangles {                         // points making up each triangle
    T1-1     T1-2     T1-3          // point numbers of the 3 vertices of first triangle.
    T2-1     T2-2     T-33          // point numbers of the 3 vertices of second triangle.
       "        "
       "        "
}                          // end of triangles block
```

// The first edge of triangle k is from Point Tk-1 to Point Tk-2.

// The second edge of triangle k is from Point Tk-2 to Tk-3.

// The third edge of triangle k is from Point Tk-3 to Tk-1.

// **Note**: Construction Triangles

// Any triangle that contains any of the first four points

//(construction points) is a **construction triangle** and is usually

// not displayed.

// **Neighbours Block**

//

// For each triangle, this gives for each edge the number of the triangle that is

// the neighbour of that edge of the triangle.

// The order of the entries in the neighbours block must match the order of the

// triangles in the Triangles Block. So there is exactly one entry for each triangle.

//

// The Neighbours Block is **MANDATORY**

```
neighbours {
    t1_e1_nb_tr    t1_e2_nb_tr   t1_e3_nb_tr
    t2_e1_nb_tr    t2_e2_nb_tr   t2_e3_nb_tr
"      "      "
```

```
    tn_e1_nb_tr    tn_e2_nb_tr   tn_e3_nb_tr
  }                              // end of neighbour block
```

// where *tk_e1_nb_tr* *tk_e2_nb_tr* *tk_e3_nb_tri* are the triangle numbers from the
// Triangles block of the neighbouring triangle for each edge of the k'th triangle
// For each triangle, the order of the neighbouring triangles must match the order
// that the edges are defined for the triangle in the triangles block
// **Note:** the neighbour value of **0** is used for the outside triangles that contain
// exactly two of the points 1, 2, 3 or 4 and so have edges that have no neighbouring triangle

// Nulling Block
//
// Triangles can be visible or nulled (invisible)
// Whether a triangle is null or visible is individually given where:
// **1** means the triangle is **null**, and
// **2** means the triangle is **visible**.

// The order of the entries in the nulling block must be the same as the order of the
// triangles in the Triangles Block.

// The Nulling Block is **MANDATORY**
//

// Triangles can be visible or nulled (invisible)

```
  nulling {
    V1   V2   V3  ...     V15   V16
    V17    Cv18    C19   ...  V31   V32
      "      "      "
    Vn-2    Vn-1    Vn
  }                              // end of nulling block
```

// Base Colour
// The tin has a base colour that is the default colour for all triangles

```
  colour  tin_base_colour                // optional - base colour of the tin
```

// Colours Block

//

// Triangles can be given colours other than the base colour by including

// a *colours* block. The colour for each triangle in then individually given

// (-1 means base colour). The order is the same as the order of the triangles in

// the Triangles Block.

//

// If all the triangles are the base colour, then simply omit the Colours Block

```
colours {
   C1    C2    C3                 // colour for each triangle given in triangle order
   C4   C5   C6   C7              // colour "-1" means use the base tin colour.
      "    "    "
      "    "    "
}                        // end of colours block
```

// Input Block

//

// More information about how the tin was created by **12d Model**.

// None of this information is needed when reading a tin into **12d Model**.

// This block can be omitted

```
input {                                           // data for reconstructing tin from strings
   preserve_strings     true/false                // if true, preserve breaklines etc.
   remove_bubbles       true/false                //
   weed_tin             true/false
   triangle_data        true/false
   sort_tin             true/false
   cell_method          true/false

   models {
      "model_name_1"              // name of the first model making up the tin
      "model_name_2"              // name of the second model making up the tin
      "    "    "
      "    "    "
   }                              // end of models block
```

```
    }                                          // end of input block
}                                              // end of tin 12a definition
```

1.4.7.2 Visible Triangles Only

The format to write out only the visible triangles in a tin is a simple format for most software packages to write. However because the null regions are not explicitly given, more processing time is required to read the tin back in and construct all the null regions.

The keyword denoting the format where just the visible triangles of a **tin** element are written out is **tin** and its definition is:

```
tin {
  name    tin_name          // MANDATORY name of the tin when created in 12d Model

  time_created text               // optional - time tin first created
  time_updated text               // optional - time tin last modified


// Attributes Block:

// The attributes style, faces, null_length, null_angle, null_combined_value
// and null_combined_angle are special attributes that has extra information used by
// 12d Model to create the tin. These special attributes should not be deleted.
//
// The attributes in this block and the Attributes block itself are optional.
// When a tin is read into 12d Model from a 12da file, the style is used
// as the Tin style.

  attributes {
    text "style"          text        // name of line style for the tin
    integer "faces        0/1         // 0 non triangle data, 1 triangle data
    real "null_length"    value       // values for null by angle/length
    real "null_angle"      value  // angle in radians
    real "null_combined_length"   value
    real "null_combined_angle"     value       // angle in radians

//                  any other attributes
  }                                           // end of attributes block


// Points Block
//
```

// Coordinates of the points at the vertices of the triangles

// The points are implicitly numbered by the order in the list (starting at point 1).

//

// The Points Block is MANDATORY

```
points {                          // x y z for each point in the tin
   x-value   y-value   -value     // point 1
      "         "        "        // point 2
      "         "        "
}                                 // end of points block
```

// Triangles Block

// This gives the triangles that make up the tin.

// Each triangle is given as a triplet of the point numbers that make up

// the triangle vertices (the point numbers are the implicit position of the points

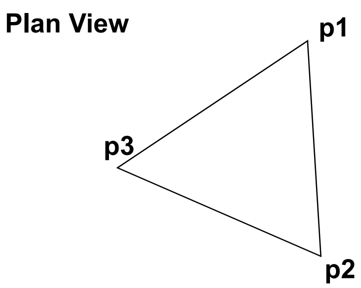
// given in the Points Block.

// The order of the triangles is unimportant but the **order of the points in the**

// **triangle is important**.

// The vertices of each triangle must be listed in a **clockwise** order when looking at the tin

// from above.



// The Triangles Block is MANDATORY

```
triangles {                       // points making up each triangle
   T1-1   T1-2   T1-3             // point numbers of the 3 vertices of first triangle.
   T2-1   T2-2   T-33             // point numbers of the 3 vertices of second triangle.
      "      "
      "      "
}                      // end of triangles block
```

// Base Colour
//   The tin has a base colour that is the default colour for all triangles

```
colour tin_base_colour          // optional - base colour of the tin
```

// Colours Block
//
// Triangles can be given colours other than the base colour by including
// a *colours* block. The colour for each triangle in then individually given
// (-1 means base colour). The order is the same as the order of the triangles in
// the Triangles Block.
//
// If all the triangles are the base colour, then simply omit the Colours Block

```
colours {
  C1   C2   C3            // colour for each triangle given in triangle order
  C4  C5  C6  C7          // colour "-1" means use the base tin colour.
    "   "   "
    "   "   "
}                  // end of colours block
```

// Input Block
//
// More information about how the tin was created by ***12d Model***.
// None of this information is needed when reading a tin into ***12d Model***.
// This block can be omitted

```
input {                                   // data for reconstructing tin from strings
  preserve_strings     true/false         // if true, preserve breaklines etc.
  remove_bubbles       true/false         //
  weed_tin             true/false
  triangle_data        true/false
  sort_tin             true/false
  cell_method          true/false

  models {
    "model_name_1"              // name of the first model making up the tin
```

```
"model_name_2"                    // name of the second model making up the tin
"       "       "
"       "       "
    }                             // end of models block


  }                               // end of input block
}                                 // end of tin 12a definition
```

Continue to 1.4.8 Super Tin or return to 1.4 Commands or 1 12d Archive File Format.

1.4.8 Super Tin

Super Tins, which consists of a number of **tins** (***t***riangulated ***i***rregular ***n***etworks), can be written out and read in from a 12da.

Each super tin has text name, ***tin_name***, of up to two hundred alphanumeric characters and spaces and although the tin names are stored with upper or lower case alphabet characters, for comparisons of the tin names, the names are considered to be case insensitive.

Within a project, the name of a tin or a super tin must be unique amongst the **combined** list of tin names and super tin names.

```
super_tin {
  name    tin_name          // MANDATORY name of the super tin

  time_created  text                // optional - time super tin first created
  time_updated  text                // optional - time super tin last modified


// Attributes Block:

// This is mainly information used by 12d Model to create the super tin.
// The attributes in this block and the Attributes block itself are optional.
//   When a super tin is read into 12d Model from a 12da file, the style is used
//   as the Super Tin style.

  attributes {
        text "style"              text        // name of line style for the tin
//                any other attributes
  }                                                 // end of attributes block


// Super Tin Colour
//      The super tin has a base colour

  colour  tin_base_colour           // optional - base colour of the super tin

// Tins Block
//
// This is the list of tins that make up the super tin.
// This block is MANDATORY
```

```
    tins {                                  // list of tins for the super tin
      "tin_name_1"                          // name of the first tin making up the super tin
      "tin_name_2"                          // name of the second tin making up the super tin
      "     "     "
      "     "     "
    }                                       // end of tins block

}                                           // end of super tin 12a definition
```

Note that the tins that make up the super tin must exist in the ***12d Model project*** for the super tin to be fully defined.

Continue to 1.4.9 Trimesh or return to 1.4 Commands or 1 12d Archive File Format.

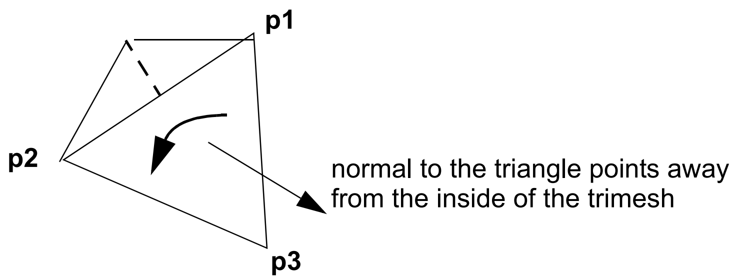
1.4.9 Trimesh

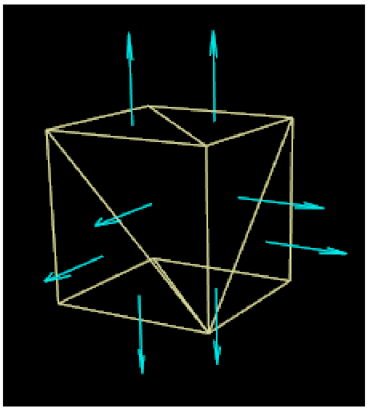
A trimesh is made up of 3D triangles and can be described by giving the list of ***m*** vertices in the trimesh and the three vertices that make up each of the ***n*** triangular faces. The normal to each triangle face points to the "outside" of the trimesh.

The order of the points **p1**, **p2** and **p3** in the triangle triple is important and must be such that the direction of the normal vector to each triangle points away from the inside of the trimesh.

That is, the normal vector of the triangle which is given by the cross product of the two vectors **p1p2** and **p1p3** points away from the inside of the trimesh.

Hence when looking towards the triangle from the outside, the points **p1**, **p2** and **p3** are in a **counter clockwise** order around the triangle.





The 12da definition of a trimesh is:

```
primitive_3d {
   name    primitive_3d_namer                // name of the primitive_3d
   colour    primitive_3d_name_colour        // colour of the primitive_3d
                                             // the primitive_3d has a base colour that is the
                                             // default colour for the primitive

   time_created text                         // optional - time primitive first created
   time_updated text                         // optional - time primitive last modified

// Attributes Block:
```

// The attributes in this block and the Attributes block itself are optional.

```
  attributes {
//            attributes
  }                                   // end of attributes block
```

// Trimesh Block

// At a minimum, the trimesh block contains information on the vertices and faces that make up the trimesh.

```
    trimesh_3d {
//            vertices_block
//            faces_block
//            edges_block         // optional for checking only
//            info_block          // optional
      blend real_number           // optional - blend value for the trimesh
//            vertex_infos_block  // optional
//            vertex_flags_block  // optional
//            edge_infos_block    // optional
//            edge_flags_block    // optional
//            face_infos_block    // optional
//            face_flags_block    // optional

    }                             // end of trimesh_3d block
```

//

// Vertices Block

// of the points at the vertices of the triangle faces that make up the trimesh.

// The vertices are implicitly numbered by the order in the list (starting at point 1).

//

```
    vertices {                    // x y z for each vertex in the trimesh
      x-value  y-value  -value    // vertex 1
        "        "        "       // vertex 2
```

```
    "           "           "
}                                       // end of vertices block
```

// The *faces_block* is MANDATORY and consists of the triangles//

// Each triangle in the trimesh is given as a triplet of the vertex that make up

// the triangle (the vertex numbers are the implicit position of the vertices

// given in the Vertices Block.

// The order of the faces in the faces block is important for many calculations, mesh properties, geometric structures

//

// The Faces Block is MANDATORY

```
faces {                                 // vertices making up each triangle
T1-1      T1-2      T1-3                // vertex numbers of the 3 vertices of first triangle.
T2-1      T2-2      T-33                // vertex numbers of the 3 vertices of second triangle.
    "           "
    "           "
}                       // end of faces block
```

// The *edges_block* is OPTIONAL and consists of the edges//

// Each edge in the trimesh is given as a pair of the vertex that make up

// the edge (the vertex numbers are the implicit position of the vertices

// given in the Edges Block.

// The order of the edges in the edges block is important for many calculations, mesh
// properties, geometric structures. The correct order can only be formed inside
// **12d *Model***. For manual construction of the 12da file for trimesh, the user should

// leave out the *edges_block.*

//

// The Edges Block is OPTIONAL

```
edges {                                 // vertices making up each triangle
T1-1      T1-2      /                   // vertex numbers of the 2 vertices of first edge.
T2-1      T2-2                          // vertex numbers of the 2 vertices of second edge.
    "           "
    "           "
}                       // end of edges block

}                       // end of trimesh_3d block
```

```
    }                                   // end of primitive_3d 12a definition
```

// Info block contain four field: flag, key, colour and name

```
info {                                  // vertices making up each triangle
flag    // integer, 12d Model internal use only.
key     // integer from 0 to 255, 12d Model internal use only.
colour  // 12d Model colour.
name    // string name.
}                       // end of faces block
```

// Info block of primitive_3d is OPTIONAL

// vertex_infos, edge_infos and face_infos block of primitive_3d are OPTIONAL and consists of info_blocks

```
vertex_infos {                                  // making up info blocks
flag-1    key-1    colour-1    name-1           // first info.
flag-2    key-2    colour-2    name-2           // second info.
   "        "
   "        "
}                       // end of vertex_infos block
```

// vertex_flags, edge_flags and face_flags block of primitive_3d are OPTIONAL and consists of sequences of indexes for info block. The size of vertex_flags should equals the number of vertices in the trimesh; and the same for edge_flags, face_flags. If the index is 0 it means there is no information on the current vertex (edge, face).

```
vertex_flags {                                  // making up info blocks
index-1                                         // info index of the first vertex.
index-2                                         // info index of the second vertex.
   "        "
   "        "
}                       // end of vertex_flags block
```

// for example if the trimesh has two kinds of vertex info

```
vertex_infos {                                  // making up info blocks
0   0   green   "no name"                       // first info.
0   1   blue   "no name"                        // second info.
   "        "
   "        "
}                       // end of vertex_infos block
```

// and 5 points.

```
vertex_flags {                                  // making up info blocks
2                                   // info index of the first vertex.
0                                   // info index of the second vertex.
```

1 // info index of the third vertex.

2 // info index of the fourth vertex.

0 // info index of the fifth vertex.

} // end of vertex_flags block

Then the trimesh has two blue points (number 1 and 4), one green point (number 3), and two points without colour (number 2 and 5).

Continue to 1.5 12da Definition for each String Type or return to 1.4 Commands or 1 12d Archive File Format.

1.5 12da Definition for each String Type

For the 12da definition of each string type, see:

1.5.1 Arc String
1.5.2 Circle String
1.5.3 Drainage String
1.5.4 Face String
1.5.5 Feature String
1.5.6 Interface String
1.5.7 Plot Frame String
1.5.8 Super String
1.5.9 Super Alignment String
1.5.10 Text String

And for the superceded strings, see:

1.5.11 2d String
1.5.12 3d String
1.5.13 4d String
1.5.16 Alignment String
1.5.14 Pipe String
1.5.17 Pipeline String
1.5.15 Polyline String

Or return to 1 12d Archive File Format.

1.5.1 Arc String

```
string arc {
  model model_name   name string_name
  colour colour_name   style style_name
  chainage start_chainage   interval value   radius value
  xcentre value   ycentre value   zcentre value
  xstart value   ystart value   zstart value
  xend value   yend value   zend value
}
```

Continue to 1.5.2 Circle String or return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.

1.5.2 Circle String

string circle {

model *model_name* name *string_name*

colour *colour_name* style *style_name*

chainage *start_chainage* interval *value* radius *value*

zcentre *value* xcentre *value* ycentre *value*

}

Continue to 1.5.3 Drainage String or return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.

1.5.3 Drainage String

```
string drainage {
  chainage   start_chainage
  model   model_name   name  string_name
  colour   colour_name   style  style_name
  breakline  point or line
  attributes {
    text   Tin   finished_surface_tin
    text   NSTin   natural_surface_tin
    integer    "_floating"   1|0            // 1 for floating, 0 not floating
  }
  outfall     outfall_value                 // z-value at the outfall
  flow_direction    0|1                     // 0 drainage line is defined from downstream
                                            // to upstream

  data {                                    // key word - geometry of the drainage string
    x-value    y-value    z-value    radius   bulge
       "          "          "
       "          "          "
  }
  pit {                          // pit/manhole - one pit record for each pit/manhole
                                 // in the order along the string
      name           text                   // pit name
      type           text                   // pit type
      road_name      text                   // road name
      road_chainage       chainage          // road chainage
      diameter            value             // pit diameter
      floating       yes|no                 // is pit floating or not
      chainage       pit_chainage           // internal use only
     ip              value                  // internal use only
      ratio          value                  // internal use only
      x              x-value                // x-value of top of pit
      y              y-value                // y-value of top of pit
      z              z-value                // z-value of top of pit
  }
  pit_v2 {                       // pit/manhole - one pit record for each pit/manhole
                                 // in the order along the string
                                 // is an alternative to pit with version 14 features
```

```
    name              text                   // pit name
    type              text                   // pit type
    road_name         text                   // road name
    road_chainage     chainage               // road chainage
    diameter          value                  // pit diameter
    floating          yes|no                 // is pit floating or not
    chainage          pit_chainage           // internal use only
   ip                 value                  // internal use only
    ratio             value                  // internal use only
    x                 x-value                // x-value of top of pit
    y                 y-value                // y-value of top of pit
    z                 z-value                // z-value of top of pit
    base_extended          none/width/length  // extended direction of the pit base
    base_angle_mode             choices       //
    base_angle                  angle         //
    base_height                 value         //
    riser_enabled               true/false    //
    riser_diameter              value         //
    riser_width                 value         //
    riser_extended         none/width/length  // extended direction of the riser
    riser_offset_x              value         //
    riser_offset_y              value         //
    riser_colour                colour        //
    base_thickness_top          value                  //
    riser_thickness_front          value //
    riser_thickness_back           value  //
    riser_thickness_left           value  //
    riser_thickness_right           value  //
}
pipe {                       // one pipe record for each pipe connecting pits/manholes
                             // in the order they occur along the string
    name              text                   // pipe name
    type              text                   // pipe type
    diameter          value                  // pit diameter
    us_level          value                  //
    ds_level          value                  //
    us_hgl            value                  //
    ds_hgl            value                  //
    flow_velocity     value                  //
```

```
    flow_volume        value              //
}
property_control {
  name              text                  // lot name
  colour            colour_name
  grade             value                 // grade of pipe in units of "1v in"
  cover             value                 // cover of the of pipe
  diameter          value                 // diameter of the of pipe
  boundary          value                 // boundary trap value
  chainage          chainage              // internal use only
  ip                value                 // internal use only
  ratio             value                 // internal use only
  x                 x-value               // x value of where pipe connects to sewer
  y                 y-value               // y value of where pipe connects to sewer
  z                 z-value               // internal use only

  data {                        // key word - geometry of the property control
  x-value    y-value    z-value    radius   bulge
     "          "          "
     "          "          "
}
house_connection {   // warning - house connections may change in future versions
  name              text                  // house connection name
  hcb               integer               // user given integer
  colour            colour_name
  grade             value                 // grade of connection in units of "1v in"
  depth             value
  diameter          value
  side              left or right
  length            value
  type              text                  // connection type
  material          text                  // material type
  bush              text                  // bush type
  level             value
  adopted_level     value
  chainage          chainage              // internal use only
  ip                value                 // internal use only
  ratio             value                 // internal use only
  x                 x-value               // x value of where pipe connects to sewer
```

```
        y            y-value            // y value of where pipe connects to sewer
        z            z-value            // internal use only
    }

}            // end of drainage-sewer data
```

Continue to 1.5.4 Face String or return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.

1.5.4 Face String

```
string face {
  model   model_name    name   string_name
  colour   colour_name   style   style_name
  chainage start_chainage      breakline point or line
  hatch_angle value
  hatch_distance value
  hatch_colour colour
  edge_colour colour
  fill_mode 0 or 1
  edge_mode 0 or 1
  data {                                               // keyword
     x-value     y-value     z-value
       "           "           "
  }
}
```

Continue to 1.5.5 Feature String or return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.

1.5.5 Feature String

```
string feature {
  model  model_name   name   string_name
  colour   colour_name   style   style_name
  chainage start_chainage   interval value   radius value
  zcentre  value   xcentre value   ycentre value
}
```

Continue to 1.5.6 Interface String or return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.

1.5.6 Interface String

```
string interface {
  chainage start_chainage
  model model_name   name string_name
  colour colour_name   style style_name
  breakline point or line
  data {                                          // keyword
    x-value    y-value    z-value      mode
       "         "          "            "        // mode = -1 cut
        "         "          "           "        //         0 surface
  }                                               //         1 fill
}
```

Continue to 1.5.7 Plot Frame String or return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.

1.5.7 Plot Frame String

Plot frames can be written out and read in from a 12da file.

```
string plot_frame {
  name              frame_name
  title_file        filename
  border            0 or 1
  viewport          0 or 1
  user_title_file   0 or 1
  title_1           text
  title_2           text
  plot_file         filename
  text_size         mm
  sheet_code        text
  width             value
  height            value
  scale             value
  rotation          value
  xorigin           value
  yorigin           value
  left_margin       mm
  right_margin      mm
  top_margin        mm
  bottom_margin     mm
  plotter           text
  colour            colour
  textstyle         textstyle_name
}
```

Continue to 1.5.8 Super String or return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.

1.5.8 Super String

Because the super string is so versatile, its 12da format looks complicated but it is very logical and actually quite simple.

In its most primitive form, the super string is simply a set of (x,y) values as in a 2d string, or (x,y,z) values as in a 3d string, or (x,y,z,radius,bulge_flag) as for a polyline string or even lines, arcs and transitions (spirals and non-spiral transitions). see 1.2 Available Transition Types.

Additional blocks of information can extend the definition of the super string. For example, text, pipe diameters and visibility.

Some of the properties of the super string extend what were constant properties for the entire string in other string types. For example, *breakline* type for the string extends to *tinability* of *vertices* and *segments*. One colour for the string extends to individual colours for each segment.

Other properties such as vertex id's (point numbers), visibility and culvert data are entirely new.

For user attributes, the super string still has the standard user attributes defined for the entire string, but user attributes for each vertex and segment are also supported.

The definition of a closed string has been refined for polyline and super strings. For other string types, closing a string simply meant having the first vertex the same as the last vertex. Hence the vertex was duplicated.

For a super string, being closed is a property of the string and no extra vertex is needed. That is, the first and the last vertices are not the same for a closed super string and the super string knows there is an additional segment from the last vertex back to the first vertex.

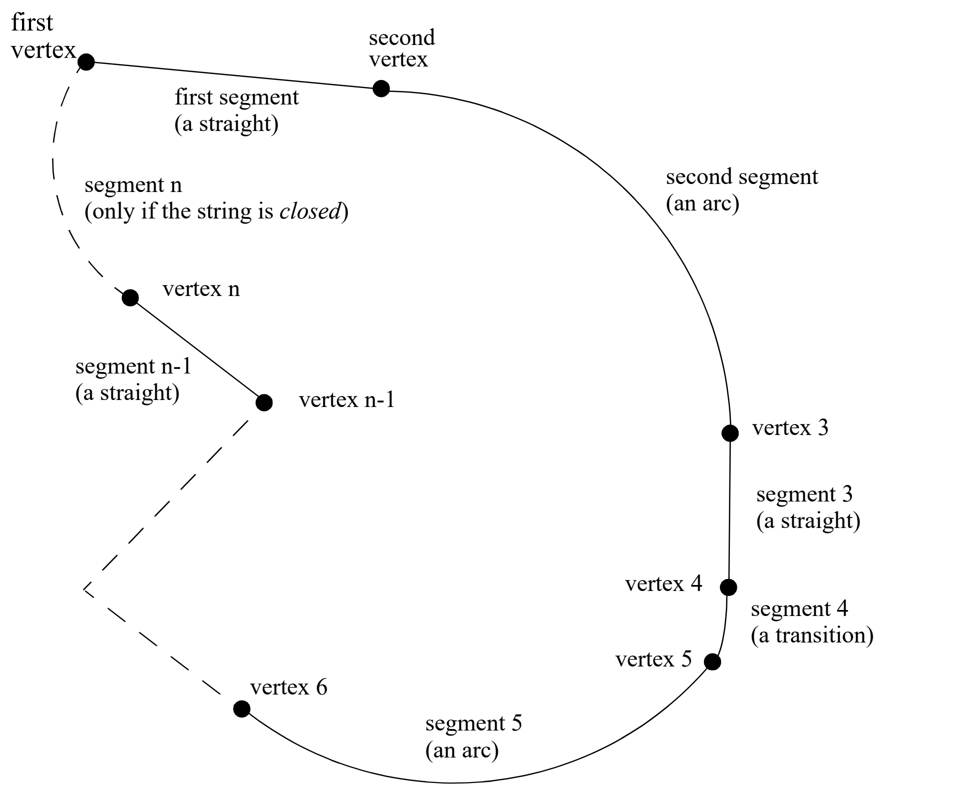
Hence in the 12da format, there is a *closed* flag for the super string:

closed *true* or *false*

where *true* can be 1 or T or t or Y or y (or words starting with T, t, Y or y))
and *false* is 0 or F or f or N or n (or words starting with F, f, N or n.

Thus if a string has *n* vertices, then an open string has *n-1* segments joining the vertices and a closed string has *n* segments since there is an additional segment from the last to the first vertex.

With the additional data for vertices and segments in the super string, the data is in vertex or segment order. So for a string with *n* vertices, there must be *n* bits of vertex data. For segments, if the string is open then there only needs to be *n-1* bits of segment data but for closed strings, there must be *n* bits of data. For an open string, *n* bits of segment data can be specified and the *nth* bit will be read in and stored. If the string is then closed, the *nth* bit of data will be used for the extra segment.



Vertices and Segments Forming the Super String

The full 12da definition of the super string is:

```
string super {
  chainage start_chainage
  model model_name   name string_name
  colour colour_name   style style_name
  breakline point or line
  closed true or false
  interval {
    chord_arc     value          // chord-to-arc tolerance for curves
    distance      value          // chainage interval to break the geometry up
  }

  block of info {
  }
  block of info {
  }
  block of info {
```

```
    }
}
```

The blocks of info can be broken up into four types.

(a) blocks defining the position of the vertices in x, y and z

data_2d **or** *data_3d. See* 1.5.8.1 Blocks Defining the Position of the Vertices in x, y and z.

(b) blocks defining the geometry of the segments

radius_data and *major_data* **or** *geometry_data.*

See 1.5.8.2 Blocks Defining the Geometry of the Segments.

(c) a superseded block defining vertices and segment geometry data

See 1.5.8.3 Block Defining Both the Vertices and Segments - Superceded.

(d) extra information for the vertices and/or segments

These include blocks for colour of each segment, vertex ids, round pipe diameters, culvert widths and heights etc.

See 1.5.8.4 Other Blocks.

The definition for the blocks of each type now follows.

1.5.8.1 Blocks Defining the Position of the Vertices in x, y and z

For (x, y) Values with a Constant z

If there is only (x,y) values at each vertex (like a 2d string):

```
data_2d {                                    // keyword
            x-value    y-value
               "          "
               "          "
}
```

and if there is a non-null constant z for the string

```
z    value
```

Without any more information, the segments will default to being straight lines.

If some of the segments in the super string are not straights (arcs, transitions or offset transitions) then either the ***radius_data*** and ***major_data*** blocks, OR the ***geometry_data*** block must also be used.

For (x,y,z) Values

If there is (x,y,z) values at each vertex (like a 3d string):

```
data_3d {                                    // keyword
    x-value    y-value    z-value
       "          "          "
       "          "          "
}
```

Without any more information, the segments will default to being straight lines.

If some of the segments in the super string are not straights (arcs, transitions or offset transitions) then either the ***radius_data*** and ***major_data*** blocks, OR the ***geometry_data*** block must also be used.

1.5.8.2 Blocks Defining the Geometry of the Segments

If the segments only includes arcs and straights, then the *radius_data* and *major_data* blocks can be used. See 1.5.8.2.1 Straights and Arcs Only for the Segments

If the segments include transitions or offset transitions, then the *geometry_data* block **must** be used. See 1.5.8.2.2 Straights, Arcs, Transitions and Offset Transitions for the Segments

1.5.8.2.1 Straights and Arcs Only for the Segments

If either *data_2d* or data_3d was used to defined the points at the ends of the segments and some of the segments are arcs and there are no transitions, then the radius information for the segments is given in the *radius_data* and *major_data* blocks.

There must be a value for each segment and if a segment is a straight, a radius of zero (0) is used.

```
radius_data {                                       // keyword
                radius for first segment
                radius for second segment
                  . . .
                radius for last segment
}

major_data {                                        // keyword
              bulge flag for first segment
              bulge flag for second segment
                    . . .
              bulge flag for last segment
}
```

1.5.8.2.2 Straights, Arcs, Transitions and Offset Transitions for the Segments

If either *data_2d* or data_3d was used to defined the points at the ends of the segments and some of the segments are transitions, then the geometry for each segment ***must*** be given in a *geometry_data* block.

```
geometry_data {
  segment_info_1 {
    information on the first segment
  }
  segment_info_2 {
    information on the second segment
  }
      "     "
```

```
        "     "
      segment_info_n-1 {          // the last segment if it is open
        information on the (n-1) segment
      }
      segment_info_n {            // the last segment if it is closed
        information on the n-th segment
      }
    }
```

where the *segment_info* blocks are from the following:

1. straight

   See 1.5.8.2.2.1 Straight

2. arc

   See 1.5.8.2.2.2 Arc

3. transition with no offset

   See 1.5.8.2.2.3 Spiral - spiral and non-spiral transitions without offsets. Also see 1.2 Available Transition Types.

4. transitions with or without offsets

   See 1.5.8.2.2.4 Curve block - Transition and Offset Transitions. Also see 1.2 Available Transition Types.

1.5.8.2.2.1 Straight

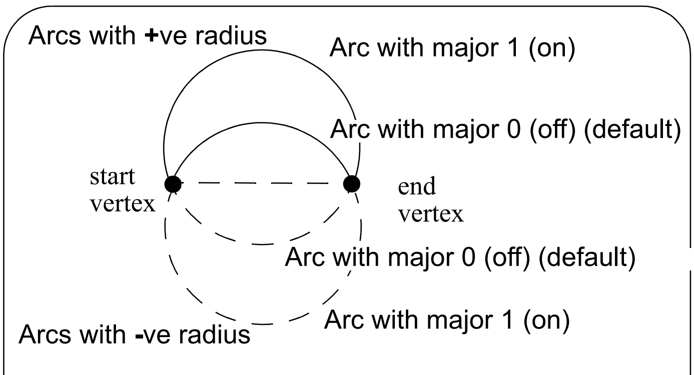
No parameters are needed for defining a straight segment. The *straight* block is simply:

```
  straight {                      // no parameters are needed for a straight
  }
```

1.5.8.2.2.2 Arc

There are four possibilities for an arc of a given radius placed between two vertices.

We use *positive* and *negative* radius, and a flag *major* which can be set to 1 (on) or off (0) to differentiate between the four possibilities.



Arcs with same radius but with major on or off

So the *arc* block is:

```
arc {
   radius  value     // radius of the arc (+ve is above the line connecting the vertices)
   major   0 or 1    // 0 is the smaller arc, 1 the larger arc).
}
```

1.5.8.2.2.3 Spiral - spiral and non-spiral transitions without offsets

There can be a partial transition between adjacent vertices. The partial transition is defined by the parameters

- **l**1 length of the full transition up to the start vertex
- **r**1 radius of the transition at the start vertex
- **a**1 angle in decimal degrees of the tangent to the transition at the start vertex
- **l**2 length of the full transition up to the end vertex
- **r**2 radius at the end vertex
- **a**2 angle in decimal degrees of the tangent to the transition at the end vertex

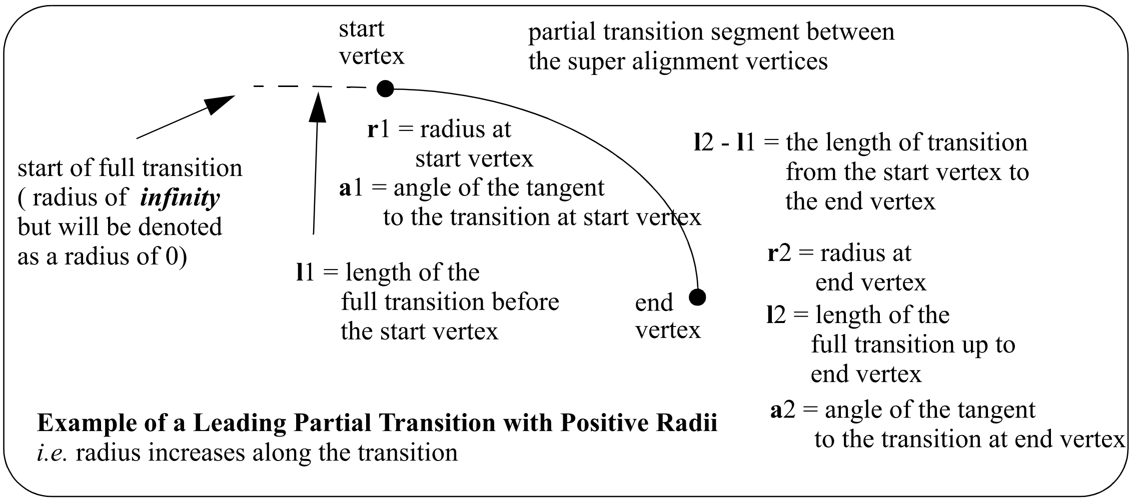
Since a radius can not be zero, a radius of infinity is denoted by *zero*.

The transition is said to be a *leading* transition if the absolute value of the radius is increasing along the direction of the transition (the transition will *tighten*). Otherwise it is a *trailing* transition.

If a leading transition is a full transition then **r**1 = 0 and **l**1 = 0. Similarly if a trailing transition is a full transition then **r**2 = 0 and **l**2 = 0.

For a partial transition, if the coordinates of the start of the full transition are needed then they can be calculated from l1,r1,a1, l2,r2,a2 and the coordinates of the start and end vertices.

Note that the radii can be positive or negative. If the radii's are positive then a leading transition will curl to the right (and will be above the line joining the start and end vertices).



Example of a Leading Partial Transition with Positive Radii
i.e. radius increases along the transition

The parameters for the *spiral* block are:

```
spiral {
   type      value     // type can be clothoid, cubic parabola, westrail-cubic,
                       // cubic spiral, natural clothoid, bloss,

                       // bloss, sinusoidal, cosinusoidal

   leading   1 or 0    // 1 denotes a leading transition, 0 a trailing transition
```

```
l1          value          // length of the full transition at start vertex
r1          value          // radius at the start vertex
a1          value          // angle in decimal degrees of the tangent to the transition
                           // at the start vertex
l2          value          // length of the full transition at end vertex
r2          value          // radius at end vertex
a2          value          // angle in decimal degrees of the tangent to the transition
                           // at the end vertex
}
```

1.5.8.2.2.4 Curve block - Transition and Offset Transitions

The curve block can be used in place of the spiral block and covers transitions with both zero not zero offsets.

An ***offset transition*** is a fixed perpendicular offset (***offset_real)*** of a base transition where the base transition is a Euler spiral (or a certain approximation to it) or some other specially defined curve. The base transition has a start point where the absolute radius of the curve is infinity and then has a continuously decreasing absolute radius as you continue along the curve (this may be in a forward or reverse direction).

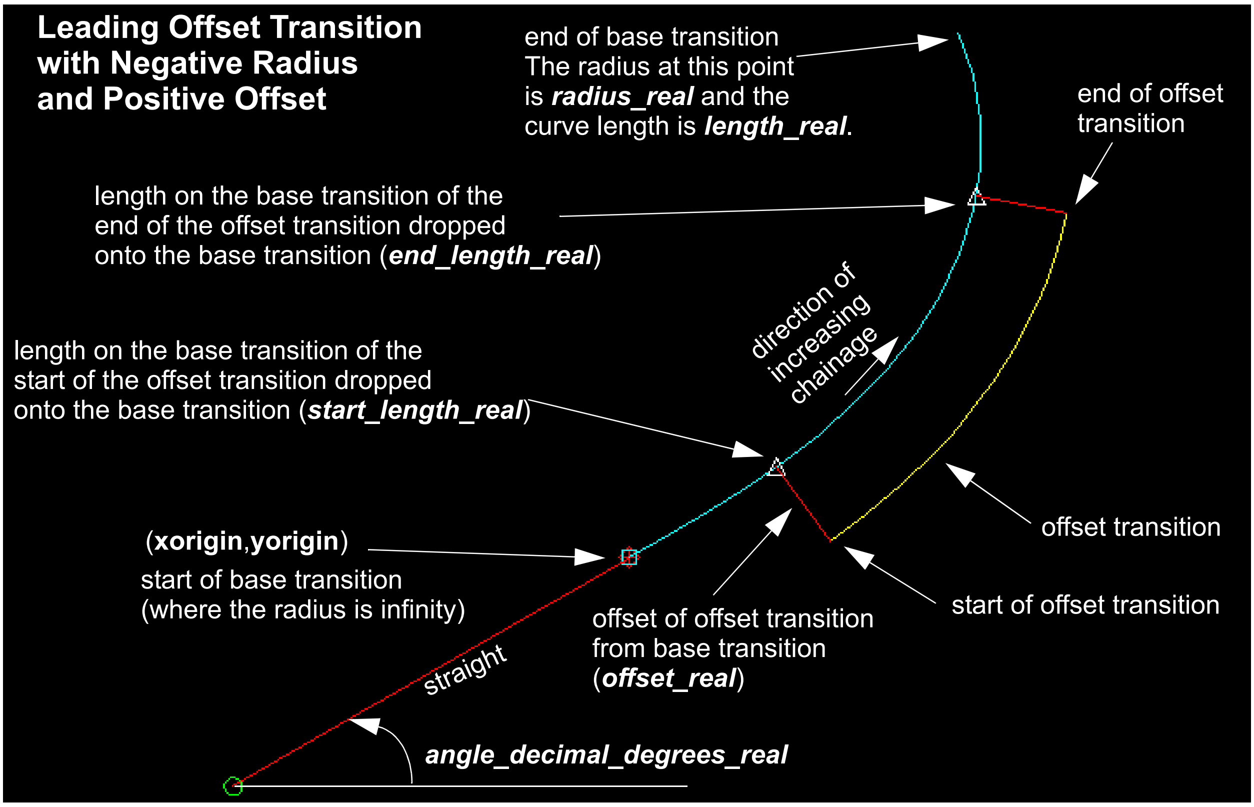
The ***base transition*** is defined by giving its starting point **(xorigin, yorigin)** where the radius is infinity and the angle of the tangential line at the start point is ***angle_decimal_degrees_real*** and the fact that the radius ***radius_real*** occurs at a given curve length ***length_real*** along the base transition.

The ***offset transition*** is a fixed offset (***offset_real***) from the base transition and goes from a start point that is specified by giving the length on the base transition where the start point drops perpendicularly onto the base transition (***start_length_real***) and to the end point that is specified by length on the base transition where the end point drops perpendicularly onto the base transition (***end_length_real***). The offset can be positive or negative.

If you are travelling along the curve in a forward direction (increasing chainage) then the base transition is said to be a **leading transition** if the absolute radius decreases as you go along the curve, and a **trailing transition** if the absolute radius decreases.

The ***end radius*** can be ***positive*** or ***negative***.

If you are travelling along the curve in a forward direction then for a leading transition, if the **end radius** is **positive** then the curve curls to the right, and for a negative end radius, the curve curls to the left.



Note: when the offset is zero, the offset transition is a standard transition which is then the same as the curves in the ***spiral*** block.

The ***curve*** block covers both spiral and non-spiral transitions with a zero or non zero offset.

The parameters for the *curve* block are:

`curve {`

`type`	*transition_type*	// any of the transitions supported in **12d** ***Model***
`leading`	*1 or 0*	// 1 denotes a leading transition, 0 a trailing transition
`xorigin`	*value*	// (**xorigin**,**yorigin**) is the origin of the base transition
`yorigin`	*value*	// That is, where the radius is infinity

`radius`	*value*	// **radius** is the radius at the end of the base transition
		// If radius is positive, the curve goes to the right when
		// travelling in decreasing absolute radius

`length`	*value*	// **length** is the curve length to the end of the base transition

`start`	*value*	// **start** is the curve length on the base transition where the
		// end of the offset transition drops perpendicularly onto the
		// base transition

`end`	*value*	// **end** is the curve length on the base transition where the
		// start of the offset transition drops perpendicularly onto the
		// base transition

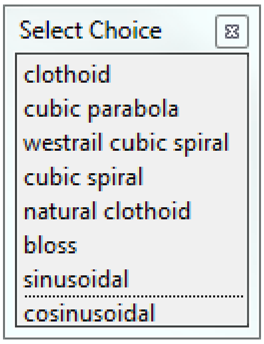
`angle`	*value*	// **angle** in decimal degrees is the angle of the tangent of the
		// base transition at the origin of the base transition.
		// It is measured in decimal degrees in a counter clockwise
		// direction from the positive x-axis

`offset`	*value*	// **offset** is the perpendicular offset distance of the offset

```
                                    // transition from the base transition.
                                    // For a leading transition, a positive value offsets from the
                                    // base transition to the right and a negative value offsets it
                                    // to the left, as you travel in a forward direction.
    mvalue    value                 // if the transition is a cubic parabola then mvalue is the
                                    // mvalue for the cubic parabola. Otherwise mvalue is zero.
}
```

Notes

1. The ***spiral*** block covers both spiral and non-spiral transitions without offsets.
1. The ***cur ve*** block covers both spiral and non-spiral transitions with and without offsets.
2. The transitions/spirals supported by ***12d Model*** are:



Clothoid - spiral approximation used by Australian road authorities and Queensland Rail.

Cubic parabola – special transition curve used by NSW railways. Not a spiral.

Westrail cubic – spiral approximating used by WA railways.

Cubic spiral – low level spiral approximation. Only ever used in surveying textbooks.

Natural Clothoid – the proper Euler spiral. Not used by any authority.

Bloss – special transition used by Deutsche Bahn. Not a spiral.

Sinusoidal - special transition. Not a spiral.

Cosinusoidal - special transition. Not a spiral.

1.5.8.3 Block Defining Both the Vertices and Segments - Superceded

For compatibility with the polyline, the ***data*** block gives the (x,y,z,radius,bulge) values at each vertex of the string and so defines both the vertices and the geometry of the segments in the one block.

```
data {                                             // keyword
          x-value   y-value   z-value   radius   bulge
             "         "         "
             "         "         "
}
```

A radius of zero (0) is used to denote a *straight* segment.

This block is now superceded although it may still exist in older 12da files.

1.5.8.4 Other Blocks

See

1.5.8.4.1 Colour

There can be one colour for the entire super string which is given by the `colour` command at the beginning of the string definitions (before the blocks of information) or the colour varies for each segment of the super string and is specified in a colour_data block.

```
colour_data {                                        // keyword
            colour for first segment
            colour for second segment
               . . .
            colour for last segment
}
```

1.5.8.4.2 Vertex Id's (Point Numbers)

Each vertex can have a vertex id (point number). This is not the order number of the vertex in the string but is a separate id which is usually different for every vertex in every string. The vertex id can be alphanumeric.

```
point_data {                                         // keyword
            vertex id or first vertex                // alphanumeric
            vertex id for second vertex
               . . .
            vertex id for last vertex
}
```

1.5.8.4.3 Pipe Diameters

There can be one pipe diameter value for the entire super string or the pipe diameter varies for each segment of the super string.

```
diameter_value value
```

or

```
diameter_data {                                      // keyword
```

pipe diameter for first segment

pipe diameter for second segment

. . .

pipe diameter for last segment

}

1.5.8.4.4 Culvert Dimensions

There can be one culvert width and height for the entire super string or the culvert width and height vary for each segment of the super string.

```
culvert_value {
    width    value
    height   value
}
```

or

```
culvert_data { properties {width   value       // width and height for first segment
                           height  value
                          }
               properties {width   value       // width and height for second segment
                           height  value
                          }
               . . .
               properties {width   value       // width and height for last segment
                           height  value
                          }
}
```

Note that one super string cannot have both pipe diameters and culvert dimensions.

1.5.8.4.5 Justification for Pipe or Culverts

There can be only one justification for the pipe or culvert for the entire super string.

```
justify   justification        // bottom or invert
                               // top or obvert
                               // centre (default)
```

1.5.8.4.6 Tinability

For a *super string*, the concept of breakline has been extended to a property called **tinable** which can be set independently for each vertex and each segment of the super string.

If a vertex is tinable, then the vertex is used in triangulations. If the vertex is not tinable, then the vertex is ignored when triangulating.

If a segment is tinable, then the segment is used as a side of a triangle during triangulation. This may not be possible if there are *crossing* tinable segments.

```
vertex_tinable_data {                          // keyword
```

```
            tinable flag for first vertex         // 1 for tinable
            tinable flag for second vertex        // 0 for not tinable
            . . .
            tinable flag for last vertex
```

```
segment_tinable_data {                              // keyword
            tinable flag for first segment        // 1 for tinable
            tinable flag for second segment       // 0 for not tinable
            . . .
            tinable flag for last segment
}
```

Note that even if a segment is set to tinable, is can only be used if both its end vertices are also tinable.

1.5.8.4.7 Visibility

For a *super string*, the concept of visibility and invisibility for vertices and segments has been introduced.

```
vertex_visible_data {                               // keyword
            visibility flag for first vertex      // 1 for visible
            visibility flag for second vertex     // 0 for invisible
            . . .
            visibility flag for last vertex
}
```

```
segment_visible_data {                              // keyword
            visibility flag for first segment     // 1 for visible
            visibility flag for second segment    // 0 for invisible
            . . .
            visibility flag for last segment
}
```

1.5.8.4.8 Vertex Text and Vertex Annotation

There can be the same piece of text for every vertex in the super string or a different text for each vertex of the super string. How the text is drawn is specified by vertex annotation values. Note that in vertex annotations, all vertices must be either worldsize or all vertices papersize. That is, worldsize and papersize can not be mixed - the first one found is used for all vertices.

```
vertex_text_value        text
```

or

```
vertex_text_data {                                  // keyword
            text for first vertex                 // text string, enclose
            text for second vertex                // by " " if there are any
```

```
                 . . .                          // spaces in the text string
            text for last vertex
}

vertex_annotate_value {                         // keyword
                 angle value   offset value   raise value
                 textstyle textstyle_name   slant degrees   xfactor value
                 worldsize value or papersize value or screensize value
                 justify   "top|middle|bottom-left|centre|right"
                 colour colour_name
}
```

or

```
vertex_annotate_data {                          // keyword
        properties {  angle value   offset value   raise value
                        textstyle textstyle   slant degrees   xfactor value
                      worldsize value or papersize value or screensize value
                        justify   "top|middle|bottom-left|centre|right"
                        colour colour_name
        }
        properties {    text properties second vertex
        }
        properties {      ...
        }
        properties {    text properties for last vertex
        }
}
```

1.5.8.4.9 Segment Text and Segment Annotation

There can be the same piece of text for every segment in the super string or a different text for each segment of the super string. How the text is drawn is specified by segment annotation values. Note that in segment annotations, all segments must be either worldsize or all segments papersize. That is, worldsize and papersize can not be mixed - the first one found is used for all segments. However, vertex text and segment text do not both have to be papersize or worldsize.

```
segment_text_value        text
```

or

```
segment_text_data {                             // keyword
                text for first segment          // text string, enclose
                text for second segment         // by " " if there are any
                  . . .                         // spaces in the text string
              text for last segment
```

```
    }

segment_annotate_value {                         // keyword
                 angle value  offset value  raise value
                 textstyle textstyle  slant degrees  xfactor value
                 worldsize value or papersize value or screensize value
                 justify  "top|middle|bottom-left|centre|right"
                 colour colour_name
    }
```

or

```
segment_annotate_data {                          // keyword
           properties { angle value  offset value  raise value
                    textstyle textstyle  slant degrees  xfactor value
                 worldsize value or papersize value or screensize value
                    justify  "top|middle|bottom-left|centre|right"
                    colour colour_name
           }
           properties { text properties second segment
           }
           properties {  ...
           }
           properties {  text properties for last segment
           }
    }
```

1.5.8.4.10 Symbols

There can be the same symbol (defined as a linestyle) for every vertex in the super string or a different symbol for each vertex of the super string. If a symbol does not have a colour, then it uses the string colour or the segment colour.

```
    symbol_value {                               // keyword
                 style linestyle_name  colour colour_name  size value
                 rotation value                    // in dms
                 offset value  raise value
    }
```

or

```
symbol_data {                                    // keyword
           properties { style linestyle_name  colour colour_name  size value
                 style linestyle  colour colour  size value
```

```
            rotation value                  // in dms
            offset value  raise value
        }
        properties {  symbol and properties for second vertex
        }
        properties {  ...
        }
        properties {  symbol and properties for last vertex
        }
}
```

1.5.8.4.11 Vertex Attributes

Each vertex can have one or more user defined named attributes.

```
vertex_attribute_data {                         // key word
        attributes {  attribute_type  attribute_name  attribute_value
                      attribute_type  attribute_name  attribute_value
                                 . . .
                      attribute_type  attribute_name  attribute_value
        }
        attributes {  named attributes for second vertex
        }
        attributes {  ...
        }
        attributes {  named attributes for last vertex
        }
}
```

1.5.8.4.12 Segment Attributes

Each segment can have one or more user defined named attributes.

```
segment_attribute_data {                        // keyword
        attributes {  attribute_type  attribute_name  attribute_value
                      attribute_type  attribute_name  attribute_value
                               . . .
                      attribute_type  attribute_name  attribute_value
        }
        attributes {  named attributes for second segment
        }
        attributes {  ...
```

```
}
attributes  {     named attributes for last segment
}
}
```

Continue to 1.5.9 Super Alignment String or return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.

1.5.9 Super Alignment String

In an *alignment* string, only the intersection point method (IP's) could be used to construct the horizontal and vertical geometry. The IP definition is actually a *constructive* definition and the tangents points and segments between the tangent points (lines, arcs, transitions *etc.*) are calculated from the IP definition. For an alignment string, only the IP definitions are included in the 12da file.

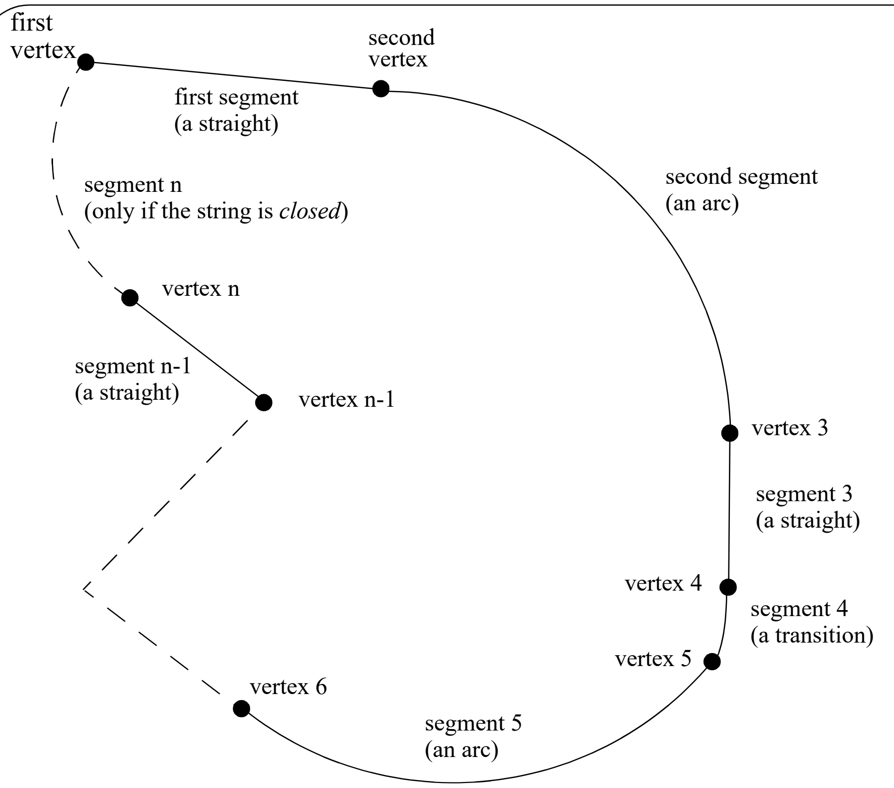
For a ***super alignment***, the horizontal and vertical geometry are also defined separately and with construction definitions but the construction definition can be much more complex than just IP's. For example, an arc could be defined as being tangential to two offset elements, or constrained to go through a given point.

If the horizontal construction methods are consistent then the horizontal geometry can be solved, and the horizontal geometry expressed in terms of consecutive segments (lines, arcs, transitions) that are easily understood and drawn.

Similarly if the vertical construction methods are consistent then the vertical geometry can be solved, and the vertical geometry expressed in terms of consecutive segments (lines, arcs, parabolas) that are easily understood and drawn.

Unlike the *alignment*, the *super alignment* stores ***both*** the ***construction methods*** (the ***parts***) and the resulting ***vertices*** and ***segments*** (lines, arcs, transitions *etc.*) that make up the horizontal and vertical geometry (the ***data***).

For many applications such as uploading to survey data collectors or machine control devices, only the ***horizontal data*** and the ***vertical data*** are required, not the *construction* methods (*i.e.* the ***horizontal*** and ***vertical parts***). When reading the 12da of a *super alignment*, only the ***horizontal*** and ***vertical data*** needs to be read in and the constructive methods (the ***horizontal*** and ***vertical parts***) can be skipped over.



Vertices and Segments Forming the Horizontal Data for a Super Alignment

Notes

1. Just using the horizontal and vertical data is valid ***as long as the super alignment geometry is consistent*** and **solves**, and the horizontal and vertical parts can be then created.

   There are flags in the 12da of the super alignment to say that the horizontal and vertical geometry is consistent and solves.

2. Segments meeting at a common vertex do not have to be tangential although for most road and rail applications, they should be.

The full 12da definition of the *super alignment* is:

```
string super_alignment {
//
    name              string_name
    chainage          start_chainage
    colour            colour_name
    style             style_name
    breakline         point or line
    closed            true or false
    spiral_type       transition_type          // the spiral_types are clothoid,
                                               // cubic parabola, westrail-cubic,
                                               // cubic spiral,
                                               // natural clothoid, bloss, sinusoidal
                                               // and cosinusoidal. Note that some
                                               // spiral_type's are non-spiral
                                               // transitions
    valid_horizontal  true or false            // if true then the horizontal geometry
                                               // is consistent and solves
    valid_vertical    true or false            // if true then the horizontal geometry
                                               // is consistent and solves

    block of info {
                 }
    block of info {
                 }
    block of info {
                 }

}                                              // end of super alignment
```

where the block of info can be one of more of:

attributes, horizontal_parts, horizontal_data, vertical_parts, vertical_data.

The *attributes* block has been described in the earlier section 1.3 Attributes.

The structure of the blocks *horizontal_parts*, *horizontal_data which define the horizontal geometry, and vertical_parts* and *vertical_data* which define the vertical geometry will now be described in more detail.

For information on *horizontal geometry,* go to 1.5.9.1 Horizontal Geometry
vertical geometry 1.5.9.2 Vertical Geometry

1.5.9.1 Horizontal Geometry

The horizontal geometry is described by two blocks - the ***horizontal_parts*** block and the ***horizontal_data*** block.

The ***horizontal_parts*** block contains the ***methods*** to construct the horizontal geometry such as float (fillet) an arc of a certain radius between two given lines or create a transition (spiral or non-spiral transition) between a line and an arc.

If the horizontal construction methods are consistent, then they can be solved to form a string made up of lines, arcs and transitions. The ***horizontal_data*** block is simply a list of the vertices and segments (lines, arcs *etc.*) that make up the ***solved*** geometry.

If the geometry in the *horizontal_parts* can be solved and produces a valid *horizontal_data* block, then the flag ***valid_horizontal*** in the super_alignment block is set to *true*.

```
valid_horizontal   true or false   //true if the horizontal geometry can be solved
                                   // and hence create a valid horizontal_data
horizontal_parts {/                // methods for creating the horizontal geometry
    ....
}
horizontal_data {                  // the horizontal segments that make up the
                                   // solved geometry
   ....
}
```

For information on *horizontal_parts,* go to the section 1.5.9.1.1 Horizontal_parts
horizontal_data 1.5.9.1.3 Horizontal_data

1.5.9.1.1 Horizontal_parts

The *horizontal_parts* block describes the methods used to construct the horizontal geometry of the super alignment. The parts that make up the horizontal geometry are defined in chainage order from the start to the end of the super alignment.

```
horizontal_parts {      // methods for creating the horizontal geometry
     blocks defining the sequential parts
     making up the horizontal geometry
}
```

Apart from the special case of parts defined by horizontal intersection points and their accompanying transitions and arcs, the other parts in the *horizontal_parts* block are not documented.

1.5.9.1.2 Horizontal_parts for defined by IP Method Only

For a horizontal intersection point (HIP) with no transitions or arc defined at that HIP, the part is defined by:

```
ip {
        id    value    // part id - a number that is unique for each horizontal and
                       // vertical part, and the value of part id is a multiple of 100
        x     value    // x coordinate of the horizontal intersection point
        y     value    // y coordinate of the horizontal intersection point
}
```

For a horizontal intersection point (HIP) with an arc but no transitions defined at that HIP, the part is defined by

```
arc {
        id    value    // part id - a number that is unique for each horizontal and
                       // vertical part, and the value of part id is a multiple of 100
        r     value    // radius of the arc at the HIP
        x     value    // x coordinate of the HIP
        y     value    // y coordinate of the HIP
}
```

For a horizontal intersection point (HIP) with an arc and transitions defined at that HIP, the part is defined by

```
spiral {
        id    value    // part id - a number that is unique for each horizontal and
                       // vertical part, and the value of part id is a multiple of 100
        r     value    // radius of the arc at the HIP
        l1    value    // length of the leading transition at the HIP
        l2    value    // length of the trailing transition at the HIP
        x     value    // x coordinate of the HIP
        y     value    // y coordinate of the HIP
```

```
    }
```

Note that the *transition* used in the ***spiral*** block is given by *spiral_type* in the super_alignment block.

Hence a super alignment with horizontal geometry defined by IP methods only would consist of a horizontal_parts section with only the above ip, arc and spiral blocks in it.

```
horizontal_parts {

  ip_spiral_arc {
                 values              // values defining the ip_spiral_arc block
                 "
                 values
               }
      ....
  ip_spiral_arc {
                 values              // values defining the ip_spiral_arc block
                 "
                 values
}
```

For example,

1st HIP
HIP only

Unique Part id
incrementing by 100

```
horizontal_parts {
  ip {
      id 100
      x 42606.66161172
      y 37239.28824481
  }
```

2nd HIP
HIP only

```
  ip {
      id 200
      x 43134.36832349
      y 37330.26705997
  }
```

3rd HIP
HIP with arc and
leading and trailing
transitions

```
  spiral {
      id 300
       r 50
       l1 30
       l2 40
       x 43336.6595
       y 37469.2563
  }
```

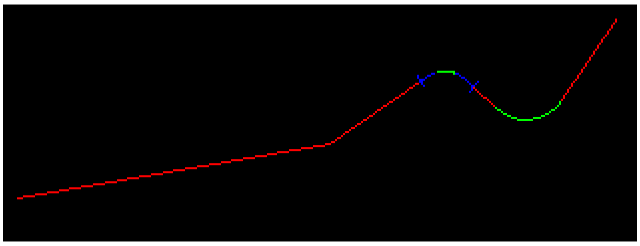
4th HIP
HIP with arc only

```
  arc {
      id 400
     r 75
     x 43481.15324268
     y 37331.6431906
  }
```

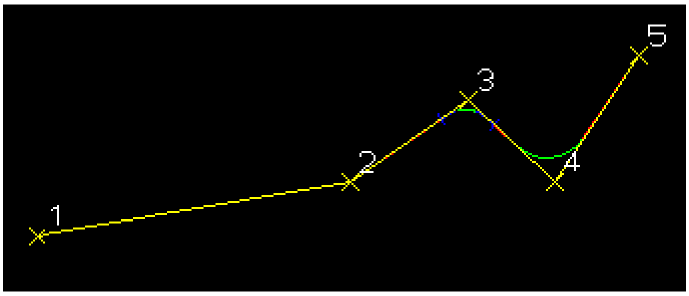
5th HIP
HIP only

```
  ip {
     id 500
     x 43627.02308964
     y 37544.94343852
  }
}
```

Plan View of Super Alignment



Super Alignment Being Edited



Horizontal Parts with IP Methods Only

1.5.9.1.3 Horizontal_data

The ***horizontal_data*** block contains the ***solved*** horizontal geometry of the super alignment.

The *solved horizontal geometry* is made up of a series of (x,y) vertices given in a *data_2d* block followed by a *geometry_data* block specifying the geometry of the segments between adjacent vertices. The segment can be a straight line, an arc, a transition (*e.g.* a spiral) a partial transition, an offset transition or a partial offset transition.

If the horizontal geometry has n vertices, then there will be (n-1) segments for an *open* super alignment or n segments if the super alignment is *closed*.

The format of the *horizontal_data* block is:

```
horizontal_data {
  name          ""
  chainage      value
  breakline     line or point
  colour        colour
  style         linestyle
  closed        0 or 1              // 0 if the string is open, 1 if it is closed

  interval {
    chord_arc     value             // chord-to-arc tolerance for curves
    distance      value             // chainage interval to break the geometry up
  }

  data_2d {
            x1-value    y1-value              // coordinates of the first vertex
            x2-value    y2-value              // coordinates of the second vertex
               "           "
               "           "
            xn-value    yn-value              // coordinates of the n-th vertex
  }

  geometry_data {
    segment_info_1 {
      information on the first segment
    }
    segment_info_2 {
      information on the second segment
    }
        "    "
        "    "
    segment_info_n-1 {              // the last segment if it is open
```

information on the (n-1) segment

```
    }
    segment_info_n {            // the last segment if it is closed
```

information on the n-th segment

```
    }
}
```

where the *segment_info* blocks are the same as for the ***geometry_data*** block in a super string. See 1.5.8.2 Blocks Defining the Geometry of the Segments.

1.5.9.2 Vertical Geometry

The *vertical* geometry is described by two blocks - the *vertical_parts* block and the *vertical_data* block.

The ***vertical_parts*** block contains the ***methods*** to construct the vertical geometry such as float (fit) a parabola of a certain length between two given lines.

If the vertical construction methods are consistent, then they can be solved to form a string made up of lines, parabolas and arcs. The ***vertical_data*** block is simply a list of the vertices and segments (lines, parabolas and arcs) that make up the *solved* geometry.

If the geometry in the *vertical_parts* can be solved and produces a valid *vertical_data* block, then the flag *valid_vertical* in the super_alignment block is set to *true*.

```
valid_vertical    true or false/     //true if the vertical geometry can be solved and
                                     // hence create a valid vertical_data
vertical_parts {                     // methods for creating the vertical geometry
     ....
}
vertical_data {                      // the vertical geometry
    ....
}
```

For information on *vertical_parts,* go to the section 1.5.9.2.1 Vertical_parts
vertical_data 1.5.9.2.3 Vertical_data

1.5.9.2.1 Vertical_parts

The *vertical_parts* block describes the methods used to construct the vertical geometry of the super alignment. The parts that make up the vertical geometry are defined in chainage order from the start to the end of the super alignment.

```
vertical_parts {                     // methods for creating the vertical geometry
      blocks defining the sequential parts
      making up the vertical geometry
}
```

Apart from the special case of parts defined by vertical intersection points and their accompanying parabolas and arcs, the other parts in the *vertical_parts* block are undocumented.

1.5.9.2.2 Vertical_parts When Defined by IP Method Only

For a vertical intersection point (VIP) with no parabola or arc defined at that VIP, the part is defined by:

```
ip {
        id   value    // part id - a number that is unique for each horizontal and
                      // vertical part, and the value of part id is a multiple of 100
        x    value    // chainage coordinate of the VIP
        y    value    // height coordinate of the VIP
}
```

For a vertical intersection point (VIP) with a parabola defined by a k value at that VIP, the part is defined by

```
kvalue {
        id   value    // part id - a number that is unique for each horizontal and
                      // vertical part, and the value of part id is a multiple of 100
        k    value    // k-value of the parabola at the VIP
        x    value    // chainage coordinate of the VIP
        y    value    // height coordinate of the VIP
}
```

For a vertical intersection point (VIP) with a parabola defined by length at that VIP, the part is defined by

```
length {
        id   value    // part id - a number that is unique for each horizontal and
                      // vertical part, and the value of part id is a multiple of 100
        l    value    // length of the parabola at the VIP
        x    value    // chainage coordinate of the VIP
        y    value    // height coordinate of the VIP
}
```

For a vertical intersection point (VIP) with a parabola defined by an effective radius at that VIP, the part is defined by

```
radius {
        id   value    // part id - a number that is unique for each horizontal and
                      // vertical part, and the value of part id is a multiple of 100
        r    value    // effective radius of the parabola at the VIP
        x    value    // chainage coordinate of the VIP
        y    value    // height coordinate of the VIP
}
```

For a vertical intersection point (VIP) with an asymmetric parabola defined by the start and end lengths at that VIP, the part is defined by

```
length {
        id    value   // part id - a number that is unique for each horizontal and
                      // vertical part,
                      // and the value of part id is a multiple of 100
        l1    value   // start length of the asymmetric parabola at the VIP
        l2    value   // end length of the asymmetric parabola at the VIP
        x     value   // chainage coordinate of the VIP
        y     value   // height coordinate of the VIP
}
```

For a vertical intersection point (VIP) with an arc defined by a radius at that VIP, the part is defined by

```
arc {
        id    value   // part id - a number that is unique for each horizontal and
                      // vertical part,
                      // and the value of part id is a multiple of 100
        r     value   // radius of the arc at the VIP
        x     value   // chainage coordinate of the VIP
        y     value   // height coordinate of the VIP
}
```

Hence a super alignment with vertical geometry defined by IP methods only would consist of a vertical_parts section with only the above ip, parabola and arc blocks in it.

```
vertical_parts {

  ip_parabola_arc {
                      values      // values defining the ip_parabola_arc block
                      "
                      values
  }
      ....
  ip_parabola_arc {
                      values      // values defining the ip_parabola_arc block
                      "
                      values
  }
}
```

For example,

```
vertical_parts {
  ip {
    id 600
    x -50.8459652
    y 159.79764161
  }
  kvalue {
    id 700
    k 1.25
    x 38.4627
    y 179.2126
  }
  length {
    id 800
    l 50
    x 172.61694837
    y 154.72967932
  }
  asymmetric {
    id 900
    l1 25
    l2 75
    x 270.0182
    y 208.1493
  }
  arc {
    id 1000
    r 1000
    x 424.2402
    y 196.5637
  }
  radius {
    id 1100
    r 200
    x 526.7263
    y 201.5302
  }
  ip {
    id 1200
    x 637.69216273
    y 198.71894484
  }
}
```

1st VIP
VIP only

Unique Part id
incrementing by 100

2nd VIP
Parabola defined
by k value

3rd VIP
Parabola defined
by length

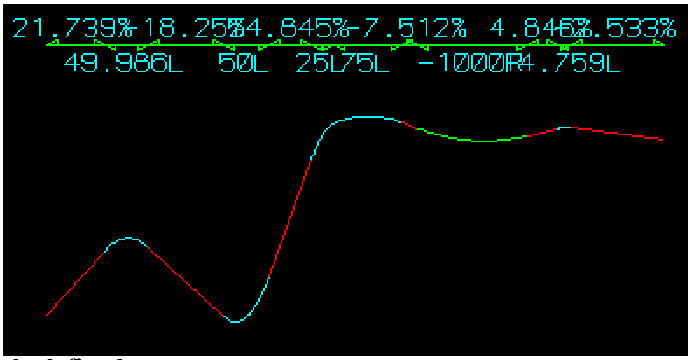
4th VIP
Asymmetric parabola defined
by two lengths

5th VIP
Arc with radius

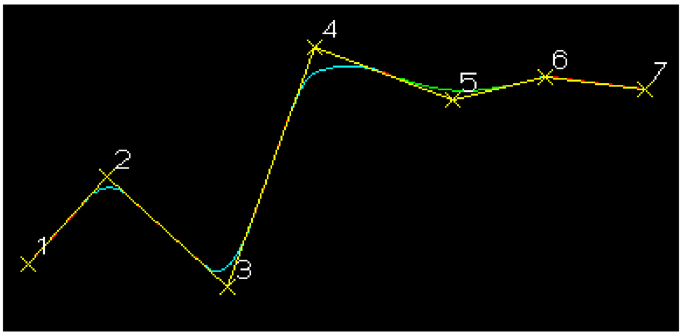
6th VIP
Parabola defined
by effective radius

7th VIP
VIP only

Section View of Super Alignment



Vertical Geometry Being Edited



Vertical Parts with IP Methods Only

1.5.9.2.3 Vertical_data

The ***vertical_data*** block contains the *solved* vertical geometry of the super alignment.

The *solved vertical geometry* is made up of a series of (chainage,height) vertices given in a *data_2d* block followed by a *geometry_data* block specifying the geometry of the segments between adjacent vertices. The segment can be a straight line, a parabola or an arc.

If the vertical geometry has n vertices, then there will be (n-1) segments for an *open* super alignment or n segments if the super alignment is *closed*.

The format of the *vertical_data* block is:

```
vertical_data {
  name        ""
  chainage    value
  breakline   line or point
  colour      colour
  style       linestyle
  closed      0 or 1              // 0 if the string is open, 1 if it is closed
  interval {
    chord_arc     value           // chord-to-arc tolerance for curves
    distance      value           // chainage interval to break the geometry up
  }

  data_2d {
           ch1-value   ht1-value        // coordinates of the first vertex
           ch2-value   ht2-value        // coordinates of the second vertex
               "           "
               "           "
           chn-value   htn-value        // coordinates of the n-th vertex
  }

  geometry_data {
    segment_info_1 {
      information on the first segment
    }
    segment_info_2 {
      information on the second segment
    }
        "    "
        "    "
    segment_info_n-1 {            // the last segment if it is open
      information on the (n-1) segment
    }
```

```
    segment_info_n {              // the last segment if it is closed
      information on the n-th segment
    }
  }
```

where the *segment_info* blocks are from the following:

(a) Straight

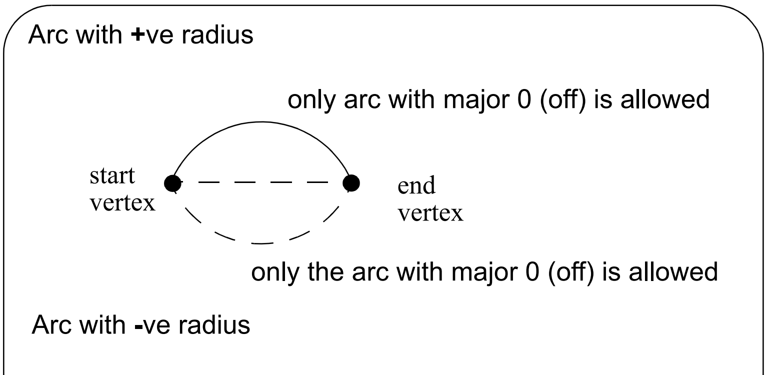
No parameters are needed for defining a straight segment. The *straight* block is simply:

```
straight {                        // no parameters are needed for a straight
}
```

(b) Arc

Since vertical geometry can't go backwards in chainage value, the majors arcs can not be used and hence there are only possibilities for an arc of a given radius placed between two vertices.

We use *positive* and *negative* radius to differentiate between the four possibilities.



Arcs with same absolute radius

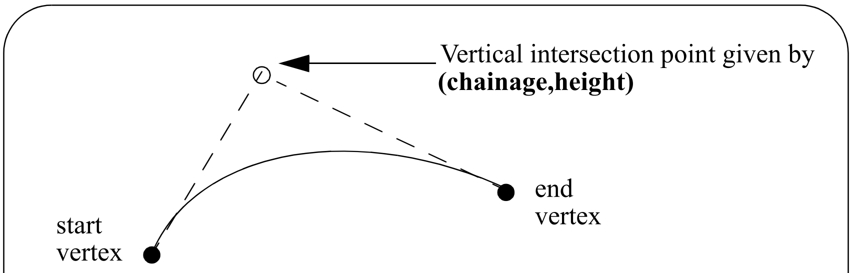
So the *arc* block is:

```
arc {
  radius value      // radius of the arc (+ve is above the line connecting vertices)
  major  value      // this is ignored since only minor arcs are used
}
```

(c) Parabola

There can be a parabola between adjacent vertices. The parabola is defined by giving the coordinates of the vertical intersection point for the parabola

chainage chainage of the VIP of the parabola
height height of the VIP of the parabola



Example of a Parabola

The parameters for the *parabola* block are:

`parabola {`		
`chainage`	*value*	// chainage of the VIP of the parabola
`height`	*value*	// height of the VIP of the parabola
`}`		

Continue to 1.5.10 Text String or return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.

1.5.10 Text String

```
string  text {
  x value   y value   z value
  model model_name   name string_name   colour colour_name
  text text_value
  angle value   offset value   raise value
  textstyle textstyle_name   slant degrees   xfactor value
  worldsize value or papersize value or screensize value
  justify  "top|middle|bottom-left|centre|right"
}
```

The string types in the following sections have been superceded.

Continue to 1.5.11 2d String or return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.

1.5.11 2d String

The 2d string has been superceded and has been replaced by the super string (see 1.5.8 Super String).

```
string 2d {
   z  value    chainage  start_chainage
   model  model_name  name  string_name
   colour  colour_name   style  style_name
   breakline  point or line
   data {                                        // keyword
      x-value   y-value
        "         "
        "         "
   }
}
```

Continue to 1.5.12 3d String or return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.

1.5.12 3d String

The 3d string has been superceded and has been replaced by the super string (see 1.5.8 Super String).

```
string 3d {
  chainage start_chainage
  model model_name   name   string_name
  colour colour_name  style style_name
  breakline point or line
  data {                                        // keyword
    x-value     y-value     z-value
       "           "           "
       "           "           "
  }
}
```

Continue to 1.5.13 4d String or return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.

1.5.13 4d String

The 4d string has been superceded and has been replaced by the super string (see 1.5.8 Super String).

```
string 4d {
  angle value   offset value   raise value
  worldsize value or papersize value or screensize value
  chainage start_chainage
  model model_name   name string_name
  colour colour_name   style style_name
  breakline point or line
  textstyle text   slant degrees   xfactor value
  justify   "top|middle|bottom-left|centre|right"
  data {                                    // keyword
    x-value   y-value   z-value   text      // text can not be blank
      "         "         "         "       // use "" for no text.
      "         "         "         "
  }
}
```

Continue to 1.5.14 Pipe String or return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.

1.5.14 Pipe String

The pipe string has been superceded and has been replaced by the super string (see 1.5.8 Super String).

```
string pipe {
  diameter value   chainage start_chainage
  model model_name   name string_name
  colour colour_name   style style_name
  breakline point or line
  data {                                        // keyword
    x-value  y-value  z-value
       "        "        "
       "        "        "
  }
}
```

Continue to 1.5.15 Polyline String or return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.

1.5.15 Polyline String

The polyline string has been superceded and has been replaced by the super string (see 1.5.8 Super String).

The definition of a closed string has been refined for polyline and super strings. For other string types, closing a string simply meant having the first vertex the same as the last vertex. Hence the vertex was duplicated.

For a polyline string, being closed is a property of the string and no extra vertex is needed - the first and the last vertices are not the same and the polyline string knows there is an additional segment from the last vertex back to the first vertex.

In the 12da format, there is a new *closed* flag for the polyline string:

closed *true* or *false*

where *true* can be 1 or T or t or Y or y (or words starting with T, t, Y or y))
and *false* is 0 or F or f or N or n (or words starting with F, f, N or n.

```
string polyline {
  chainage start_chainage
  model model_name   name string_name
  colour colour_name   style style_name
  breakline point or line
  closed true or false

  data {                                       // keyword
    x-value   y-value   z-value   radius  bulge_flag
       "         "         "
       "         "         "
  }
}
```

Continue to 1.5.16 Alignment String or return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.

1.5.16 Alignment String

The alignment string has been superceded and has been replaced by the super alignment (see 1.5.9 Super Alignment String).

In an alignment string the horizontal and vertical geometry are given separately and both can only be defined by the intersection point method (IP's).

For the horizontal geometry, the (x,y) position of the horizontal intersection points (HIPs) are given in the order that they appear in the string, plus the circular radius and left and right transition lengths on each HIP.

Hence a horizontal intersection point is given by either

x-value *y-value* *radius* `// circular curve, no transition`

or

x-value *y-value* *radius* `spil1` *left-transition-length* `spil2` *right-transition-length*

radius, left-transition-length, right-transition-length can be zero (meaning they don't exist).

For the vertical geometry, the (chainage,height) position of the vertical intersection points (VIPs) are given in increasing chainage order, plus either the radius of the circular arc or the length of the parabolic curve on each VIP.

Hence for a vertical intersection point is given by either

ch_value *z-value* *length* `parabola`

or

ch_value *z-value* *radius* `circle`

where

the word *parabola* is optional. *length* and *radius* can be zero, meaning that the parabola or arc doesn't exist.

```
string alignment {
  model model_name   name string_name
  colour colour_name   style style_name
  chainage start_chainage   interval value
  draw_mode value              // 1 to draw crosses at HIPs and VIPs, 0 don't draw
  spiral_type text             // spiral_type covers both spiral and non-spiral transitions.
                               // For an alignment string, the supported transition types
                               // are clothoid, cubic parabola, westrail-cubic, cubic spiral
                               // More transition are supported in the super alignment
                               //
  hipdata {                    //  some hips must exist and precede the VIP data
    x-value y-value   radius                                  // or
    x-value y-value   radius   spil1 left-transition-length   spil2   right-transition-length
       "        "        "        "    "                        "        "
  }
  vipdata {                                                   // vips optional
    ch_value   z-value   parabolic-length                     // or
    ch_value   z-value   parabolic-length        parabola     // or
```

```
        ch_value    z-value    radius              circle
          "           "          "          "

      }

    }
```

Continue to 1.5.17 Pipeline String or return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.

1.5.17 Pipeline String

The pipeline string has been superceded and has been replaced by the super alignment (see 1.5.9 Super Alignment String).

This is the same as an alignment string except that it has the additional keywords

diameter, which gives the diameter of the pipeline in world units

and

length of the typical pipe making up the pipeline (used for deflections).

```
string pipeline {
  model model_name   name string_name
  colour colour_name   style style_name
  diameter diameter   length pipe-length
  chainage start_chainage   interval value
  spiral_type text              // spiral_type covers both spiral and non-spiral transitions
                                // supported by 12d. For an alignment string, the
                                // supported transition types are clothoid, cubic parabola,
                                // westrail-cubic, cubic spiral. Other transition types
                                // are supported in the super alignment
  hipdata {                     // some hips must exist and precede vips
   x-value  y-value    radius                              // or
   x-value  y-value    radius    spil1  left-transition-length    spil2   right-transition-length
       "        "         "        "     "                    "         "
  }
  vipdata {                                               // vips optional
    ch-value    z-value    parabolic-length               // or
    ch-value    z-value    parabolic-length       parabola // or
    ch-value    z-value    radius      circle
        "          "          "          "
  }
}
```

Return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.

1.5.18 LAS Cloud String

```
string las_cloud_data {
  name                        // name
  colour                      // colour
  time_created text           // optional - time first created
  time_updated text           // optional - time last modified
  data_block or ref_data_block
}
```

The data block contains:

```
<data>
  category_block
  format_block
  range_block
  points_block
</data>
```

The category block contains **categories** tag and a list of boolean value (true or false).

```
categories {
  boolean_value boolean_value ... boolean_value
}
```

The range block contains four integer values.

```
range {
  xmin xmin_value
  xmax xmax_value
  ymin ymin_value
  ymax ymax_value
}
```

The format block is.

```
format format_name
```

Where *format_name* must come from the list

v10_p0 v10_p1

v11_p0 v11_p1

v12_p0 v12_p1 v12_p2 v12_p3

v13_p0 v13_p1 v13_p2 v13_p3

v14_p0 v14_p1 v14_p2 v14_p3 v14_p4 v14_p5 v14_p6 v14_p7 v14_p8 v14_p9 v14_p10

The points block must match the format given in the format block. For each format type vX_pY where X comes from the set: 10 11 12 13 14 and Y comes from the set 0 1 2 3 4 5 6 7 8 9 10; there are two choice of points data: **points_vX_pY** and **compact_points_vX_pY**.

```
points_vX_pY {
  point_pY
```

```
    point_pY
    ...
    ...
    point_yY
  }
  compact_points_vX_pY {
    compact_point_pY
    compact_point_pY
    ...
    ...
    compact_point_yY
  }
```

The point_p0 block is.

```
  p {
    x x_coordinate
    y y_coordinate
    z z_coordinate
    i intensity               \\ integer between 0 and 65535
    rn return_number          \\ integer between 0 and 7
    rc return_count           \\ integer between 0 and 7
    sd scan_direction         \\ integer between 0 and 1
    fe flight_line_edge       \\ integer between 0 and 1
    cl classification         \\ integer between 0 and 255
    sr scan_rank_angle        \\ integer between -128 and 127
    ud user_data              \\ integer between 0 and 255
    id point_source_id        \\ integer between 0 and 65535
  }
```

The compact_point_p0 block is the same as point_p0 but without any inner tag.

```
  p {
    x_coordinate
    y_coordinate
    z_coordinate
    intensity                 \\ integer between 0 and 65535
    return_number             \\ integer between 0 and 7
    return_count              \\ integer between 0 and 7
    scan_direction            \\ integer between 0 and 1
    flight_line_edge          \\ integer between 0 and 1
    classification            \\ integer between 0 and 255
```

```
    scan_rank_angle          \\ integer between -128 and 127
    user_data                \\ integer between 0 and 255
    point_source_id          \\ integer between 0 and 65535
  }
```

The point_p1 block is the same as point_p0 but with a time at the end.

```
  p {
    x x_coordinate
    y y_coordinate
    z z_coordinate
    i intensity              \\ integer between 0 and 65535
    rn return_number         \\ integer between 0 and 7
    rc return_count          \\ integer between 0 and 7
    sd scan_direction        \\ integer between 0 and 1
    fe flight_line_edge      \\ integer between 0 and 1
    cl classification        \\ integer between 0 and 255
    sr scan_rank_angle       \\ integer between -128 and 127
    ud user_data             \\ integer between 0 and 255
    id point_source_id       \\ integer between 0 and 65535
    t gps_time               \\ real number
  }
```

The compact_point_p1 block is the same as point_p1 but without any inner tag.

```
  p {
    x_coordinate
    y_coordinate
    z_coordinate
    intensity                \\ integer between 0 and 65535
    return_number            \\ integer between 0 and 7
    return_count             \\ integer between 0 and 7
    scan_direction           \\ integer between 0 and 1
    flight_line_edge         \\ integer between 0 and 1
    classification           \\ integer between 0 and 255
    scan_rank_angle          \\ integer between -128 and 127
    user_data                \\ integer between 0 and 255
    point_source_id          \\ integer between 0 and 65535
    gps_time                 \\ real number
  }
```

The point_p2 block is the same as point_p0 but with a colour (64bit integer) at the end.

```
  p {
```

```
    x x_coordinate
    y y_coordinate
    z z_coordinate
    i intensity              \\ integer between 0 and 65535
    rn return_number         \\ integer between 0 and 7
    rc return_count          \\ integer between 0 and 7
    sd scan_direction        \\ integer between 0 and 1
    fe flight_line_edge      \\ integer between 0 and 1
    cl classification        \\ integer between 0 and 255
    sr scan_rank_angle       \\ integer between -128 and 127
    ud user_data             \\ integer between 0 and 255
    id point_source_id       \\ integer between 0 and 65535
    c las_colour             \\ 64 bit integer
  }
```

The compact_point_p2 block is the same as point_p2 but without any inner tag.

```
  p {
    x_coordinate
    y_coordinate
    z_coordinate
    intensity                \\ integer between 0 and 65535
    return_number            \\ integer between 0 and 7
    return_count             \\ integer between 0 and 7
    scan_direction           \\ integer between 0 and 1
    flight_line_edge         \\ integer between 0 and 1
    classification           \\ integer between 0 and 255
    scan_rank_angle          \\ integer between -128 and 127
    user_data                \\ integer between 0 and 255
    point_source_id          \\ integer between 0 and 65535
    las_colour               \\ 64 bit integer
  }
```

The point_p3 block is the same as point_p1 but with a colour (64bit integer) at the end.

```
  p {
    x x_coordinate
    y y_coordinate
    z z_coordinate
    i intensity              \\ integer between 0 and 65535
    rn return_number         \\ integer between 0 and 7
    rc return_count          \\ integer between 0 and 7
```

```
    sd scan_direction          \\ integer between 0 and 1
    fe flight_line_edge        \\ integer between 0 and 1
    cl classification          \\ integer between 0 and 255
    sr scan_rank_angle         \\ integer between -128 and 127
    ud user_data               \\ integer between 0 and 255
    id point_source_id         \\ integer between 0 and 65535
    t gps_time                 \\ real number
    c las_colour               \\ 64 bit integer
  }
```

The compact_point_p3 block is the same as point_p3 but without any inner tag.

```
  p {
    x_coordinate
    y_coordinate
    z_coordinate
    intensity                  \\ integer between 0 and 65535
    return_number              \\ integer between 0 and 7
    return_count               \\ integer between 0 and 7
    scan_direction             \\ integer between 0 and 1
    flight_line_edge           \\ integer between 0 and 1
    classification             \\ integer between 0 and 255
    scan_rank_angle            \\ integer between -128 and 127
    user_data                  \\ integer between 0 and 255
    point_source_id            \\ integer between 0 and 65535
    gps_time                   \\ real number
    las_colour                 \\ 64 bit integer
  }
```

The point_p4 block is the same as point_p1 but with a wave data at the end (not yet implemented).

The compact_point_p4 block is the same as point_p4 but without any inner tag.

The point_p5 block is the same as point_p3 but with a wave data at the end (not yet implemented).

The compact_point_p5 block is the same as point_p5 but without any inner tag.

The point_p6 block is.

```
  p {
    x x_coordinate
    y y_coordinate
    z z_coordinate
    i intensity                \\ integer between 0 and 65535
    rn return_number           \\ integer between 0 and 15
    rc return_count            \\ integer between 0 and 15
```

```
    cf classification_flags    \\ integer between 0 and 15
    sc scanner_channel         \\ integer between 0 and 3
    sd scan_direction          \\ integer between 0 and 1
    fe flight_line_edge        \\ integer between 0 and 1
    cl classification          \\ integer between 0 and 255
    ud user_data               \\ integer between 0 and 255
    sr scan_rank_angle         \\ integer between -128 and 127
    id point_source_id         \\ integer between 0 and 65535
    t gps_time                 \\ real number
  }
```

The compact_point_p6 block is the same as point_p6 but without any inner tag.

```
  p {
    x_coordinate
    y_coordinate
    z_coordinate
    intensity                  \\ integer between 0 and 65535
    return_number              \\ integer between 0 and 15
    return_count               \\ integer between 0 and 15
    classification_flags       \\ integer between 0 and 15
    scanner_channel            \\ integer between 0 and 3
    scan_direction             \\ integer between 0 and 1
    flight_line_edge           \\ integer between 0 and 1
    classification             \\ integer between 0 and 255
    user_data                  \\ integer between 0 and 255
    scan_rank_angle            \\ integer between -128 and 127
    point_source_id            \\ integer between 0 and 65535
    gps_time                   \\ real number
  }
```

The point_p7 block is the same with point_p6 with a las colour (64bit integer) at the end.

```
  p {
    x x_coordinate
    y y_coordinate
    z z_coordinate
    i intensity                \\ integer between 0 and 65535
    rn return_number           \\ integer between 0 and 15
    rc return_count            \\ integer between 0 and 15
    cf classification_flags    \\ integer between 0 and 15
    sc scanner_channel         \\ integer between 0 and 3
```

```
    sd scan_direction          \\ integer between 0 and 1
    fe flight_line_edge        \\ integer between 0 and 1
    cl classification          \\ integer between 0 and 255
    ud user_data               \\ integer between 0 and 255
    sr scan_rank_angle         \\ integer between -128 and 127
    id point_source_id         \\ integer between 0 and 65535
    t gps_time                 \\ real number
    c las_colour               \\ 64bit integer
  }
```

The compact_point_p7 block is the same as point_p7 but without any inner tag.

```
  p {
    x_coordinate
    y_coordinate
    z_coordinate
    intensity                  \\ integer between 0 and 65535
    return_number              \\ integer between 0 and 15
    return_count               \\ integer between 0 and 15
    classification_flags       \\ integer between 0 and 15
    scanner_channel            \\ integer between 0 and 3
    scan_direction             \\ integer between 0 and 1
    flight_line_edge           \\ integer between 0 and 1
    classification             \\ integer between 0 and 255
    user_data                  \\ integer between 0 and 255
    scan_rank_angle            \\ integer between -128 and 127
    point_source_id            \\ integer between 0 and 65535
    gps_time                   \\ real number
    las_colour                 \\ 64bit integer
  }
```

The point_p8 block is the same with point_p7 with a near infrared (integer between 0 and 255) at the end.

```
  p {
    x x_coordinate
    y y_coordinate
    z z_coordinate
    i intensity                \\ integer between 0 and 65535
    rn return_number           \\ integer between 0 and 15
    rc return_count            \\ integer between 0 and 15
    cf classification_flags    \\ integer between 0 and 15
    sc scanner_channel         \\ integer between 0 and 3
```

```
    sd scan_direction          \\ integer between 0 and 1
    fe flight_line_edge        \\ integer between 0 and 1
    cl classification          \\ integer between 0 and 255
    ud user_data               \\ integer between 0 and 255
    sr scan_rank_angle         \\ integer between -128 and 127
    id point_source_id         \\ integer between 0 and 65535
    t gps_time                 \\ real number
    c las_colour               \\ 64bit integer
    ir near_infrared           \\ integer between 0 and 255
  }
```

The compact_point_p8 block is the same as point_p8 but without any inner tag.

```
  p {
    x_coordinate
    y_coordinate
    z_coordinate
    intensity                  \\ integer between 0 and 65535
    return_number              \\ integer between 0 and 15
    return_count               \\ integer between 0 and 15
    classification_flags       \\ integer between 0 and 15
    scanner_channel            \\ integer between 0 and 3
    scan_direction             \\ integer between 0 and 1
    flight_line_edge           \\ integer between 0 and 1
    classification             \\ integer between 0 and 255
    user_data                  \\ integer between 0 and 255
    scan_rank_angle            \\ integer between -128 and 127
    point_source_id            \\ integer between 0 and 65535
    gps_time                   \\ real number
    las_colour                 \\ 64bit integer
    near_infrared              \\ integer between 0 and 255
  }
```

The point_p9 block is the same as point_p6 but with a wave data at the end (not yet implemented).

The compact_point_p9 block is the same as point_p9 but without any inner tag.

The point_p10 block is the same as point_p8 but with a wave data at the end (not yet implemented).

The compact_point_p10 block is the same as point_p10 but without any inner tag.

The ref_data block contains:

```
  ref_data {
    category_block   // same as category in data block
    file_name las_ref_file_name
```

```
  range_block       // same as range in data block
}
```

Return to 1.5 12da Definition for each String Type or 1 12d Archive File Format.