CAM System Customization Reference

This folder holds developer / integrator documentation for customizing the CAM system by editing its declarative XML — adding your own operations, machines and defaults — together with related topics. It is intended for OEMs, integrators and partners, and for advanced users who customize the system.

Everything here is built on one foundation — the XMLProperties framework, which describes the system's configurable objects declaratively. Start with the framework (it also covers the shared ground rules every area relies on), then move on to the operation- and machine-specific areas.

Documentation areas

- XMLProperties framework — start here: the declarative property framework itself (mental model, descriptor syntax, expression language, using properties from code, user defaults) and the shared ground rules every area relies on — why you don't edit `Supplement`, where the shipped files live, and the conventions used in the examples.
- Operation XML descriptors — describing and registering technological operations, plus the full operation inheritance tree.
- Machine descriptors — describing a machine: its file layout, kinematic scheme, *Machine Setup* parameters, advanced topics and 3D models.

The `Supplement/docs` folder may also host unrelated documentation; everything about the XMLProperties framework and its descriptors lives under the areas listed above.

XMLProperties framework

> XML paths in this article refer to configuration files in the installed CAM system (`$(SCHEMAS_FOLDER)`), not manual attachments. Available files depend on the version and configuration.

The XMLProperties framework is how the CAM system describes its configurable objects (operations, machines, tools, ⋯) declaratively. Instead of hard-coding the parameters of a milling operation or a machine kinematic scheme, the system loads a tree of property descriptors from XML. Each descriptor carries a name, a value type, a default value, display metadata, and optional rules (visibility, enable/disable, computed values). The running application turns this tree into the inspector UI, the data model behind every project, and the values exposed through the CAM API. Because the model is declarative, you can add a parameter, re-group the inspector, change a default, or describe a brand-new machine — all by editing XML, without recompiling the application.

This area documents the framework itself — the part common to *every* kind of descriptor. The operation- and machine-specific guides build on it: Operations, Machines.

Read top to bottom — each page builds on the previous one:

- Framework overview — the mental model: types vs. instances, namespaces, how files are loaded and combined, and where your own files go.
- Descriptor syntax — the core reference: `SCType`, `SCProperty`, the type system, inheritance and overriding, and the full attribute catalog.
- Expression language — computed values for `DefaultValue`, `Visible`, `Enabled` and friends: operators, functions, property references and accessors.
- Using XML properties from code — the one access pattern shared by the CAM API, native Delphi/C++, and postprocessors (.NET & `.sppx`).
- User defaults — overriding the shipped *default values* with your own preferred defaults, stored in `.usrdef` files and applied without editing `Supplement`.

(See the documentation root for the intended audience.)

The ground rules

These apply across *every* area — operations and machines included — so read them once here.

> You do not edit the **`Supplement`** folder. The files under `Supplement` are the system's distribution files and are replaced on update. Your custom content goes into separate, dedicated folders that the system loads automatically. The `Supplement` files are still your best *reference* — read them to learn the patterns — but author your own files elsewhere. See Where your custom files go.

Where the shipped files live

The shipped descriptor files live under the program installation root, for example:

```
C:\Program Files\<Vendor>\<Product>\Supplement\
```

These are read-only system files — use them as a reference, but do not modify them. Loading starts from a small set of root files under `Supplement`, each of which pulls in the rest with `<SCInclude>` — including the optional, wildcarded extension folders where your own files are picked up:

Root file	Purpose
`SCConfig.xml`	Master entry point — common types, operations, project schema, user defaults
`Machines/MachinesConfig.xml`	Machine descriptors
`STApplicationConfig.xml`	Application-level settings
`Postprocessor/PostprocessorConfig.xml`	Postprocessor descriptors

Conventions used in the examples

Short *synthetic* snippets (e.g. types named `TFoo`, `Bar`) illustrate pure syntax. *Real* fragments quoted from the shipped `Supplement` files show how the rules are used in production; these link back to the source file so you can see the full context. File paths inside each guide are written relative to the `Supplement` folder.

What this framework underlies

Operations and machines (documented in their own areas) are the main things you customize with this framework. Other descriptor families — tools, the postprocessor configuration and the application configuration — are built on the same framework, but they are internal serialization data, maintained by the system's developers and not intended for customization; they are therefore not documented in detail.

Framework overview

> XML paths in this article refer to configuration files in the installed CAM system (`$(SCHEMAS_FOLDER)`), not manual attachments. Available files depend on the version and configuration.

This chapter gives you the mental model behind the XMLProperties framework. Once you understand the four core ideas below, the rest of the syntax is detail.

1 What the framework is for

Every configurable object in the CAM system — a technological operation, a machine, a tool, a project — is described by a tree of properties. A property has a name, a value, a type, a default, and metadata that tells the UI how to present it and tells the engine how to behave.

These property trees are not written in code. They are declared in XML descriptor files and loaded at startup. This means you can:

- add or remove parameters on an operation or machine,
- change default values (including separate metric / inch defaults),
- reorganize how parameters are grouped and shown in the inspector,
- make a parameter appear, disappear, enable or disable depending on other parameters,
- define an entirely new operation or machine,

all by editing XML.

2 Types vs. instances

This is the single most important distinction in the framework.

- A descriptor (type) is a *template*. It says "a `RoughPasses` block has a boolean `Enabled` defaulting to `true` and a `Step` of `1 mm`." Descriptors are declared with `<SCType>` (and, at the top level, `<SCProperty>`). They form an inheritance chain: a type can be based on another type and extend or override it.

- An instance (property pointer) is an *actual value* attached to a concrete object — for example, the `RoughPasses.Step` of one particular facing operation in your project. Instances are created from descriptors when an object is created, and they hold the real, editable data.

You author descriptors. The application creates instances from them. When you read or write values through the CAM API (see Using XML properties from code), you work with instances.

A key consequence: a property instance whose value still equals its descriptor's default is *not* stored verbatim — it is computed from the descriptor on demand. Changing a default in XML therefore affects every existing instance that hasn't been explicitly edited.

3 The shape of a descriptor tree

Descriptors nest. A `ComplexType` descriptor contains child descriptors; those children may themselves be complex; arrays contain a single child that acts as the element template; enumerations contain their items as children. So a single operation descriptor is a deep tree whose leaves are simple values (numbers, strings, booleans, enum selections).

```
<SCType ID="TFinishPasses" type="ComplexType">
    <SCType ID="Enabled" type="Boolean" DefaultValue="true"/>
    <SCType ID="Step"    type="Double"  DefaultValue="1" DimensionKind="Linear"/>
    <SCType ID="Count"   type="Integer" DefaultValue="1"/>
</SCType>
```

Navigation through the tree later (in expressions and in the API) uses dotted paths like `FinishPasses.Step`.

4 Namespaces

Descriptors are grouped into namespaces. A namespace is a self-contained dictionary of type names: the `TSTFaceMillingOp` operation type lives in the `Operations` namespace, a machine type lives in the `Machines` namespace, and so on. Two namespaces may contain types with the same name without clashing.

Namespaces are declared with `<SCNameSpace ID="...">`. The same physical type file is often included into more than one namespace (for example, the shared `CommonTypes.xml` is included almost everywhere), so the common building blocks are available wherever they are needed.

The main namespaces you will encounter:

Namespace	Contents
`Operations`	Technological operation types
`Machines`	Machine types and their kinematic building blocks
`Project`	Project / setup schema
`Application`	Application-level settings

Namespace	Contents
`Postprocessor`	Postprocessor descriptors
`OperationRegistrator`	Small "registration" records that announce which types are real operations (see Operation descriptors)

5 How files are loaded and combined

Loading begins at a root file (such as `SCConfig.xml`) and follows `<SCInclude>` directives to assemble the whole model. Three rules matter:

1. Order is significant. Files are processed top to bottom. A type must be declared before it is used as a base type, and later declarations can override earlier ones. This is why the root files include abstract/base types before the concrete types that derive from them.
2. Includes can be optional and wildcarded. `Optional="True"` means "load this if it exists, otherwise skip silently"; wildcards like `*_ExtOp.xml` let you drop extension files into a folder without editing the root config. Several of these optional, wildcarded includes point at folders outside `Supplement` — that is exactly how your custom content is loaded without touching the system files. See §7.
3. Paths use folder variables. Instead of absolute paths, includes (and file references such as icons) use `$(VARIABLE)` placeholders that resolve to real folders at runtime. See §7 for the list.

```
<!-- from SCConfig.xml -->
<SCInclude>$(SUPPLEMENT_FOLDER)\CommonTypes.xml</SCInclude>
<SCInclude>$(SUPPLEMENT_FOLDER)\Operations.xml</SCInclude>
<!-- optional, wildcarded, loaded only if present -->
<SCInclude Optional="True">$(LOCAL_OPERATIONS_FOLDER)\*.usrdef</SCInclude>
```

> Not every namespace has a root file on disk. Assembling a namespace is just "open the namespace, load a sequence of files into it, close it." That sequence is usually declared in XML (`<SCNameSpace>` + `<SCInclude>`), but the application can also build a namespace in memory at startup by loading the same building-block files in order. The `Project` namespace is built this way — there is no single project root config file; the system loads `CommonTypes.xml`, `ProjectSchema/CommonProjectSchema.xml`, `ProjectSchema/ProjectSchema.xml` and a few others into a fresh `Project` namespace. The result is identical to the declarative form — so if you can't find a root file for a namespace, that is why.

6 What the framework drives

A single descriptor tree feeds several subsystems at once. Understanding this explains why a descriptor carries so many different attributes:

- The data model — the actual values stored in a project.
- The inspector UI — labels, grouping, ordering, icons, which rows are visible/enabled, and special editors. Most "presentation" attributes (`Caption`, `Visible`, `Compact`, `Category`, `ImageFile`, ⋯) exist for this.
- The processing engine — solvers and the toolpath/NC pipeline read property values by name.
- The CAM API — external code reads and writes the same properties.

The next chapter is the reference for the syntax that all of this is built on.

7 Where your custom files go

The `Supplement` folder is read-only system content — it is overwritten on every update, so any change you make there is lost. Instead, the root configs include a number of *optional, wildcarded* locations outside `Supplement`, reserved for your content. Put your files there and they are loaded automatically alongside the system descriptors, in the right order.

The locations are addressed through path variables so they resolve correctly per installation and per user:

You want to add⋯	How	Loaded by
A new / extended operation type (automatic by name)	drop an `*_ExtOp.xml` file in `$(COMMON_CONTAINERS_FOLDER)`	`Operations.xml`
A new operation type (registered via wizard)	register your operation XML file with the *Operations Manager* (see below) → it is added to `$(OPERATIONS_FOLDER)\UserOperationsList.xml`	`Operations.xml`
User default overrides	place `*.usrdef` files in `$(LOCAL_OPERATIONS_FOLDER)`	`SCConfig.xml`
A custom machine	build it with the machine-building tools; stored in `$(SCHEMAS_FOLDER)` / `$(CUSTOM_SCHEMAS_FOLDER)`	the machine-building subsystem

Guidelines:

- Operations — two ways. There are two independent mechanisms for adding a custom operation, and both load *after* the system operations, so either can declare new operation types or override existing ones:
  - By naming convention. Drop an `*_ExtOp.xml` file into `$(COMMON_CONTAINERS_FOLDER)`. Any file matching the mask is picked up automatically — nothing else to configure.
  - Via the Operations Manager wizard (menu Utilities → Operations Manager). You point the wizard at your operation's XML file (kept wherever you like); it registers that file by adding an entry to `$(OPERATIONS_FOLDER)\UserOperationsList.xml`. That list file is *maintained by the wizard* — you don't hand-edit it.

  In both cases the operation file itself is an ordinary descriptor file (an `<SCCollection>` in the `Operations` namespace); only how it gets registered differs.

- Machines. Custom machines are normally created interactively with the MachineMaker application, which writes their kinematic schemes to the machine schema folders — not to `Supplement`. Hand-written XML is only for special cases; the machine descriptor syntax in the Machine descriptors guide is what those schemes are built from.

- Read, don't edit, **`Supplement`**. Treat the shipped files purely as a reference for the patterns and base types you build on.

The mechanics are identical wherever the file lives: same `<SCCollection>` root, same namespaces, same syntax. Only the *folder* differs.

Next: Descriptor syntax

Descriptor syntax

> XML paths in this article refer to configuration files in the installed CAM system (`$(SCHEMAS_FOLDER)`), not manual attachments. Available files depend on the version and configuration.

This is the core reference chapter. It covers the XML elements, the type system, how inheritance and overriding work, and the catalog of attributes you can put on a descriptor.

1 The document skeleton

Every descriptor file is an XML document with `<SCCollection>` as its root element. Inside it you place namespaces, includes, type declarations and property declarations.

```
<?xml version="1.0" encoding="UTF-8" ?>
<SCCollection>
    <SCNameSpace ID="Operations">
        <SCInclude>$(SUPPLEMENT_FOLDER)\CommonTypes.xml</SCInclude>
        <SCType ID="TFoo" type="ComplexType"> ... </SCType>
    </SCNameSpace>
</SCCollection>
```

The structural elements are:

Element	Role
`SCCollection`	Document root. Always present.
`SCNameSpace`	Opens a namespace (`ID="Operations"`, `"Machines"`, ⋯). Everything inside is declared in that namespace. May be nested.
`SCInclude`	Pulls in another file at this point. Supports `Optional="True"` and wildcards.
`SCType`	Declares a type (a reusable descriptor) — or, when nested inside another type, declares a member of it. The workhorse element.
`SCProperty`	Declares a named global property — a singleton value/object at namespace scope (see §6).

Files outside any `<SCNameSpace>` contribute to the namespace into which they are included — that is why `CommonTypes.xml` has no namespace of its own: it is included into many.

2 `SCType` — the workhorse

`SCType` does double duty depending on where it appears.

At the top level (or directly under a namespace) it *declares a type* — a named, reusable template:

```
<SCType ID="T2DPoint" Caption="Point" type="ComplexType">
    <SCType ID="X" Caption="X" type="Double" DefaultValue="0"
DimensionKind="Linear"/>
    <SCType ID="Y" Caption="Y" type="Double" DefaultValue="0"
DimensionKind="Linear"/>
</SCType>
```

`ID` is the type's name; `type` names its base (here the built-in `ComplexType`). The nested `<SCType>` elements declare the type's *members*.

Nested inside another type an `<SCType>` *declares a new member* of the enclosing type. The member's `ID` is its name within the parent, and its `type` is the member's value type — which may be a built-in type or any previously declared type:

```
<SCType ID="TSTFaceMillingOp" type="TSTMillOp">
    <!-- a member named MillingType, of the previously declared type TMillMode -->
    <SCType ID="MillingType" type="TMillMode" Caption="Milling type"/>
</SCType>
```

So: **`ID`** + **`type`** = "create a thing called *ID* that is a *type*", whether that "thing" is a top-level type or a member of a complex type.

3 The type system

3.1 Built-in (primitive) types

The `type` attribute accepts these built-in names (case is not significant — `Double` and `double` are the same):

Built-in **`type`**	Holds	Notes
`Boolean`	`True` / `False`	
`Integer`	whole number	

Built-in `type`	Holds	Notes
`Double`	real number	Use `DimensionKind` for unit handling
`String`	text	
`Enumerated`	one choice from a fixed list	Items are child `SCType`s of `type="none"`
`ComplexType`	a record of named members	Children are the members
`Array`	a list of like elements	One child defines the element template; see §5
`none` (or `None`)	nothing	Used for enumeration items, separators and pure UI placeholders

Anything that is *not* one of these names is treated as a reference to a previously declared type — that is how composition and inheritance work.

3.2 Enumerations

An enumeration lists its options as child `SCType`s with `type="none"`. The enumeration's `DefaultValue` is the `ID` of the default option.

```
<SCType ID="TXYZCoordinate" Caption="XYZ Coordinate" type="Enumerated"
DefaultValue="CoordX">
    <SCType ID="CoordAuto" Caption="Auto" type="none"/>
    <SCType ID="CoordX"    Caption="X"    type="none"/>
    <SCType ID="CoordY"    Caption="Y"    type="none"/>
    <SCType ID="CoordZ"    Caption="Z"    type="none"/>
</SCType>
```

(real example: `CommonTypes.xml`)

The stored value of an enum property is the option **`ID`** (e.g. `CoordX`), not its caption. Each option may carry its own metadata (a `Caption`, an `ImageFile`, a `Priority`, a `Visible` rule, …).

3.3 Complex types

A `ComplexType` is a record: its child `SCType`s are its fields. Complex types can nest arbitrarily and can be reused as the type of other members.

```
<SCType ID="TColor" Caption="Color" type="ComplexType">
    <SCType ID="R" type="Double" DefaultValue="0.5"/>
    <SCType ID="G" type="Double" DefaultValue="0.5"/>
    <SCType ID="B" type="Double" DefaultValue="0.5"/>
</SCType>
```

In the inspector a `TColor` is rendered as a colour swatch with a picker dialog rather than three numeric rows — see §3.4.

3.4 Types with special behaviour

A handful of frequently used types are declared in `CommonTypes.xml` as thin aliases of a base type — you reference them by name like any declared type — but each adds extra serialization or inspector behaviour beyond its base. Declare a member with one of these when you want that behaviour:

Type	Base	What it adds
`TranslatableString`	`String`	Its value enters the localization subsystem and can be translated. By default only node `Caption`s are localized; use this type when a string *value* (typically a default value) must be translatable too.
`CDATAString`	`String`	Serialized inside a `CDATA` section, so the value may contain characters XML would otherwise forbid — letting it keep its original formatting, such as line breaks and tabs.
`FileName`	`String`	Holds a file path. On save/load the path is automatically converted between absolute and relative form and against path variables such as `$(SUPPLEMENT_FOLDER)`. In the inspector it offers a standard file-open dialog; a `Filter` attribute (form `Caption (*.ext)|mask1;mask2`)

Type	Base	What it adds
		restricts that dialog to chosen file types.
`DynamicArray`	`Array`	Behaves like an `Array` (§5), but the inspector shows the element count in the row's own value field and lets the user change it, adding or removing child elements on the fly.
`TColor`	`ComplexType`	A colour value (`R`/`G`/`B`, §3.3) rendered in the inspector as a colour swatch with a colour-picker dialog.

(real example: `CommonTypes.xml`. A `FileName` dialog filter — the machine postprocessor field — in `Machines/AbstractMachine.xml`: `Filter="Postprocessors (*.sppx, *.dll)|*.sppx;*.dll"`.)

4 Inheritance and overriding

4.1 Deriving a type

To base a new type on an existing one, name the existing type in `type`. The new type starts with all the members of the base and then adds the members you declare:

```
<SCType ID="T3DPoint" Caption="Point" type="T2DPoint">
    <!-- inherits X and Y from T2DPoint, adds Z -->
    <SCType ID="Z" type="Double" DefaultValue="0" DimensionKind="Linear"/>
</SCType>
```

(real example: `CommonTypes.xml`)

Inheritance chains can be long. A concrete operation, for instance, derives from a family of abstract operations (`TSTFaceMillingOp` → `TSTMillOp` → ⋯ → `TOperationDescriptor`).

4.2 Overriding an inherited member

There are two different things you can write inside a type body, and the difference is essential:

- `<SCType ID="Name" type="...">` — declares a *new* member called `Name`.
- `<Name ... />` — overrides the *inherited* member called `Name`. The tag *is* the member's name; you do not repeat `type`, you only restate the attributes you want to change.

```
<SCType ID="ZCleanup" type="TZCleanup" Caption="Cleanup height">
    <!-- override inherited members' defaults: -->
    <Enabled DefaultValue="True"/>
    <Height  DefaultValue="2"/>
</SCType>
```

(real example: `Operations/MillOperations/FaceMillingOp.xml`)

Overrides can reach down into nested members and even into enumeration options. The pattern `<EnumItem Visible="True"/>` re-enables an option that a base type had hidden:

```
<CirclesDivision DefaultValue="Halves">
    <AbsQuadrants Visible="True"/>   <!-- show an option the base hid -->
    <AbsHalves    Visible="True"/>
</CirclesDivision>
```

(real example: `Machines/AbstractMachine.xml`)

4.3 Metric and inch defaults

Numeric properties can declare two defaults: `DefaultValue` for the metric unit system and `InchDefaultValue` for the imperial one. The system picks the right one according to the active unit system — they are *independent values*, not an automatic unit conversion, so set each to a sensible round number.

```
<SCType ID="Outer" type="Double" DefaultValue="0.02" InchDefaultValue="0.001"
        DimensionKind="Linear"/>
```

(real example: `Operations/AbstractOP.xml`)

5 Arrays

An `Array` (or the predefined `DynamicArray`) holds a variable number of elements of one type. You declare the element template as the array's single child:

```
<SCType ID="TStringList" Caption="List of strings" type="Array">
    <SCType ID="L" Caption="Line" type="String" DefaultValue=""/>
</SCType>
```

For arrays whose elements need a stable identity (so the application can match elements across edits), set `CollectionKeyField` to the name of the member that holds the key:

```
<SCType ID="List" type="Array" CollectionKeyField="Name">
    <SCType ID="Script" type="TScript"/>
</SCType>
```

(real example: `Operations/AbstractOP.xml`)

Inside an array element, the running index is available to expressions through `Attribute(ItemIndex)`, which is handy for auto-numbered captions (see the expression language):

```
<SCType ID="J23ValuePair" Caption="Point [Attribute(ItemIndex)]"
type="TJ23ValuePair"/>
```

6 `SCProperty` — global named properties

While `SCType` declares *templates*, `SCProperty` declares a named global property: a single, addressable value or object at namespace scope. These act as shared constants/objects that expressions elsewhere can reference by name.

```
<SCProperty ID="MetricMeasurementsSystem" type="TMeasurementsSystem">
    <LinearUnits DefaultValue="Millimeters"/>
    <CuttingSpeedUnits DefaultValue="MetersPerMinute"/>
</SCProperty>

<!-- a computed global string that derives its value from another global -->
<SCProperty ID="LinearUnits$" type="String"
            DefaultValue="
[CurrentMeasurementsSystem.LinearUnits.EnumValue.Attribute(Caption)]"/>
```

(real example: `CommonTypes.xml`)

> The **`$`** naming convention. A trailing `$` in a property `ID` (e.g. `LinearUnits$`) is a *convention* marking a derived, display-oriented string property. The `$` is simply part of the name; it has no special parser meaning.

7 Path variables

File references — both `<SCInclude>` targets and attributes like `ImageFile`, `Icon`, `SPPFile` — use `$(VARIABLE)` placeholders instead of absolute paths. They resolve at runtime relative to the installation and the user profile.

Variable	Resolves to (typical)
`$(SUPPLEMENT_FOLDER)`	`<install root>\Supplement`
`$(MACHINES_FOLDER)`	per-user `…\Machines`
`$(SCHEMAS_FOLDER)`	`$(MACHINES_FOLDER)\Schemas`
`$(OPERATIONS_FOLDER)`	per-user `…\Operations`
`$(LOCAL_OPERATIONS_FOLDER)`	local user operations folder
`$(COMMON_CONTAINERS_FOLDER)`	shared containers/extensions folder
`$(PROGRAM_PERSONAL)`	per-user program data

Always use these variables rather than hard-coded paths, so your files keep working regardless of where the product is installed or which user profile is active.

8 Attribute catalog

Attributes set on `SCType` / `SCProperty` fall into two groups. A small set of core attributes is understood by the descriptor loader itself; everything else is a consumer attribute, carried along on the descriptor and interpreted by the UI, the engine, or a specific presenter. From an author's point of view both are "just attributes" — the table below tells you what each one does.

Boolean attributes take the literal text `True` / `False` (case-insensitive).

8.1 Identity and type

Attribute	Value	Meaning
`ID`	name	The member/type name. Required on declarations.
`type` / `Type`	type name	Value type — a built-in or a previously declared type.
`Caption`	string / expression	Human-readable label shown in the UI. May be computed.
`Version`	integer	Schema version of the type; bump it when you change

Attribute	Value	Meaning
		structure so older saved data is migrated.
`Obsolete`	bool	Marks a member as deprecated; kept for compatibility, generally hidden.

8.2 Values and units

Attribute	Value	Meaning
`DefaultValue`	literal / expression	Default value (metric). For enums, the default option's `ID`.
`InchDefaultValue`	literal / expression	Default value when the imperial unit system is active.
`DimensionKind`	see below	Physical dimension of a numeric value, so the UI shows units and converts correctly.
`UnitsChar`	string / expression	Explicit unit label to display (e.g. `ml/s`, or `[TimeUnits_Sec$]`).

`DimensionKind` values: `None`, `Linear`, `InverseLinear`, `Angular`, `Feedrate`, `CuttingSpeed`, `Revolution`. Use `None` for dimensionless numbers and direction components; `Linear` for distances; `Angular` for angles.

8.3 State and behavior

Attribute	Value	Meaning
`Enabled`	bool / expression	Whether the property is active/editable. `False` greys it out.
`Visible`	bool / expression	Whether the row is shown. Drive it from other properties for dynamic forms.
`ReadOnly`	bool	Value is displayed but not editable.

Attribute	Value	Meaning
`EnabledValue`	member name	For a boolean "switch" member: names the sibling that it enables when true (used with `Compact`).
`IsStructural`	bool	Changing this value changes the object's structure (triggers a rebuild rather than a simple value update).
`IsDynamic`	bool	The member's caption/icon is computed per-instance (e.g. array items that name themselves).

8.4 Presentation and grouping

Attribute	Value	Meaning
`Category`	category id	Assigns the property to an inspector category/tab (e.g. `TPropertiesCategoryList.Feeds`).
`Parent`	member name	Re-parents this row under another member in the inspector tree, regardless of declaration order.
`Priority` / `Order`	integer	Sort order of a row within its inspector group/category (higher `Priority` floats up). For placement in the *new operation* menu, use `MultiGroup` + `OrderInGroup` instead (see Operation descriptors).
`Expanded`	bool	Whether a complex row starts expanded in the tree.

Attribute	Value	Meaning
`Compact`	bool	Collapse a complex type onto a single inspector row (see §9).
`Transparent`	bool	The container itself is not shown; only its children appear (a pure grouping wrapper).
`Text`	string / expression	Summary text shown for a collapsed complex row.
`ImageFile`	path	Icon for the row (use a `$(…)` path variable).

8.5 Editors / presenters

These attributes attach a specialized editor or renderer to a property. The value is the registered name of a presenter provided by the application; you reference it, you don't define it.

Attribute	Meaning
`EnumPresenter`	Custom drop-down/selector for the row (tool selector, magazine selector, ⋯).
`BtnPresenter`	Renders the row as an action button (e.g. "Click to calculate").
`VisPresenter` / `CaptionPresenter`	Custom value/caption renderer.
`IsRadioEdit` / `RadioEditValue`	Combine an enum with sibling value members into one row (see §9).
`Filter` / `CustomDialogID`	For file-name properties: file dialog filter / custom dialog.

8.6 Arrays and integration

Attribute	Meaning
`CollectionKeyField`	Member that uniquely identifies an array element.
`CollectionValueField`	Member that holds the element's payload value.

Attribute	Meaning
`Optional`	On `<SCInclude>`: load only if the file exists.
`ClassName` / `ClassGUID` / `InterfaceName` / `InterfaceGUID`	Bind the property to a COM object/interface the application instantiates for it. Advanced; rarely needed in hand-authored descriptors.

> Unknown attributes are preserved, not rejected. If you put an attribute the loader doesn't recognize, it is kept on the descriptor and is available to consumers and the API. That is by design — it is how presenters receive their parameters (e.g. `AxisIdx`, `HolderType`). It also means a typo in a *known* attribute name fails silently, so double-check spelling and casing.

9 Compact and radio-edit rows

Two presentation patterns appear constantly in real descriptors and are worth understanding because they change how several members map to one inspector row.

Compact

`Compact="True"` on a complex type collapses it onto one inspector row: the first visible child is promoted next to the parent's label. The classic shape is an `Enabled` flag plus the value it gates, with `EnabledValue` linking them:

```
<SCType ID="SmoothCorners" type="ComplexType" Compact="True">
    <SCType ID="Enabled" type="Boolean" DefaultValue="True" EnabledValue="Radius"/>
    <SCType ID="Radius"  type="TPercentageValue" Visible="False">
        <PercentValue DefaultValue="25"/>
    </SCType>
</SCType>
```

(real example: `Operations/MillOperations/FaceMillingOp.xml`)

The user sees a single "Smooth corners" row with a checkbox and a value.

Radio edit

`IsRadioEdit="True"` on an enumeration merges it with sibling value members into one row, where the enum choice selects which sibling is active. Each enum option points at its sibling via `RadioEditValue`:

```
<SCType ID="ReferenceType" type="Enumerated" IsRadioEdit="True">
    <SCType ID="Angle"  Caption="(degrees)"     type="none" RadioEditValue="Angle"/>
```

```
    <SCType ID="Linear" Caption="[LinearUnits$]" type="none"
RadioEditValue="Linear"/>
</SCType>
<SCType ID="Angle"  type="Double" DefaultValue="5"  Visible="False"
DimensionKind="Angular"/>
<SCType ID="Linear" type="Double" DefaultValue="50" Visible="False"
DimensionKind="Linear"/>
```

(real example: `Machines/AbstractMachine.xml`)

Next: Expression language

Expression language

> XML paths in this article refer to configuration files in the installed CAM system (`$(SCHEMAS_FOLDER)`), not manual attachments. Available files depend on the version and configuration.

Many attribute values are not constants — they are expressions evaluated against other properties. This is what makes descriptors dynamic: a default that depends on another field, a row that appears only under certain conditions, a caption that reflects the current unit system.

1 Where expressions are used

Any of these attributes may hold an expression instead of a literal:

- `DefaultValue`, `InchDefaultValue` — computed defaults.
- `Visible`, `Enabled`, `ReadOnly` — dynamic state (the expression must yield a boolean).
- `Caption`, `Text`, `UnitsChar` — computed display strings.
- a handful of others used by specific subsystems (e.g. error-state flags).

An attribute value is treated as an expression when it contains a property reference in `[...]` brackets or a recognized function/operator. A plain value like `DefaultValue="1"` or `DefaultValue="Millimeters"` is taken literally.

2 Property references: `[...]`

A reference in square brackets resolves to the value of another property.

```
Visible="[Enabled]"
DefaultValue="[ApproachFeed.ValuePerMinut]"
```

Name resolution

A bare name like `[Enabled]` is looked up in this order:

1. siblings of the current property,
2. the parent chain,
3. global properties / context objects in scope.

So `[Enabled]` next to the property usually means "the sibling `Enabled`".

Dotted paths

Dots navigate into complex types and across the tree:

```
DefaultValue="[ApproachFeed.LatheFeedRelativeMeasurement]"
Visible="[TechOperation.IsPrevOpInOtherChannel]"
```

A path can also address an enumeration option by name — `[Group.LinearAxis]` is the `LinearAxis` option of the `Group` enum. Comparing a property to one of its options is the idiomatic way to test an enum:

```
Visible="[SimulationType] = [SimulationType.Painting]"
DefaultValue="([Group] != [Group.LinearAxis]) AND (([Max]-[Min]) GT 10000)"
```

Context roots

Besides nearby properties, some context objects are available as path roots, giving expressions access to the wider environment. The ones you'll see most:

Root	Gives access to
`TechOperation`	The operation that owns the property (flags, neighbours, magazine info, …).
`CAMApplication`	Application-level state (e.g. `[CAMApplication.PLMManager.ConnectionCount]`).
`CurrentMeasurementsSystem`	The active unit system and its captions.

3 Accessors inside `[...]`

Within a reference you can call accessors on the resolved property:

Accessor	Returns
`.Attribute(Name)`	The value of the descriptor attribute `Name` (e.g. `Caption`, `ItemIndex`, or any custom attribute).
`.EnumValue`	The currently selected option of an enum (so you can chain `.Attribute(Caption)` to get its label).
`.Value`	The property value explicitly (usually implicit).
`.Count`	Element count of an array.

```
<!-- the caption of the currently chosen LinearUnits option -->
DefaultValue="[CurrentMeasurementsSystem.LinearUnits.EnumValue.Attribute(Caption)]"

<!-- this array element's running index, for a self-numbering caption -->
Caption="Point [Attribute(ItemIndex)]"
```

Inside an array element template, `$ParentNode` refers to the enclosing collection element, e.g. `Caption="Rule [$ParentNode.Attribute(ItemIndex)]"`.

4 Operators

Group	Operators
Arithmetic	`+ - * / ^` (power) `DIV` (integer divide) `MOD`
Comparison	`= != <> < > <= >=` and the word forms `LT GT LE GE`
Logical	`AND OR NOT`

Word forms exist because XML attribute values are quoted and the symbolic `</>` can be awkward to read next to markup — `GT`/`LT`/`GE`/`LE` are the common choice in real files. Parentheses group sub-expressions as usual.

```
Visible="[Enabled] and ([StepCount] GT 1)"
DefaultValue="[UpperWorkLevel] + 5"
SupportShortestPathRotation DefaultValue="([Group] != [Group.LinearAxis]) AND
(([Max]-[Min]) GT 10000)"
```

5 Functions

Conditionals — `CASE`, `IF`, `SWITCH`

These three cover almost all real-world logic.

`IF(condition, valueIfTrue, valueIfFalse)` — the plain ternary.

```
DefaultValue="IF([SimulateToolChange], 'Auto', 'DoNotUse')"
```

`CASE(...)` — the most common conditional in descriptors. In its two-value form it behaves like `IF`: `CASE(condition, valueIfTrue, valueIfFalse)`. It also chains: a series of `condition, result` pairs, evaluated in order, with an optional final default.

```
<!-- if/else form -->
DefaultValue="CASE([IsSpatial], 'SnapToWorkpieceCS', 'SnapToToolCS')"

<!-- chained: first matching condition wins, last bare value is the default -->
DefaultValue="CASE(
    [HolderType]=[HolderType.LeftLatheSpindle],  '1',
    [HolderType]=[HolderType.RightLatheSpindle], '2',
    -1)"
```

(real example: `Machines/MachineTypes.xml`)

`SWITCH(value, case1, result1, case2, result2, …, default)` — selects by equality against `value`, like a classic switch statement. Useful when one property maps to many outcomes:

```
RotationsSequence DefaultValue="
    SWITCH([ControlData.ToolFrameOutput.Format],
        [ControlData.ToolFrameOutput.Format.EulerXYZ], 'XYZ',
        [ControlData.ToolFrameOutput.Format.EulerXZY], 'XZY',
        'XYZ')"          <!-- final value is the default -->
```

(real example: `Machines/AbstractMachine.xml`)

Numeric functions

Function	Meaning
`MIN(a, b, …)` / `MAX(a, b, …)`	Minimum / maximum
`ABS(x)`	Absolute value
`ROUND(x [, digits])` / `SMARTROUND(x)`	Rounding
`SQRT`, `EXP`, `LN`, `SGN`/`SIGN`	Common math
`SIN`, `COS`, `TAN`, `ASIN`, `ACOS`, `ARCTAN`, …	Trigonometry

String functions

Function	Meaning
`STR(x)`	Number → string
`CHR(code)`	Character from code point

Function	Meaning
`CONCATENATE(sep, s1, s2, …)`	Join strings with a separator
`ISVALUEINSET(v, a, b, …)`	True if `v` equals one of the listed values

6 String literals

String literals inside expressions use single quotes: `'XYZ'`, `'Auto'`.

```
DefaultValue="IF([SimulateToolChange], 'Auto', 'DoNotUse')"
```

7 Worked examples

A value that defaults to another property — keep two feeds in sync unless edited:

```
<ValuePerMinut DefaultValue="[ApproachFeed.ValuePerMinut]"
               InchDefaultValue="[ApproachFeed.ValuePerMinut]"/>
```

A row that appears only when a flag is set and a count is meaningful:

```
<SCType ID="StepValue" type="Double" DefaultValue="1" DimensionKind="Linear"
        Visible="[Enabled] and ([StepCount] GT 1)"/>
```

An enum default chosen from another enum's state:

```
<SCType ID="DefaultState" type="Enumerated"
        DefaultValue="if([RotaryTrans.TCPMAvailable] AND (NOT
[RotaryTrans.TCPMDefault]), 'Auto', 'Off')">
    <SCType ID="Off"  type="none"/>
    <SCType ID="Auto" type="none"/>
</SCType>
```

(real example: `Machines/AbstractMachine.xml`)

> Tips. Keep expressions side-effect free — they are evaluated whenever the UI refreshes. Test enum comparisons with the `[Enum.Option]` form rather than string literals, so a renamed option fails loudly instead of silently mismatching. When a `Visible`/`Enabled` expression references a property in another branch of the tree, use a full dotted path from a known root.

Next: Operation descriptors

Using XML properties from code

The properties you declare in XML are the same properties you read and write from code. They are reachable from several environments — the CAM API (.NET, C++, ⋯), native Delphi / C++ packages, .NET postprocessors, and `.sppx` postprocessors — and, crucially, the access pattern is identical in all of them. This chapter describes that common pattern once, then shows the small per-environment specifics.

1 One model, one access pattern, many hosts

Recall the types-vs-instances distinction. From any host you work with instances: the live property tree of a concrete object, where each node is a property pointer. From a pointer you read or write its value, step into children, search, and inspect its descriptor for type metadata.

A pointer always offers the same named accessors, whatever the host:

Accessor	What it does
`Str[name]` / `Int[name]` / `Flt[name]` / `Bol[name]`	read and write a child's value as string / integer / double / boolean
`Ptr[name]`	get a child pointer (to step into a `ComplexType`)
`Arr[name]`	get a child array
`ValueAsString` / `ValueAsInteger` / `ValueAsDouble` / `ValueAsBoolean`	the pointer's *own* value
`FindChild` / `FindProperty`	locate a child by name

Names can be compound, and the case does not matter. Rather than stepping with `Ptr`, pass the dotted path straight to the accessor, and address array elements by key or index with `(…)`:

```
Flt["RoughPasses.StepValue"]          # nested value, dotted
Int["controldata.usearc"]             # case-insensitive
Flt["Axes(AxisXPos).Value"]           # array element by key, then a child
Str["Project.Operations(0).Name"]     # array element by index
```

The same accessor names, the same dotted/case-insensitive lookups and the same `(key)` array syntax appear in every host; only the language wrapping differs:

Host	Pointer type	COM wrapping?
CAM API (.NET / C++ / ⋯)	`IST_XMLPropPointer`	yes
Native Delphi / C++ packages	`IST_XMLPropPointer`	no
.NET postprocessors	`INamedProperty`	no
`.sppx` postprocessors (Pascal-like language)	built-in	no

So §2–9 below are the .NET CAM API flavour: the `ComWrapper`, `Invoke` / `InvokeAndWrap` and `using` you will see there are .NET-COM wrapping, *not* part of the property model. The native and postprocessor hosts (§10–12) use the very same accessor names without that ceremony.

Finding a property's full name

To use the accessors you need a property's full, dotted name. The quickest way to get it is from the running program:

1. Open the operation's property inspector and find the property.
2. At the right edge of its row click the ⋯ (three-dots) button to open the row menu, and choose "Copy property name to clipboard."

The same menu has "Customize⋯", which opens a window showing not only the property's computed value but its source expression — handy when the value is parametrized from other properties (see the expression language). *(This window exists only in recent versions.)*

> Older versions — a workaround. If there is no *Customize⋯* window, use the operation's context menu → "Save as user operation." In the window that opens, find the property in the inspector, right-click it, and use "Copy property name to clipboard" there. Then strip the leading **`RefOperations(0).`** from the copied name — it appears only because that window shows a *copy* of the properties inside a separate *UserOperation* entity, not the live operation. What remains is the name you pass to the accessors.

2 Getting the property tree of an object

Domain objects expose their property tree through an `XMLProp` member. For a technological operation:

```
using var xmlPropsCom = operationCom.InvokeAndWrap(operation => operation.XMLProp);
```

> **`XMLProp`** is the object's unified property tree. For properties kept only in `XMLProp` (the modern case) it is authoritative — reading/writing it needs no commit step. But some operations still mirror certain values in their own object fields (legacy code); there `XMLProp` and the object can diverge and must be synchronized with `SaveToXMLProp` / `LoadFromXMLProp` — see §7.

The same `XMLProp` pattern applies to machines, tools and other objects.

3 Reading and writing values by name

Each node exposes typed indexers keyed by child name: `Str`, `Flt`, `Int`, `Bol`. This is the idiom used throughout the examples — set a value by assigning to the indexer:

```
// from FullWorkflow3DProject/TechnologyHelper.cs
using var xmlPropsCom = operationCom.InvokeAndWrap(operation => operation.XMLProp);
xmlPropsCom.Invoke(xmlProps =>
{
    xmlProps.Str["DrillingType"] = "HolePocketing";   // write a string / enum
option id
});
```

Reading is symmetric:

```
xmlPropsCom.Invoke(xmlProps =>
{
    double step  = xmlProps.Flt["StepValue"];
    int    count = xmlProps.Int["StepCount"];
    bool   on    = xmlProps.Bol["Enabled"];
    string mode  = xmlProps.Str["DrillingType"];
});
```

As in every host (§1), the name can be a dotted path and is case-insensitive, so you can reach a nested value in one call instead of stepping with `Ptr` (§4):

```
double step = xmlProps.Flt["RoughPasses.StepValue"];
bool   on   = xmlProps.Bol["roughpasses.enabled"];   // case ignored
```

Check existence first

A property may not exist on a given object (e.g. an optional or custom parameter). Guard reads with `PropExists`:

```
var value = xmlProps.PropExists["DrillingType"]
    ? xmlProps.Str["DrillingType"]
    : "";
```

> Enums — by ID or by index. An enumeration's value is its *option `ID`*, so `Str["DrillingType"] = "ChipRemoving"` sets it by ID (the option's `ID`, not its caption — see §3.2). You can also read or write it by zero-based option index with `Int`: `Int["DrillingType"] = 2` selects the *third* option, and reading `Int["DrillingType"]` returns the current option's index. (This works in every host.)

The convenience helper layer (.NET)

The .NET helper package wraps those indexers as extension methods on `ComWrapper<IST_XMLPropPointer>`, so you can avoid the explicit `Invoke`:

```
double step = xmlPropsCom.Flt("StepValue");      // getter
xmlPropsCom.SetFlt("StepValue", 2.5);            // setter
xmlPropsCom.SetStr("DrillingType", "ChipRemoving");
string calc = xmlPropsCom.CStr("SomeComputed");  // evaluated value of a
computed property
```

Helpers exist for each type (`Str`/`Flt`/`Int`/`Bol` and their `Set…` forms), plus `Ptr`, `Arr`, `FindProperty`, `FindPropertyInWholeScope`, and more. Use whichever style fits your code; they call the same underlying interface.

4 Navigating the tree

Step into complex types

`Ptr[name]` (or the `Ptr(name)` helper) descends into a `ComplexType` child; continue with the typed indexers. This mirrors the dotted paths from your XML (`RoughPasses.StepValue`):

```
xmlPropsCom.Invoke(xmlProps =>
{
    var rough = xmlProps.Ptr["RoughPasses"];
    rough.Bol["Enabled"]   = true;
    rough.Flt["StepValue"] = 1.0;
});
```

Search by name

When you don't want to spell out the full path, search the subtree:

```
using var dt = xmlPropsCom.FindPropertyInWholeScope("DrillingType");
string current = dt.ValueAsString();
dt.SetNodeValue("ChipRemoving");
```

`FindPropertyInWholeScope` returns the first match at any depth — prefer `Ptr` chains (or the scoped `FindProperty`) when a name might not be unique.

5 Arrays

An array child is an `IST_XMLPropArray`, obtained with `Arr[name]`. It supports element creation and addition on top of the normal pointer operations. This is the real pattern used to add a custom array element:

```
// from ExtensionOperationParamsNet/OperationCustomPropsHelper.cs
const string arrayId = "CustomOperationPropertiesArray";
using var paramArrayCom = ComWrapper.Create(operation.XMLProp.Arr[arrayId]);

if (!operation.XMLProp.PropExists[fullId])
{
    var newItem = paramArrayCom.Invoke(a =>
a.CreateNewItem("CustomOperationProperty"));
    newItem.Str["Name"] = paramId;
    paramArrayCom.Invoke(a => a.AddItem(newItem));
}
```

Addressing an array element by key

When an array element has a `Name`/key member, you can address it directly with the `ArrayName(key)` path syntax in the indexer — no manual iteration:

```
var fullId = $"{arrayId}({paramId})";          // e.g.
CustomOperationPropertiesArray(MyParam)
operation.XMLProp.Str[fullId] = "true";        // read/write the element's value
bool exists = operation.XMLProp.PropExists[fullId];
```

6 A pointer's own value

The accessors in §3 read a child by name (`Str["StepValue"]`). When you already hold the pointer to the property *itself* (rather than its parent), read or write its own value. This is typed too — `ValueAsString`, `ValueAsDouble`, `ValueAsInteger`, `ValueAsBoolean` (see the accessor list in §1). `ValueAsString` applies to any property type and returns its string representation:

```
double v   = somePropCom.ValueAsDouble();   // typed own value
string raw = somePropCom.ValueAsString();   // any type → its text form
somePropCom.SetNodeValue("75");             // set the pointer's own value
```

7 Operations: syncing object fields with `XMLProp`

(Operations only.) For an operation, `XMLProp` is its unified property tree with two jobs: a single runtime tree (so common code can show every operation in the inspector, and values can be copied between operations) and the form in which the operation's properties are serialized to/from XML when the project is saved or loaded.

Historically, operations held their values in plain object fields, and `XMLProp` was added later on top. So legacy properties live in *both* places — the object's fields and `XMLProp` — and the two copies must be kept in step:

- **`SaveToXMLProp`** — push the operation's current field values into **`XMLProp`**. Call it before reading `XMLProp` (e.g. before the inspector is populated from the tree) so the tree reflects the live object state.
- **`LoadFromXMLProp`** — read values from **`XMLProp`** back into the object's fields. Call it after `XMLProp` was edited (e.g. after the user changed a value in the inspector) so the object model picks the change up.

Modern properties live only in `XMLProp` — no duplicate field, so no synchronization is needed; reading/writing `XMLProp` is enough. The sync calls matter mainly for the older classes, of which there are still many.

`LoadFromXMLProp` also accepts a *different* operation's `XMLProp`, which is how an operation's settings are copied from another:

```
operationCom.LoadFromXmlProp(otherXmlPropsCom);   // apply another property tree to this operation
```

`SaveToXmlProp()` returns the operation's properties as a tree (with the object fields flushed into it) — handy to snapshot, diff or transfer.

8 Inspecting descriptors and defaults

Every property pointer links to its descriptor (`IST_XMLPropDescriptor`), which carries the metadata you declared in XML — name, caption, simple type, default value, custom attributes, inheritance. Use it when you need to reason about the *schema* rather than the value (for example, enumerating an object's parameters generically, or reading a custom attribute you set on the descriptor).

Defaults are reachable through `IST_PropDefaultsEditor` (queried from the descriptor): it distinguishes the system default (declared in the descriptor XML) from any user default override, for both metric and inch unit systems, and lets you read, set or clear the user default.

9 Practical notes

- Dispose COM wrappers. Property pointers and arrays are COM objects; wrap them in `using` so they are released promptly. In the examples, work is done inside `Invoke` / `InvokeAndWrap` lambdas which manage lifetime for you.
- Use the exact **`ID`** from the descriptor — names must match what the XML declares.
- Set enums and booleans with their canonical text — option `ID` for enums; for booleans the examples use the strings `"true"`/`"false"` via `Str[...]`, or the typed `Bol[...]` indexer.
- Don't fight computed values. A property whose value is driven by a `DefaultValue`/`Visible` expression will recompute; read its evaluated result with `CStr` (or `ValueAsCalculatedString`), and make it a real editable field if you need to store a fixed value.
- Where to look next. The `cam-api-examples` repository (`Operation/ExtensionOperationParamsNet`, `FullWorkflow`, `ProjectMachine`) contain complete, compilable usages of everything above.

10 Native use (Delphi / C++)

The native packages expose the same property pointer through the public `IST_XMLPropPointer`. There is no COM wrapper, no **`Invoke`**, no **`using`**: you hold an ordinary interface reference and call the same accessors directly. A domain object's `LoadFromXMLProp` / `SaveToXMLProp` (and `LoadConditionsFromXMLProp`) methods receive such a pointer:

```
procedure LoadFromXMLProp(XMLProp: IST_XMLPropPointer);
var p: IST_XMLPropPointer;
begin
  if XMLProp <> nil then with XMLProp do begin
    p := FindProp('SafeLevel');
    if p <> nil then with p do begin
      fZRetract.IsAbs  := Int['ReferenceType'] = 0;   // the same
Int[…]/Flt[…] accessors
      fZRetract.IncVal := Flt['RelValue'];
      fZRetract.AbsVal := Flt['AbsValue'];
    end;
  end;
end;
```

It is the same `Int[…]` / `Flt[…]` / `Str[…]` / `Bol[…]` / `FindProp` as from .NET — just without the COM ceremony.

11 In .NET postprocessors (`INamedProperty`)

A .NET postprocessor reads the named properties of the command/project it is given and emits NC code. It uses a pure-.NET layer with no COM — **`INamedProperty`** — but the access is the same: `Str` / `Int` / `Flt` / `Bol` / `Ptr` / `Arr` indexers plus `ValueAsString` / `ValueAsInteger` / `ValueAsDouble` / `ValueAsBoolean`, with the familiar dotted, case-insensitive names and `(key)` / `(index)` array addressing:

```
double x      = cmd.Flt["Axes(AxisXPos).Value"];
int    origin = cmd.Int["OriginType"];
double ox     = cmd.Flt["WCS.OriginPoint.X"];
```

A worked postprocessor is in the `postprocessors-examples` repository under `Features/NamedParameters/DemoOfWritePPFun` (see `WritePPFun.cs`). That example is a slightly unusual context — it *writes* the parameters to a file — but the access to the properties is exactly as above.

12 In `.sppx` postprocessors (Pascal-like language)

`.sppx` postprocessors are written in a dedicated, Pascal-like language. The set of available types differs, but the method names and the way you access properties are identical. In the snippets below `Cmd` is one such property-bearing object — the current CLData command — and the same accessors apply to any property object: `.Int["…"]` / `.Flt["…"]` / `.Str["…"]` / `.Bol["…"]` for values and `.Ptr["…"]` for a child property pointer, all with the familiar dotted / `(key)` names:

```
if Cmd.Ptr["Axes(AxisXPos)"] <> 0 then          // child property pointer (here:
does the axis exist?)
  X = Cmd.Flt["Axes(AxisXPos).Value"]
if Cmd.Int["Axes(AxisAPos).BrakeState"] > 0 then ...
CurG68_2_X = Round(Cmd.Flt["WCS.OriginPoint.X"], 4)
```

(real example: `MyDocuments\Postprocessors\Mill\Fanuc (30i)_Mill.sppx`)

Next: User defaults · Back to the guide index.

User defaults — overriding shipped defaults

Every property descriptor ships with a default value — the value a new object starts with, declared as `DefaultValue` (and `InchDefaultValue`) in the XML (§8). Those shipped values live under `Supplement`, which you do not edit. Instead, the framework lets you set your own default for any property — a *user default* — that takes precedence over the shipped one. User defaults are stored in small `.usrdef` files, loaded at startup, and applied without recompiling or editing `Supplement`.

This is the supported way to answer "I always want this parameter to start at *my* value, not the factory value" — for a single seat, or rolled out across many.

System default vs. user default

Two layers sit behind every property's default:

- the system default — the value shipped in the descriptor's `DefaultValue` / `InchDefaultValue`. It is read-only and never changed by this mechanism; it is always there to fall back to.
- the user default — your override. When set, it becomes the property's *effective* default: new objects start from it, and "reset to default" returns to it.

Because the two are kept separately, you can always reset to the system default to get the factory value back, or remove your user default entirely.

Setting a user default from the inspector

You manage user defaults straight from the parameter inspector. Open a property row's context menu (the same per-row menu used to copy a property's full name — see Using XML properties from code); it offers:

Command	Effect
Save as user default	Make the property's current value your user default.
Reset to user default	Set the property back to your user default.
Reset to system default	Set the property back to the shipped (factory) value.
Remove user default	Forget your override; the system default applies again.
Customize…	Open the customization dialog for broader editing.

Changes take effect immediately — there is no need to restart the application.

> Metric vs. inch. Just as descriptors carry both `DefaultValue` and `InchDefaultValue`, a user default is stored per measurement system: setting it while working in inches records an inch value that is independent of the metric one. Each is used when the project runs in the matching unit system.

> Grouped inspector rows. When several properties share one inspector row — a `Compact` complex type, or a unit-switching radio-edit row (§9) — saving or resetting the row acts on all the properties visible in it together, not just the one you clicked.

Where user defaults are stored

User defaults are kept in `.usrdef` files. They are pulled in at startup by the master configuration with optional, wildcarded includes — for example, for the `Operations` family:

```
<SCInclude Optional="True">$(SUPPLEMENT_FOLDER)\Defaults\*.usrdef</SCInclude>
<SCInclude Optional="True">$(LOCAL_OPERATIONS_FOLDER)\*.usrdef</SCInclude>
```

They are included after the system descriptors, which is why their values win over the shipped defaults. The *Save as user default* command writes these files to your user area automatically (the `$(LOCAL_OPERATIONS_FOLDER)` location, alongside your custom operations — see §7), so in normal use you never edit them by hand. Other descriptor families are loaded the same way by their own configuration.

The `.usrdef` file format

For inspection, version control, or scripted roll-out, a `.usrdef` file is plain XMLProperties. It mirrors the path from a root type down to the overridden property, and the leaf carries the override as a `UserDefaultValue` (or `UserInchDefaultValue`) attribute:

```
<?xml version="1.0" encoding="UTF-8" ?>
<SCCollection>
    <SCNameSpace ID="Operations">
        <TST2DContouringOp>
            <HelicalMachining>
                <Enabled UserDefaultValue="True"/>
            </HelicalMachining>
        </TST2DContouringOp>
    </SCNameSpace>
</SCCollection>
```

- One **`<SCNameSpace ID="…">`** per namespace (e.g. `Operations`, `Tools`, ⋯); see namespaces.
- The element nesting follows the property path from the root type (`TST2DContouringOp`) through any sub-objects (`HelicalMachining`) to the property (`Enabled`).
- The leaf element carries **`UserDefaultValue`** for the metric default and/or **`UserInchDefaultValue`** for the inch default — never the system `DefaultValue`, which stays untouched.
- Conventionally there is one file per root type, named `<RootType>.usrdef` (e.g. `TST2DContouringOp.usrdef`).
- If a `.usrdef` refers to a type or property that the running system does not declare, that entry is simply ignored — a stale override does no harm.

Deploying defaults across workstations

Because user defaults are just files, a standard configuration can be moved between seats. Your custom operations together with their user defaults can be exported to, and imported from, a single archive — letting an integrator prepare a house standard on one machine and deploy it to others. The archive carries the operations configuration, the custom operation files, optional icons, and the `Defaults\*.usrdef` files.

Turning user defaults off

If you need the factory behaviour back wholesale, there is an escape hatch that disables all user defaults at once: it moves every `*.usrdef` aside into a backup subfolder so none are loaded. Since descriptors are read at startup, restart the application for the change to take full effect.

Back to the XMLProperties framework index.

Operation XML descriptors

This area covers the XML descriptors of technological operations — how to describe and register an operation, and how the shipped operations are organized. It is not a full manual of what each operation *does*; it is about authoring and understanding their XML descriptors. It builds on the XMLProperties framework (read the descriptor syntax and expression language first).

- Operation XML descriptors — the anatomy of an operation descriptor: the registration record, choosing a base type, the header, parameters, `ContainerID` / `SolverID`, `MultiGroup`, and where your file goes.
- Operation hierarchy reference — the full inheritance tree of every shipped operation (ID, caption, parent, container/solver, source file), to help you pick a base type. *Generated — do not hand-edit.*

See the XMLProperties framework for the shared ground rules (don't edit `Supplement`, where the shipped files live, example conventions), and the documentation root for the intended audience.

Operation descriptors

> XML paths in this article refer to configuration files in the installed CAM system (`$(SCHEMAS_FOLDER)`), not manual attachments. Available files depend on the version and configuration.

A technological operation is described by one big `ComplexType` derived from an existing operation, plus a small registration record that announces it to the system. This chapter shows the anatomy of an operation using two real files: `Operations/MillOperations/FaceMillingOp.xml` (a shipped operation) and the partner example `OperationSimpleNet.xml` from the `cam-api-examples` repository (an operation whose toolpath is computed by a C# extension).

The shipped operation files live under `Operations\` and `Operations\MillOperations\` and are pulled in by `Operations.xml` — read them as your reference. Your own operation does not go there. Register it instead by one of the two mechanisms in §7: drop an `*_ExtOp.xml` file in the common containers folder (picked up automatically), or point the Operations Manager wizard (Utilities → Operations Manager) at your operation XML file. Either way the file is an `<SCCollection>` in the **`Operations`** namespace and uses exactly the syntax shown below.

1 The two pieces

To add an operation you provide:

1. A registration record in the `OperationRegistrator` namespace, whose `TypeName` equals the `ID` of your operation type.
2. The operation type itself, derived from an existing operation base.

```
<SCCollection>
    <!-- 1. Register the type name -->
    <SCNameSpace ID="OperationRegistrator">
        <SCType ID="RegTSTFaceMillingOp" type="TRegisterOperationRecord"
Enabled="True">
            <TypeName DefaultValue="TSTFaceMillingOp"/>
        </SCType>
    </SCNameSpace>

    <!-- 2. The operation type (its ID must match TypeName above) -->
    <SCType ID="TSTFaceMillingOp" Caption="FaceMilling" type="TSTMillOp"
Enabled="True">
        ...
    </SCType>
</SCCollection>
```

(real example: `Operations/MillOperations/FaceMillingOp.xml`)

Why the registration record exists

There can be a very large number of descriptors in the system. The `OperationRegistrator` namespace is a small, fast lookup index: it lists which XML types are actually technological operations, so the system can find them directly instead of scanning the entire body of descriptors. A record is just a `TRegisterOperationRecord` whose `TypeName` points at your operation type's `ID`. If you forget the record, your type exists but is not recognized as an operation.

2 Choosing a base type

Operations inherit a great deal of structure (tool section, feeds, leads, simulation, machine state, …) from a chain of base operations. Pick the base whose behavior is closest to yours and override/extend from there:

Base type	For
`TSTMillOp` (milling family)	Milling operations — e.g. `TSTFaceMillingOp` derives from it
Lathe operation base	Lathe operations
Axial / drilling base	Drilling and axial cycles
`TSTMillExtensionOp`	Operations whose toolpath is computed by a CAM API extension (see §3)

The simplest way to find the right base is to open the shipped operation most like the one you want and reuse its base type. The full inheritance tree of every shipped operation — IDs, captions, parents, implementing classes and source files — is tabulated in the operation hierarchy reference.

3 The operation header

Near the top of the operation type you override a set of inherited "header" members that identify and present the operation:

```
<SCType ID="TSTFaceMillingOp" Caption="FaceMilling" type="TSTMillOp" Enabled="True">
    <GUID        DefaultValue="{9F01E1A0-6F3E-4C7A-9C52-9A024CED54FF}"/>
    <ContainerID DefaultValue="{F4DE00D1-7AE3-468E-BA5C-ED3C8275CBF8}"/>

    <Name    DefaultValue="FaceMilling"/>
    <Comment DefaultValue="Face Milling"/>
    <OperationGroup DefaultValue="Mill"/>
```

```
    <Image DefaultValue="Images\FaceMilling.png"/>
    <Icon  DefaultValue="Images\FaceMilling_Ico.bmp"/>
    <Video DefaultValue="Video\FaceMilling.wmv"/>
    ...
</SCType>
```

(real example: `Operations/MillOperations/FaceMillingOp.xml`)

Header member	Meaning
`GUID`	Unique, permanent identity of the operation type. Generate a fresh one; never reuse.
`ContainerID`	Class GUID of the Delphi container that represents the operation in the operation tree (see below). Usually inherited; override only when your operation needs a specific container class.
`SolverID`	Identifies the solver that computes the toolpath (see below). Usually inherited.
`Name` / `Comment`	Default display name and description.
`OperationGroup`	Broad technological group the operation belongs to (`Mill`, `Lathe`, ⋯).
`Image` / `Icon` / `Video`	Illustration, list icon, and tutorial clip.

Useful attributes on the type element itself:

- `Enabled="True"` — the operation is available. Set `False` to ship it disabled.
- `DefaultVisibility="False"` — keep a specialized operation loadable but hidden from the default menu.
- `Version` — schema version; bump it when you change the operation's structure so existing saved data migrates correctly.

ContainerID and SolverID — container, solver and toolpath

These two identifiers describe how an operation is implemented. Understanding the split helps you decide what to inherit and what to override.

- **`ContainerID`** — the container class. It is the class GUID of the Delphi class that *represents the operation* in the CAM system's operation tree. The system instantiates this container class; the

container then builds the operation instance from the XML descriptor, owns it and all its parameters, and is the object the rest of the system talks to.
- **`SolverID`** — the (optional) solver. When set, the container creates a separate solver subclass identified by `SolverID`, and that solver computes the toolpath. When `SolverID` is *not* set, there is no separate solver and the container class computes the toolpath itself.

> Why the split exists (history). Originally there were neither XML descriptors nor separate solvers — an operation was just a container class that defined all of its behavior: its parameters and its toolpath computation. XML descriptors (to declare parameters declaratively) and solvers (to factor out toolpath computation, including into CAM API extensions) were layered on later. That is why an operation can still work with only a container and no solver.

In practice you inherit both `ContainerID` and `SolverID` from your base type. Override `ContainerID` only when your operation needs a specific container class; set `SolverID` to choose the engine that computes the toolpath.

`SolverID` accepts two forms:

1. A GUID of an existing built-in solver class — reuse an engine the system already provides:

   ```
   <SolverID DefaultValue="{301F6C21-2499-4514-83D1-72A4931529BE}"/>
   ```

2. The identifier of a CAM API extension you wrote — specifically an `OperationSolver` extension. Here `SolverID` is the extension's string `id`, not a GUID:

   ```
   <!-- from OperationSimpleNet.xml -->
   <SCType ID="TSimpleNetOP" Caption="Simple .NET" type="TSTMillExtensionOp"
   Enabled="True">
       <GUID     DefaultValue="{BDD2CAD7-D2A9-43C4-86E3-399FCB439FC0}"/>
       <SolverID DefaultValue="Extension.Operation.Simple.Net"/>
       ...
   </SCType>
   ```

   The same `id` is declared in the extension's settings file, under the `OperationSolver` entry:

   ```
   // ExtensionOperationSimpleNet.settings.json
   "extensions": [
       { "OperationSolver": { "id": "Extension.Operation.Simple.Net" } }
   ]
   ```

Working, compilable `OperationSolver` extensions live in the `cam-api-examples` repository under `Operation/` (`ExtensionOperationSimpleNet`, `ExtensionOperationParamsNet`). Derive such an operation from `TSTMillExtensionOp` (or the appropriate extension base) so the standard machinery is wired up for you.

Grouping in the "new operation" menu — `MultiGroup`

`MultiGroup` controls where the operation appears in the *create new operation* tree. It lets one operation show up in several groups at once, and lets different interface configurations (e.g. *Beginner* vs *Expert*) place it in different groups. Each entry is a `TLinkToParentMultiGroup` whose `ID` is the target group; `OrderInGroup` sets its position within that group.

```
<MultiGroup>
    <SCType ID="Group3DEntry" OrderInGroup="1" type="TLinkToParentMultiGroup"/>
    <SCType ID="Roughing"     OrderInGroup="1" type="TLinkToParentMultiGroup"/>
</MultiGroup>
```

(real example: `Operations/MillOperations/FaceMillingOp.xml`)

Use `MultiGroup` + `OrderInGroup` for all menu placement and ordering.

4 Adding parameters

Parameters are just members of the operation type. Group them under inspector placeholders with `Parent`, order them, attach an icon with `ImageFile`, and make them dynamic with the expression language from [the expression language].

```
<!-- an enum that drives the operation's strategy -->
<SCType ID="Strategy" Caption="Strategy" type="Enumerated" DefaultValue="Spiral"

ImageFile="$(SUPPLEMENT_FOLDER)\operations\TypeImages\FaceMillingStrategy.bmp">
    <SCType ID="Spiral"      Caption="Spiral"           type="None"/>
    <SCType ID="OptimZigzag" Caption="Optimized zigzag" type="None"/>
    <SCType ID="Zigzag"      Caption="Zigzag"           type="None"/>
    <SCType ID="OneWay"      Caption="One way"          type="None"/>
    <SCType ID="OnePass"     Caption="One pass"         type="None"/>
</SCType>

<!-- a value placed under Strategy, shown only for some strategies -->
<SCType ID="Step" Caption="Step" type="TPercentageValue"
        Parent="Strategy" Visible="[Strategy] != 4">
    <ValueType    DefaultValue="Percent"/>
```

```
    <PercentValue DefaultValue="75"/>
</SCType>
```

(real example: `Operations/MillOperations/FaceMillingOp.xml`)

Categories and placeholders

Parameters are assigned to inspector categories and to placeholder containers inherited from the base, via `Category` and `Parent`:

```
<SCType ID="LeadInDistance" Caption="Lead in" type="TPercentageValue"
        Category="TPropertiesCategoryList.Leads" Parent="LeadsPlaceHolder"

ImageFile="$(SUPPLEMENT_FOLDER)\operations\TypeImages\FaceMillingLeadInDistance.bmp"
>
    <PercentValue DefaultValue="10"/>
</SCType>
```

`Parent="LeadsPlaceHolder"` files this row under a grouping node defined by the base operation, so related parameters collect together regardless of where they are declared.

Overriding inherited parameters

Because your operation inherits a full parameter set, much of an operation file is *overrides* of inherited members — tuning defaults, hiding what doesn't apply, or re-enabling options. Use the override form (`<MemberName .../>`) from §4.2:

```
<Stock Visible="False"/>                <!-- hide an inherited parameter -->
<ConditionsSection>
    <Feeds>
        <EngageFeed  Enabled="False"/>   <!-- disable feeds that don't apply -->
        <RetractFeed Enabled="False"/>
    </Feeds>
</ConditionsSection>
```

5 Checklist for a new operation

1. Generate a new `GUID`.
2. Choose `SolverID`: a built-in solver GUID, or the `id` of your `OperationSolver` CAM API extension (derive from `TSTMillExtensionOp` in that case).
3. Add a registration record in `OperationRegistrator` with `TypeName` = your type's `ID`.
4. Derive the type from the closest existing operation base.

5. Override the header members (`Name`, `Comment`, `OperationGroup`, `Image`, `Icon`, `Video`) and set `MultiGroup` for menu placement.
6. Declare your strategy parameters and override inherited ones as needed; assign `Category`/`Parent` so they land in the right inspector group.
7. Register the operation — save it as `<YourName>_ExtOp.xml` in `$(COMMON_CONTAINERS_FOLDER)`, or add it through the Operations Manager wizard (see §7). Do not edit the shipped files under `Supplement`.
8. Bump `Version` whenever you later change the structure.

Next: Machine descriptors

Operation hierarchy reference

> XML paths in this article refer to configuration files in the installed CAM system (`$(SCHEMAS_FOLDER)`), not manual attachments. Available files depend on the version and configuration.

> ⚙️ Auto-generated — do not edit by hand. This page is produced by `docs/xml-properties/gen-operation-hierarchy.py` from a registry snapshot joined with the Supplement XML. Re-run that script to refresh it; manual edits are overwritten.

This appendix to Operation XML descriptors maps the full inheritance tree of the technological-operation descriptors that ship with the CAM system. Use it to choose a base type to inherit from: find the shipped operation closest to what you want, then derive your operation from it (or from one of its bases).

The data combines the operation registry reported by the running system with the descriptor declarations in the XML files, so IDs, captions, parents, container classes, container IDs, solvers and source files are the real, current values.

How to read this table

- Rows are ordered depth-first from the most abstract root (`TOperationDescriptor`) down to concrete operations. The Lvl column gives the inheritance depth and Parent names the type each row derives from; together with the ordering they reconstruct the full tree.
- ID — the descriptor ID. This is the `type=` value you put on your own operation to inherit from it, and (for concrete operations) the `TypeName` used in the registration record. Always shown, including for abstract base types.
- Caption — the human-readable name shown in the UI.
- Parent — the descriptor this one derives from.
- Container class / ContainerID — the Delphi container that represents the operation in the operation tree, builds the instance from the XML descriptor, owns it, and either computes the toolpath itself or delegates to a solver. The cell shows the class name on the first line and its class GUID (`ContainerID`) on the second. See *Container, solver and toolpath* in section 4.3. Every operation has a container. Abstract base types (rows that are not registered, instantiable operations) show the container they declare or inherit; the most abstract descriptors fall back to the base container `TSTAbstractTechOp`. Abstract bases are often the best inheritance targets for a new operation.
- Solver class / SolverID — the toolpath solver, shown the same way as the container: the implementing Delphi class on the first line and its `SolverID` on the second. Blank means the operation uses the shared default solver framework (and, when it has no separate solver, its container class computes the toolpath directly). Only operations with a dedicated solver show a value here.

- Declared in — the XML file (relative to the `Supplement` folder) that declares the descriptor. Open it to study the full descriptor.

> Picking a base. For a new milling operation the usual targets are `TST3DMillOp` (3D) or `TSTAbstract5DOp` (3D/5D), or a concrete sibling you want to resemble. For lathe, derive from `AbstractLatheOP` or `LatheContouringOp`. For an operation whose toolpath is computed by your own CAM API extension, derive from an extension base (e.g. `TSTMillExtensionOp`) and set `SolverID` to your extension's id - see section 4.3.

> Maintenance. This page is generated by `docs/xml-properties/gen-operation-hierarchy.py` from a registry snapshot (`data/WholeOpList.json`) joined with the Supplement XML. Re-run that script after refreshing the snapshot from a newer build. A few internal/encrypted operation types are intentionally omitted.

145 registered operations + 50 abstract base types = 195 entries.

Full hierarchy

Lvl	ID	Caption	Parent	Container class / ContainerID	Solver class / SolverID	Declared in
0	`TOperationDescriptor`	Operation descriptor	`TOperationPublicDescription`	`TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}`		`Operations/AbstractOP.xml`
1	`TAbstractTechnologicalOperation`	Abstract Technological Operation	`TOperationDescriptor`	`TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}`		`Operations/AbstractOP.xml`
2	`TAbstractExternalOperation`	Abstract external operation	`TAbstractTechnologicalOperation`	`TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}`		`Operations/AbstractExtOp.xml`
3	`AbstractOP`	Abstract operation	`TAbstractExternalOperation`	`TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}`		`Operations/AbstractExtOp.xml`
4	`AbstractTurnOP`	Abstract Turn operation	`AbstractOP`	`TSCLatheExtOpConnector {E05F6B10-19BF-4755-9185-055853197E99}`		`Operations/AbstractTurnOP.xml`
5	`AbstractAxialOP`	Abstract Axial OP	`AbstractTurnOP`	`TSCLatheExtOpConnector {E05F6B10-19BF-4755-9185-055853197E99}`		`Operations/AbstractAxialOP.xml`
6	`LatheDrillOp`	Lathe drilling (obsolete)	`AbstractAxialOP`	`TSCLatheExtOpConnector {E05F6B10-19BF-4755-9185-055853197E99}`	`TLatheDrillOp {1445D6B7-0006-4F3B-B9AF-06255D7AC38B}`	`Operations/LatheDrilling.xml`
5	`AbstractLatheOP`	Abstract Lathe OP	`AbstractTurnOP`	`TSCLatheExtOpConnector {E05F6B10-19BF-`		`Operations/AbstractLatheOP.xml`

Lvl	ID	Caption	Parent	Container class / ContainerID	Solver class / SolverID	Declared in
				4755-9185-055853197E99}		
6	TNCLatheOp	G-code based lathe	AbstractLatheOP	TNCLatheOp {CE71FFAF-DCFD-434C-9EDB-187F0B47FD58}		Operations/NCLatheOperation.xml
6	LatheContouringOp	Lathe contouring	AbstractLatheOP	TSTLatheContouring {6B3F2DDC-A312-4BB4-8B55-B1250BFAA6DF}		Operations/LatheContouring.xml
7	TLatheFaceMachining	Lathe facing	LatheContouringOp	TSTLatheContouring {6B3F2DDC-A312-4BB4-8B55-B1250BFAA6DF}		Operations/LatheContouring.xml
7	TLathePartOff	Lathe part-off	LatheContouringOp	TSTLatheContouring {6B3F2DDC-A312-4BB4-8B55-B1250BFAA6DF}		Operations/LatheContouring.xml
8	TPartOffWithTakeover	Part-off with takeover	TLathePartOff	TSTPartOffWithTakeover {B9A194EC-12B3-48F0-8958-718D2276D69D}		Operations/PartOffWithTakeover.xml
7	TLatheODAdaptive	OD Adaptive turning	LatheContouringOp	TSTLatheContouring {6B3F2DDC-A312-4BB4-8B55-B1250BFAA6DF}		Operations/LatheContouring.xml
7	TLatheODFinishing	OD Finishing	LatheContouringOp	TSTLatheContouring {6B3F2DDC-A312-4BB4-8B55-B1250BFAA6DF}		Operations/LatheContouring.xml
8	TLatheIDFinishing	ID Finishing	TLatheODFinishing	TSTLatheContouring {6B3F2DDC-A312-4BB4-8B55-B1250BFAA6DF}		Operations/LatheContouring.xml
7	TLatheODGrooving	OD Grooving	LatheContouringOp	TSTLatheContouring {6B3F2DDC-A312-4BB4-8B55-B1250BFAA6DF}		Operations/LatheContouring.xml
8	TLatheFaceGrooving	Face grooving	TLatheODGrooving	TSTLatheContouring {6B3F2DDC-A312-4BB4-8B55-B1250BFAA6DF}		Operations/LatheContouring.xml
8	TLatheIDGrooving	ID Grooving	TLatheODGrooving	TSTLatheContouring {6B3F2DDC-A312-4BB4-8B55-B1250BFAA6DF}		Operations/LatheContouring.xml
7	TLatheODTurning	OD Roughing	LatheContouringOp	TSTLatheContouring {6B3F2DDC-A312-4BB4-8B55-B1250BFAA6DF}		Operations/LatheContouring.xml

Lvl	ID	Caption	Parent	Container class / ContainerID	Solver class / SolverID	Declared in
8	TLatheIDTurning	ID Roughing	TLatheODTurning	TSTLatheContouring {6B3F2DDC-A312-4BB4-8B55-B1250BFAA6DF}		Operations/LatheContouring.xml
7	TLatheODThreading	OD Threading	LatheContouringOp	TSTLatheContouring {6B3F2DDC-A312-4BB4-8B55-B1250BFAA6DF}		Operations/LatheContouring.xml
8	TLatheIDThreading	ID Threading	TLatheODThreading	TSTLatheContouring {6B3F2DDC-A312-4BB4-8B55-B1250BFAA6DF}		Operations/LatheContouring.xml
7	TLatheProfileThreading	Profile threading	LatheContouringOp	TSTLatheContouring {6B3F2DDC-A312-4BB4-8B55-B1250BFAA6DF}		Operations/LatheContouring.xml
6	LatheFacingOp	Lathe facing (obsolete)	AbstractLatheOP	TSCLatheExtOpConnector {E05F6B10-19BF-4755-9185-055853197E99}	TApRetMaker {301F6C21-2499-4514-83D1-72A4931529BE}	Operations/LatheFacing.xml
6	LatheFinishOp	Lathe finishing (obsolete)	AbstractLatheOP	TSCLatheExtOpConnector {E05F6B10-19BF-4755-9185-055853197E99}	TFinishApRet {ED06D20C-844D-439F-8E05-986C6934AE04}	Operations/LatheFinish.xml
6	LatheGroovOp	Lathe grooving (obsolete)	AbstractLatheOP	TSCLatheExtOpConnector {E05F6B10-19BF-4755-9185-055853197E99}	TGrooveOp {0316AE41-3636-4BE9-829B-DF06EE5B67FF}	Operations/LatheGroove.xml
6	LathePartOff	Lathe part-off (obsolete)	AbstractLatheOP	TSCLatheExtOpConnector {E05F6B10-19BF-4755-9185-055853197E99}	TCutOffOp {71606D33-F543-4056-9A38-FF14D053F28C}	Operations/LatheCut.xml
6	LatheRoughOp	Lathe roughing (obsolete)	AbstractLatheOP	TSCLatheExtOpConnector {E05F6B10-19BF-4755-9185-055853197E99}	TControlApRet {384F541C-1867-413D-A73E-BAC1405CE29B}	Operations/LatheRough.xml
6	LatheThreadOp	Lathe threading (obsolete)	AbstractLatheOP	TSCLatheExtOpConnector {E05F6B10-19BF-4755-9185-055853197E99}	TThreadOp {FAD79D4F-5558-4BEA-BE55-A50AC1F406CD}	Operations/LatheThread.xml
6	LatheContouringOP		AbstractLatheOP	TSCLatheExtOpConnector {E05F6B10-19BF-4755-9185-055853197E99}		Operations/LatheContouring.xml
7	TSTLatheContouringCopy	Copy of lathe operation	LatheContouringOP	TSTLatheContouringCopy {F35539E7-F3FE-47D6-8A4F-311A7425A7D7}		Operations/LatheContouring.xml
4	AbstractAuxiliaryOp	Auxiliary operation	AbstractOP	TSCAuxiliaryOp {4B61AE6C-BB81-		Operations/AbstractAuxiliaryOp.xml

Lvl	ID	Caption	Parent	Container class / ContainerID	Solver class / SolverID	Declared in
				4AD1-839A-0BD50696AD26}		
5	TSTAuxOpCopy	Copy of auxiliary operation	AbstractAuxiliaryOp	TSTAuxOpCopy {CDD784B2-1739-4E31-91E7-862D46EB22EB}		Operations/AbstractAuxiliaryOp.xml
2	TSTAbstractMillOp	Abstract mill operation	TAbstractTechnologicalOperation	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/AbstractMillOP.xml
3	TSTMillOpCopy	Copy of milling operation	TSTAbstractMillOp	TSTMillOpCopy {E6A604F7-77E3-4FDD-8127-D0008C30EF36}		Operations/MillOperations/Groups.xml
4	TSTFBMMillOpCopy	Copy of FBM operation	TSTMillOpCopy	TSTFBMMillOpCopy {D388D796-2038-4A5B-9901-24E5598C9886}		Operations/MillOperations/FBMMillOp.xml
4	TSTTechOpGroupCopy	Copy of Group	TSTMillOpCopy	TSTTechOpGroupCopy {FC2325C8-207C-44D8-8739-023DFD6FCBFB}		Operations/MillOperations/Groups.xml
4	TSTFeatureSubOp	Copy of milling operation	TSTMillOpCopy	TSTFeatureSubOp {6B6DA451-4AAB-40BB-BA25-12BDD5608C50}		Operations/MillOperations/Groups.xml
4	TSTHoleMachiningOpCopy	Copy of milling operation	TSTMillOpCopy	TSTHoleMachiningOpCopy {6DE68897-FC1D-4BAA-83D9-115CBAC08213}		Operations/MillOperations/HoleMachiningOperation.xml
4	TSTFeatureOpGroupCopy	Copy of Multiply Group	TSTMillOpCopy	TSTFeatureOpGroupCopy {B58C46CE-0950-433B-9B67-9DC9ACC550BF}		Operations/MillOperations/Groups.xml
3	TNCMillOp	G-code based	TSTAbstractMillOp	TNCMillOp {D29219B7-4E0B-4457-9EAA-AFF841F00EA9}		Operations/NCOperation.xml
3	TSTMillOp	Mill operation	TSTAbstractMillOp	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/AbstractMillOP.xml
4	TST3DMillOp	3D milling operation	TSTMillOp	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/AbstractMillOP.xml
5	TST25DChamferOp	2.5D chamfer machining	TST3DMillOp	TST25DChamferOp {5348F6D3-6614-4D94-9B68-819A2BB2082E}		Operations/MillOperations/Mill25DOp.xml
5	TSTAbstract25DOp	Abstract 25D operation	TST3DMillOp	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/MillOperations/Mill25DOp.xml
6	TST25DWallMachiningOp	2.5D wall machining	TSTAbstract25DOp	TST25DWallOp {1B6B649F-B6D8-4E87-A404-D37F69C9EFD1}		Operations/MillOperations/Mill25DOp.xml

Lvl	ID	Caption	Parent	Container class / ContainerID	Solver class / SolverID	Declared in
6	TSTAbstract25DPocketOp	Abstract 25D pocket operation	TSTAbstract25DOp	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/MillOperations/Mill25DOp.xml
7	TST25DFlatLandOp	2.5D flat land	TSTAbstract25DPocketOp	TST25DCoverOp {1AF04854-47C5-4871-9F7A-51EDC18A6E9C}		Operations/MillOperations/Mill25DOp.xml
7	TST25DPocketingOp	2.5D pocketing	TSTAbstract25DPocketOp	TST25DPocketOp {1410C389-2B89-4084-BCFD-E776D55A7F2F}		Operations/MillOperations/Mill25DOp.xml
5	TSTAbstractContouringOp	Abstract contouring operation	TST3DMillOp	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/MillOperations/CurveOp.xml
6	TST2DContouringOp	2D contouring	TSTAbstractContouringOp	TSTCurve2DOp {6779C892-D689-4648-A2E9-AC3A59C25F72}		Operations/MillOperations/CurveOp.xml
7	TST25DContouringOp	2.5D contouring	TST2DContouringOp	TSTCurve2DOp {6779C892-D689-4648-A2E9-AC3A59C25F72}		Operations/MillOperations/CurveOp.xml
7	KnifeOp2D	2D knife cutting	TST2DContouringOp	TSTKnifeOp2D {4F4D1FBA-9781-4627-BE5A-41D465736892}		Operations/MillOperations/KnifeOp.xml
7	SawingOp2D	Disc cutting 2D	TST2DContouringOp	TSTSawing2DOp {B321A122-704F-4D87-BCAF-B6463710C183}		Operations/MillOperations/Sawing2D6DOp.xml
6	TST3DContouringOp	3D contouring	TSTAbstractContouringOp	TSTCurve3DOp {7A2DA7E7-E9D3-42BB-B6A2-A4172DCE4226}		Operations/MillOperations/CurveOp.xml
6	TSTJetCuttingOp	Jet cutting	TSTAbstractContouringOp	TSTCutting2DOp {52822CFE-D63A-4547-B953-8DD583A05102}		Operations/MillOperations/JetCutting.xml
5	TSTAbstractEngraveOp	Abstract engrave operation	TST3DMillOp	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/MillOperations/EngraveOp.xml
6	TSTEngraveOp	Engraving	TSTAbstractEngraveOp	TSTEngraveOp {66F71622-8106-4FE6-9A94-DB337FF15D28}		Operations/MillOperations/EngraveOp.xml
6	TSTPocketingOp	Pocketing	TSTAbstractEngraveOp	TSTPocketOp {BB586C4A-1CE5-44FB-A562-72A595980834}		Operations/MillOperations/EngraveOp.xml
7	TSTRestPocketingOp	Area rest milling	TSTPocketingOp	TSTPocketOp {BB586C4A-1CE5-44FB-A562-72A595980834}		Operations/MillOperations/EngraveOp.xml
5	TSTStringOp	Abstract string operation	TST3DMillOp	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/MillOperations/PlaneOp.xml

Lvl	ID	Caption	Parent	Container class / ContainerID	Solver class / SolverID	Declared in
6	TSTAbstractDriveOp	Abstract drive operation	TSTStringOp	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/MillOperations/DriveOp.xml
7	TSTFinishingDriveOp	Drive	TSTAbstractDriveOp	TSTDriveFinishOp {754E29F6-BC41-4F7B-9B0F-B7F73D8C7218}		Operations/MillOperations/DriveOp.xml
8	TSTRestFinishingDriveOp	Finishing drive (rest)	TSTFinishingDriveOp	TSTDriveFinishOp {754E29F6-BC41-4F7B-9B0F-B7F73D8C7218}		Operations/MillOperations/DriveOp.xml
7	TSTRoughingDriveOp	Roughing drive	TSTAbstractDriveOp	TSTDriveRoughOp {A57D346B-41EA-46AD-B8D6-9E6B921F94A1}		Operations/MillOperations/DriveOp.xml
6	TSTAbstractPlaneOp	Abstract plane operation	TSTStringOp	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/MillOperations/PlaneOp.xml
7	TSTFinishingPlaneOp	Finishing plane	TSTAbstractPlaneOp	TSTPlaneFinishOp {DC54E6A4-0E8F-4543-A8E7-CDB48EB54654}		Operations/MillOperations/PlaneOp.xml
8	TSTPlaneWaterlineOp	Complex	TSTFinishingPlaneOp	TSTPlaneWaterlineFinishOp {20DAF2AC-F037-4C3A-9AB0-0BD67D183B85}		Operations/MillOperations/Waterline.xml
9	TSTRestPlaneWaterlineFinishingOp	Complex (rest)	TSTPlaneWaterlineOp	TSTPlaneWaterlineFinishOp {20DAF2AC-F037-4C3A-9AB0-0BD67D183B85}		Operations/MillOperations/Waterline.xml
8	TSTComplexFinishingPlaneOp	Finishing plane	TSTFinishingPlaneOp	TSTPlaneFinishOp {DC54E6A4-0E8F-4543-A8E7-CDB48EB54654}		Operations/MillOperations/ComplexOp.xml
8	TSTFinishingPlaneOpForOptPlane	Finishing plane	TSTFinishingPlaneOp	TSTPlaneFinishOp {DC54E6A4-0E8F-4543-A8E7-CDB48EB54654}		Operations/MillOperations/ComplexOp.xml
8	TSTRestFinishingPlaneOp	Finishing plane (rest)	TSTFinishingPlaneOp	TSTPlaneFinishOp {DC54E6A4-0E8F-4543-A8E7-CDB48EB54654}		Operations/MillOperations/PlaneOp.xml
9	TSTRestComplexFinishingPlaneOp	Finishing plane (rest)	TSTRestFinishingPlaneOp	TSTPlaneFinishOp {DC54E6A4-0E8F-4543-A8E7-CDB48EB54654}		Operations/MillOperations/ComplexOp.xml
9	TSTRestFinishingPlaneOpForOptPlane	Finishing plane (rest)	TSTRestFinishingPlaneOp	TSTPlaneFinishOp {DC54E6A4-0E8F-4543-A8E7-CDB48EB54654}		Operations/MillOperations/ComplexOp.xml
8	TSTCrissCrossPlaneOp	Optimized plane	TSTFinishingPlaneOp	TSTPlaneFinishCrissCrossOp {1C86CE71-94AB-41C7-A60A-4665549CC4C2}		Operations/MillOperations/PlaneOp.xml
9	TSTRestCrissCrossPlaneOp	Optimized plane (rest)	TSTCrissCrossPlaneOp	TSTPlaneFinishCrissCrossOp		Operations/MillOperations/PlaneOp.

Lvl	ID	Caption	Parent	Container class / ContainerID	Solver class / SolverID	Declared in
				{1C86CE71-94AB-41C7-A60A-4665549CC4C2}		xml
7	TSTRoughingPlaneOp	Roughing plane	TSTAbstractPlaneOp	TSTPlaneRoughOp {B87DEBC4-90A1-425D-AA99-50ACD41749A6}		Operations/MillOperations/PlaneOp.xml
5	TSTAbstractWaterlineOp	Abstract waterline operation	TST3DMillOp	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/MillOperations/Waterline.xml
6	TSTFinishingWaterlineOp	Finishing waterline	TSTAbstractWaterlineOp	TSTWLFinishOp {224A6D80-F3AB-411E-B97A-60F403E98B65}		Operations/MillOperations/Waterline.xml
7	TSTCombineOp	Combine	TSTFinishingWaterlineOp	TSTWLSuperFinishOp {50AB77F3-9CBF-4A99-9B55-303D668AF7DC}		Operations/MillOperations/Waterline.xml
7	TSTComplexFinishingWaterlineOp	Finishing waterline	TSTFinishingWaterlineOp	TSTWLFinishOp {224A6D80-F3AB-411E-B97A-60F403E98B65}		Operations/MillOperations/ComplexOp.xml
7	TSTRestFinishingWaterlineOp	Finishing waterline (rest)	TSTFinishingWaterlineOp	TSTWLFinishOp {224A6D80-F3AB-411E-B97A-60F403E98B65}		Operations/MillOperations/Waterline.xml
8	TSTRestComplexFinishingWaterlineOp	Finishing waterline (rest)	TSTRestFinishingWaterlineOp	TSTWLFinishOp {224A6D80-F3AB-411E-B97A-60F403E98B65}		Operations/MillOperations/ComplexOp.xml
6	TSTRoughingWaterlineOp	Roughing waterline	TSTAbstractWaterlineOp	TSTWLRoughOp {E1BE2969-DCF9-4C3B-B131-3235D8C70565}		Operations/MillOperations/Waterline.xml
7	TSTFlatLandFinishingOp	Flat land	TSTRoughingWaterlineOp	TSTWLFlatLand {9661814C-985F-4EC1-B0CE-417E88CA49EE}		Operations/MillOperations/Waterline.xml
7	TSTRestRoughingWaterlineOp	Roughing waterline (rest)	TSTRoughingWaterlineOp	TSTWLRoughOp {E1BE2969-DCF9-4C3B-B131-3235D8C70565}		Operations/MillOperations/Waterline.xml
5	TSTPocketCladdingOp	Area cladding	TST3DMillOp	TSTPocketCladdingOp {B91BA08F-8981-45B5-8AF1-9C3B904F31CF}		Operations/MillOperations/PocketCladdingOp.xml
5	TSTCladding3DOp	Cladding 3D	TST3DMillOp	TSTCladdingWaterlineOp {FFDC70E0-D27F-4A84-B1C2-5050BB703B67}		Operations/MillOperations/Cladding3DOp.xml
5	TSTCurveCladdingOp	Curve cladding	TST3DMillOp	TSTCurveCladdingOp {AF410E0B-FC72-4B34-9B6B-85C4A8DABC15}		Operations/MillOperations/CurveCladdingOp.xml

Lvl	ID	Caption	Parent	Container class / ContainerID	Solver class / SolverID	Declared in
5	TSTDrillOp	Hole machining	TST3DMillOp	TSTDrillOp {5935E15B-69C6-4470-B8B8-8A979877DAA7}		Operations/MillOperations/DrillOp.xml
4	TSTAbstract5DOp	Abstract 5D milling operation	TSTMillOp	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/AbstractMillOP.xml
5	TSTHelicalOp	3D Helical	TSTAbstract5DOp	TSTHelicalOp {DC50D55F-DCDB-43FF-9756-F295076193E9}		Operations/MillOperations/HelicalOp.xml
6	TSTHelicalCladdingOp	Helical cladding	TSTHelicalOp	TSTHelicalCladdingOp {ADAAF276-39A0-4408-842A-AA5053316AE8}		Operations/MillOperations/HelicalCladdingOp.xml
5	TSTUni3dOp	3D Surfacing	TSTAbstract5DOp	TSTUni3dOp {94D51F4A-BFFB-485C-83DF-23ACB2C94D60}		Operations/MillOperations/Uni3DOp.xml
5	TSTWaterJetCutting4d	4D Jet cutting	TSTAbstract5DOp	TSTWaterJetCuttingOp {776687C7-3A2D-41C0-88F6-FA0419F9C62D}		Operations/MillOperations/WaterJetCuttingOp.xml
5	ToolEnd5DMachiningOp	5D Contouring	TSTAbstract5DOp	TSTIso5DOp {F1566DB8-C27B-48A8-AC02-6B5C58412EC8}		Operations/MillOperations/ToolEnd5DMachining.xml
6	ToolEnd4DMachiningOp	4D Contouring	ToolEnd5DMachiningOp	TSTIso5DOp {F1566DB8-C27B-48A8-AC02-6B5C58412EC8}		Operations/MillOperations/ToolEnd5DMachining.xml
6	KnifeOp5D	6D Knife cutting	ToolEnd5DMachiningOp	TSTKnifeOp5D {3D34BF2B-863D-418A-BC2E-C241D48AB15C}		Operations/MillOperations/KnifeOp.xml
6	ChamferingMillOp	Chamfering	ToolEnd5DMachiningOp	TSTChamferingOp {D98363DF-DE21-4A51-A4C6-412FAFF382DD}		Operations/MillOperations/ChamferingOp.xml
6	SawingOp5D	Disc cutting 6D	ToolEnd5DMachiningOp	TSTSawing5DOp {C9B37EDE-41A2-4464-BA7B-42F171CC446E}		Operations/MillOperations/Sawing2D6DOp.xml
6	TCutting5DOp	Jet cutting 5D	ToolEnd5DMachiningOp	TSTCutting5DOp {74C4D06A-D78B-4A60-A8C1-FE39A4CC6E37}		Operations/MillOperations/ToolEnd5DMachining.xml
6	SawingOp	Sawing	ToolEnd5DMachiningOp	TSTSawingOp5D {F4903DDF-32B7-4461-952C-28321A5FF105}		Operations/MillOperations/Sawing.xml
5	TSTMultiAxisOp	5D Roughing waterline	TSTAbstract5DOp	TIntMillOP {05436516-7F64-4BA5-8DBF-D98BCDE2B76F}	TIntMillOP {956D4402-BF52-4218-B3A3-52456DEA634C}	Operations/MillOperations/MultiaxisOp.xml
5	TSTUni5DOp	5D Surfacing	TSTAbstract5DOp	TSTUni5DOp {4E14343C-CB17-		Operations/MillOperations/Uni5DOp.xml

Lvl	ID	Caption	Parent	Container class / ContainerID	Solver class / SolverID	Declared in
				49B7-B833-813991124283}		
6	TSTUni4DOp	4D Surfacing	TSTUni5DOp	TSTUni5DOp {4E14343C-CB17-49B7-B833-813991124283}		Operations/MillOperations/Uni5DOp.xml
6	TSTCladding5DOp	Cladding 5D	TSTUni5DOp	TSTCladding5DOp {AF57AC8F-671A-4A20-A618-C41C043CD8A7}		Operations/MillOperations/Cladding5D.xml
7	TSTSurfaceSprayOp	Surface spraying	TSTCladding5DOp	TSurfaceSprayingOp {9E7F6C12-BFF7-40B6-A894-8A6F88880391}		Operations/MillOperations/SprayPainting.xml
6	TSTUni5DOp4D	Rotary waterline	TSTUni5DOp	TSTUni5DOp4D {C94FD54D-0284-4FED-A7C5-E1B1C63A33CA}		Operations/MillOperations/Uni5DOp.xml
5	ToolEnd5DMachiningOpTPLinker	6D Contouring	TSTAbstract5DOp	TSTiso5DOpTPLinker {430E12DD-8313-4AF3-A5DC-C8E706FE5B62}		Operations/MillOperations/ToolEnd5DMachiningTPLinker.xml
6	TSTContourSprayOp	Contour spraying	ToolEnd5DMachiningOpTPLinker	TContourSprayingOp {9AAB2B58-D711-433D-B34E-3053C5050927}		Operations/MillOperations/SprayPainting.xml
5	TSTBladeMillingOP	Blade Milling	TSTAbstract5DOp	TSTBladeMillingOP {ECCB372B-AC5F-4277-BECF-F3334107E0C6}		Operations/MillOperations/BladeMillingOP.xml
5	TSTCornerRestMachiningOP	Corners cleanup	TSTAbstract5DOp	TSTCornerRestMachiningOP {08E472D8-DAA9-4B74-9F2E-6691DE2383B4}		Operations/MillOperations/CornerRestMachiningOP.xml
5	TSTDiscRoughingOp	Disc Roughing	TSTAbstract5DOp	TSTDiscRoughingOp {9020A57A-1B99-4C10-8B5F-0C3995F5AA14}		Operations/MillOperations/DiscRoughingOp.xml
5	HoleMachiningOp	Hole machining	TSTAbstract5DOp	TSTDrillOp5D {7754D5E8-B04D-45CC-8904-76BD02E6051C}		Operations/MillOperations/HoleMachiningOperation.xml
6	LatheHoleMachiningOp	Lathe hole machining	HoleMachiningOp	TSTHoleLatheMachiningOp {AFB34779-8CA1-4729-AD46-39C246237FD5}		Operations/MillOperations/HoleMachiningOperation.xml
5	TSTImpellerMachiningOp	Impeller Machining	TSTAbstract5DOp	TIntMillOP {05436516-7F64-4BA5-8DBF-D98BCDE2B76F}	TIntMillOP {E34276BE-0345-4053-8D5F-8F540ADE793F}	Operations/MillOperations/ImpellerMachiningOp.xml
5	TSTFBMMillOp	Mill features machining	TSTAbstract5DOp	TFBMMillOp {3BE9B3B3-65B7-4C40-A135-C8BAA3E69AB0}		Operations/MillOperations/FBMMillOp.xml

Lvl	ID	Caption	Parent	Container class / ContainerID	Solver class / SolverID	Declared in
5	TAbstractProbingOp	Mill part probing	TSTAbstract5DOp	TSTMeasuringOp {CDCF31D1-89AF-4099-B9FD-C0F3F505DCC7}		Operations/MillOperations/MeasuringOp.xml
6	TMillPartProbingOP	Part probing	TAbstractProbingOp	TMillPartProbingOp {3357DF18-9BAD-413A-94AA-F660798DCB95}		Operations/MillOperations/MeasuringOp.xml
6	TMillToolProbingOP	Tool probing	TAbstractProbingOp	TMillToolProbingOP {30DDF150-F104-48C9-A7A8-1451A5A4BA94}		Operations/MillOperations/MeasuringOp.xml
6	TAbstractTurnProbingOp	Turn probing	TAbstractProbingOp	TSTTurnMeasuringOp {6BC13F5D-52DF-437C-990B-F1FB5719D177}		Operations/MillOperations/MeasuringOp.xml
7	TTurnPartProbingOP	Turn part probing	TAbstractTurnProbingOp	TTurnPartProbingOP {6A35D2E5-6C73-40F4-937B-415F0E2CFBFE}		Operations/MillOperations/MeasuringOp.xml
7	TTurnToolProbingOP	Turn tool probing	TAbstractTurnProbingOp	TTurnToolProbingOP {2464E142-C74B-42AD-AF19-D605E65F5E97}		Operations/MillOperations/MeasuringOp.xml
5	TSTMorphOp	Morph	TSTAbstract5DOp	TSTMorphOp {50A70FF7-80D3-4D36-A49F-A0A69AF853DE}		Operations/MillOperations/Contour6dOp.xml
6	TSTMorphOp4D	Morph 4D	TSTMorphOp	TSTMorphOp4D {AAB17600-1055-48F4-9C93-27C6E6C63C3C}		Operations/MillOperations/MorphOp.xml
6	TSTMorphSprayOp	Morph spraying	TSTMorphOp	TMorphSprayingOp {2D717258-1107-47E2-87CA-AD3B93D5F15C}		Operations/MillOperations/SprayPainting.xml
5	TSTNonPlanarPrintingOp	Non-planar slicing	TSTAbstract5DOp	TSTNonPlanarPrintingOp {DC4744C8-5990-42C9-B6BC-18777BD89FB8}		Operations/MillOperations/NonPlanarPrintingOp.xml
5	TSTOrthogonalMachiningOp	Orthogonal machining	TSTAbstract5DOp	TSTOrthOp {0FD22D59-BF37-46D2-A7BF-F6F36B88F83D}		Operations/MillOperations/OrthogonalOp.xml
5	TSTPencilOp	Pencil	TSTAbstract5DOp	TSTPencilOp {2A1AD087-60B0-498A-9290-F8F846756839}		Operations/MillOperations/PencilOp.xml
5	TPointWeldingOp	Point welding	TSTAbstract5DOp	TSTPointWeldingOp {587BA0E4-FAA8-46E8-B1AD-25A2A054B585}		Operations/MillOperations/PointWeldingOp.xml
5	RotaryMachiningOp	Rotary finishing	TSTAbstract5DOp	TSTRotaryOp {7362B15E-9DA8-4038-AFA8-BB096A8B8D13}		Operations/MillOperations/RotaryMachining.xml

Lvl	ID	Caption	Parent	Container class / ContainerID	Solver class / SolverID	Declared in
6	TSTRotarySprayOp	Rotary spraying	RotaryMachiningOp	TRotarySprayingOp {27955057-DB5E-47FC-9630-E5E9BFFED8BB}		Operations/MillOperations/SprayPainting.xml
5	RoughingRotaryMachiningOp	Rotary roughing	TSTAbstract5DOp	TSTRoughingRotaryOp {3DACED8F-10B5-4FB9-8FFC-79E8D36D1B28}		Operations/MillOperations/RoughingRotaryMachining.xml
5	TSTScallopOp	Scallop	TSTAbstract5DOp	TSTScallopOp {AFE96A9B-10E9-4F14-9761-D9EB62D1B61A}		Operations/MillOperations/ScallopOp.xml
6	TSTGeodesicOp	5D by meshes	TSTScallopOp	TSTGeodesicOp {EB52F74F-3B58-403F-82A2-995C532248A5}		Operations/MillOperations/GeodesicOp.xml
5	TSTSwarfOp	Swarf	TSTAbstract5DOp	TSTSwarfOp {0F267035-EBA5-4A74-B389-39719449288A}		Operations/MillOperations/SwarfOp.xml
5	TSTWaterlineOp	Undercut waterline	TSTAbstract5DOp	TSTWaterlineOp {C5678D0B-8C81-4932-997A-FD7AC56B483D}		Operations/MillOperations/WaterlineOP.xml
5	TWeldingOp5D	Welding 5D	TSTAbstract5DOp	TSTWeldingOp {877BAB4B-E53C-4472-9F1C-5A95CD3C7BA8}		Operations/MillOperations/WeldingOp.xml
5	TWeldingOp6D	Welding 6D	TSTAbstract5DOp	TSTWelding6DOp {6546C5E3-1294-419D-88D5-C7899C3D38AE}		Operations/MillOperations/Welding6DOp.xml
4	TMillOpWithDisabledProps	Aux Op	TSTMillOp	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/AuxOp.xml
5	TSTAbstractPickAndPlace	Pick and place	TMillOpWithDisabledProps	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml
6	TPNPbySelectedNode		TSTAbstractPickAndPlace	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml
7	TPNPwithGripperTool		TPNPbySelectedNode	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml
8	TPNPCalculator		TPNPwithGripperTool	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml
9	TPNPwithOutputMode		TPNPCalculator	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml
10	TPNPwithClamps		TPNPwithOutputMode	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml

Lvl	ID	Caption	Parent	Container class / ContainerID	Solver class / SolverID	Declared in
11	TPNPwithTurnTool	Bar feeding	TPNPwithClamps	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/BarFeeding.xml
12	TSTBarFeeding	Bar feeding	TPNPwithTurnTool	TSTBarFeeding {C5B4DA2F-4B75-4E86-A500-EC7D5496BAD9}		Operations/BarFeeding.xml
11	TSTPointPickAndPlaceOP	Point Pick and place	TPNPwithClamps	TSTPointPickAndPlaceOp {64F4FD79-26E6-4669-8CE1-4799842894FA}		Operations/PointPickAndPlace.xml
11	TPNP2NextOp		TPNPwithClamps	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml
12	TPNPApproachCorrector		TPNP2NextOp	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml
13	TPNPwithPickCS		TPNPApproachCorrector	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml
14	TPNPwithActions		TPNPwithPickCS	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml
15	TPNPwithSafeSurf		TPNPwithActions	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml
16	TPNPwithEngageVectors		TPNPwithSafeSurf	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml
17	TPNPwithMPL		TPNPwithEngageVectors	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml
18	TSTPlace2Next	Place to next stage	TPNPwithMPL	TSTPlace2Next {C448DE4A-8F0D-40B1-A25D-FD02DA45BF61}		Operations/PickAndPlace20.xml
18	TPNPwithRemoveWorkpiece		TPNPwithMPL	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml
19	TSTPickAndPlace	Pick and place	TPNPwithRemoveWorkpiece	TSTPickAndPlace {3D9EDEA9-E890-4EEC-B563-043BD999D144}		Operations/PickAndPlace20.xml
12	TPNPwithSyncSpindles		TPNP2NextOp	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml
13	TPNP4SwissLathes		TPNPwithSyncSpindles	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml

Lvl	ID	Caption	Parent	Container class / ContainerID	Solver class / SolverID	Declared in
14	TPNPwithTurnEngages		TPNP4SwissLathes	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml
15	TSTSubTakeoverOp	Abstract pick-n-place	TPNPwithTurnEngages	TSTSubTakeoverOp {BC7F9D0D-781C-4651-8468-7340E4456C06}		Operations/PickAndPlace20.xml
15	TPNPwithCannedCycle		TPNPwithTurnEngages	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml
16	TPNPwithSyncOp		TPNPwithCannedCycle	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/PickAndPlace20.xml
17	TSTTakeoverMTM	Turn take over	TPNPwithSyncOp	TSTTakeoverMTM {15B80C67-78CE-4F34-9A8B-BD2392203AB4}		Operations/PickAndPlace20.xml
14	TPNPwithTurnSetupLCS		TPNP4SwissLathes	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/TurnPNP.xml
15	TTurnPNPwithPickPosition		TPNPwithTurnSetupLCS	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/TurnPNP.xml
16	TTurnPNPwithReturn		TTurnPNPwithPickPosition	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/TurnPNP.xml
17	TTurnPNPwithSwissPlace		TTurnPNPwithReturn	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/TurnPNP.xml
18	TSTTurnPNP	Sub spindle working	TTurnPNPwithSwissPlace	TSTTurnPNP {8922288C-7247-470B-85A8-266FE9690AFB}		Operations/TurnPNP.xml
5	TSTSwissWaitOp	Wait tool	TMillOpWithDisabledProps	TSTSwissWaitOp {D7DCF7B3-9F48-4DA9-985B-6063F28F51F1}		Operations/AuxOp.xml
4	TSTFaceMillingOp	Face Milling	TSTMillOp	TSTFaceMillingOp {F4DE00D1-7AE3-468E-BA5C-ED3C8275CBF8}		Operations/MillOperations/FaceMillingOp.xml
3	TAbsWireEDMContouringOp	Wire EDM Contouring	TSTAbstractMillOp	TSTAbstractTechOp {AA0BC156-ECA0-45A4-A74D-D705E890F9F3}		Operations/MillOperations/WireEDMContouring.xml
4	WireEDM2dContouringOp	Wire EDM 2D Contouring	TAbsWireEDMContouringOp	TSTWireEDM2dOp {7F43BE1B-8915-47B9-805C-64EB234343C7}		Operations/MillOperations/WireEDMContouring.xml
4	WireEDM4dContouringOp	Wire EDM 4D Contouring	TAbsWireEDMContouringOp	TSTWireEDM4dOp {296FBA85-2C25-439C-83C5-AB5037034988}		Operations/MillOperations/WireEDMContouring.xml

Lvl	ID	Caption	Parent	Container class / ContainerID	Solver class / SolverID	Declared in
1	TSTTechOpGroup	Operations group	TOperationDescriptor	TSTTechOpGroup {F36447FB-EB16-4A8A-8190-1DB64431F6A0}		Operations/MillOperations/Groups.xml
2	TSTFBMRootGroup	FBM Mill	TSTTechOpGroup	TSTFBMRootGroup {93B9EADA-2102-45A1-B184-E368CAC8F32D}		Operations/MillOperations/Groups.xml
2	TSTTechMillOpGroupRaw	Group	TSTTechOpGroup	TSTTechMillOpGroupFin {857E6E2C-76ED-4B69-8A88-7535346A4E88}		Operations/MillOperations/Groups.xml
2	TSTTechMillOpGroupFin	Group	TSTTechOpGroup	TSTTechMillOpGroupFin {857E6E2C-76ED-4B69-8A88-7535346A4E88}		Operations/MillOperations/Groups.xml
3	TSTComplexFinishingOp	Finishing complex	TSTTechMillOpGroupFin	TSTComplexFinishingOp {F55EC5B0-BA4A-483A-87BB-B027D80C806C}		Operations/MillOperations/ComplexOp.xml
4	TSTRestComplexFinishingOp	Complex (rest)	TSTComplexFinishingOp	TSTComplexFinishingOp {F55EC5B0-BA4A-483A-87BB-B027D80C806C}		Operations/MillOperations/ComplexOp.xml
3	TSTOptimizedPlaneOp	Finishing optimized plane	TSTTechMillOpGroupFin	TSTOptimizedPlane {48878E1A-E7EF-43AD-800B-BA9AFEA6FE5E}		Operations/MillOperations/ComplexOp.xml
4	TSTRestOptimizedPlaneOp	Optimized plane (rest)	TSTOptimizedPlaneOp	TSTOptimizedPlane {48878E1A-E7EF-43AD-800B-BA9AFEA6FE5E}		Operations/MillOperations/ComplexOp.xml
3	TSTRootGroup	Machine	TSTTechMillOpGroupFin	TSTRootGroup {815B30BF-4E80-4337-A6D0-D58C80526F7D}		Operations/MillOperations/Groups.xml
2	TSTTechMillOpGroupRest	Group	TSTTechOpGroup	TSTTechMillOpGroupFin {857E6E2C-76ED-4B69-8A88-7535346A4E88}		Operations/MillOperations/Groups.xml
3	TSTMultiplyGroup	Multiply group	TSTTechMillOpGroupRest	TMultiplyGroup {97D77FBC-0046-4A1C-8819-64D2CACC0042}		Operations/MillOperations/Groups.xml
2	TSTFeatureOpGroup	Multiply Group	TSTTechOpGroup	TSTFeatureOpGroup {8C369EA7-83E9-4B8B-BE7C-365714A96CAF}		Operations/MillOperations/Groups.xml
2	TSTStageOpGroup	Setup stage	TSTTechOpGroup	TSTStageOpGroup {16253702-D621-4ABA-AE09-1BAA3F59D937}		Operations/MillOperations/Groups.xml
2	TSTTombOpGroup	Tomb group	TSTTechOpGroup	TSTTombOpGroup {072E54DE-A8A5-4028-B3D0-E1FC5D0FC03A}		Operations/MillOperations/Groups.xml

Lvl	ID	Caption	Parent	Container class / ContainerID	Solver class / SolverID	Declared in
2	TSTCustomPartOp		TSTTechOpGroup	TSTPartOpGroup {2802AC38-B264-47F5-897A-55C520C19F67}		Operations/MillOperations/Groups.xml
3	TSTPartOpCopy	Copy of part	TSTCustomPartOp	TSTPartOpCopy {B40179A7-FDE6-4428-B44E-967867CA47AF}		Operations/MillOperations/Groups.xml
3	TSTPartOpGroup	Part	TSTCustomPartOp	TSTPartOpGroup {2802AC38-B264-47F5-897A-55C520C19F67}		Operations/MillOperations/Groups.xml
2	TSTFBMOpGroup		TSTTechOpGroup	TSTFBMOpGroup {5ED0E2BA-19EB-45AD-B3E5-6D68E972A1DC}		Operations/MillOperations/Groups.xml

---

Back to Operation XML descriptors | index

Machine descriptors

This area covers describing a machine — its file layout, kinematic scheme and the parameters that drive how its toolpath becomes NC code. It builds on the XMLProperties framework.

Most machines are built interactively with the MachineMaker application; the guides here are for understanding a machine and for the special cases still authored, or fine-tuned, by hand.

- Machine descriptors — start here: a machine's file layout, a complete simple example, the kinematic node chain, and each section of the descriptor (Description, Control Parameters, Schema, state parameters, Tooling, Machine dimensions).
- Machine node reference — the look-up companion: the full property tables for every node type (`TMachineNode`, `TMachineAxis`, connectors, state parameters, `VisualProperties`), plus coordinate conventions and units.
- Machine Setup parameters reference — what every parameter on the machine's *Control Parameters* tab does, each with the XML fragment that overrides it.
- Advanced machine topics — node matrices, switchable / detachable nodes, turrets, tool/workpiece connectors, sub-machines and control channels.
- 3D models for machines — preparing, referencing and aligning the node geometry (`.osd` / `.stl`).

See the XMLProperties framework for the shared ground rules (don't edit `Supplement`, where the shipped files live, example conventions), and the documentation root for the intended audience.

Machine descriptors

> XML paths in this article refer to configuration files in the installed CAM system (`$(SCHEMAS_FOLDER)`), not manual attachments. Available files depend on the version and configuration.

A machine descriptor is, like an operation, a large `ComplexType` — but its most distinctive part is the kinematic scheme: a nested tree of axes and holders that describes how the machine moves. This chapter walks through a machine descriptor from the outside in: the files it is made of, a complete simple example, the idea of the kinematic chain, and then each section in turn. For the exhaustive property list of each node type, it points you to the machine node reference.

> Most machines are built with MachineMaker, not by hand. A dedicated application, MachineMaker, lets you create a machine *interactively*: for the common kinematic-scheme types it produces the descriptor for you, with no manual XML editing. Reserve hand-written XML (and manually prepared 3D models) for special, complex or unusual machines that fall outside what MachineMaker covers. This chapter documents that manual path — but if MachineMaker can build your machine, prefer it.

> Companion pages for machines — read alongside this chapter:
>
> - Machine node reference — the full property tables for every node type (`TMachineNode`, `TMachineAxis`, connectors, state parameters, `VisualProperties`) plus coordinate conventions and units.
> - Machine Setup parameters reference — what every parameter on the machine's *Control Parameters* tab does (arcs, singularities, rotary transformations, 5-axis compensation, tool change, simulation, ⋯), each with the XML fragment that overrides it.
> - Advanced machine topics — node matrices, switchable/detachable nodes, turrets, tool/workpiece connectors, and control channels & sub-machines.
> - 3D models for machines — preparing, referencing and aligning the node geometry (`.osd` / `.stl`).

A machine is a folder of files

A machine is not a single object buried in the application — it is a small set of files you can point at:

- the descriptor — one `.xml` file that *is* the machine: its identity, control parameters and kinematic scheme;
- its 3D models — the `.osd` / `.stl` geometry the descriptor references through `ImageFile` (per node) and the `Image` / `Icon` header members, conventionally kept in an `Images\` subfolder beside the descriptor (see 3D models for machines);
- optionally machine defaults (`*.usrdef`) that tweak shipped machines.

The shipped files under `Machines\` are your reference. They define the machine type library — the abstract machines and the building-block types every machine is assembled from — loaded via `Machines/MachinesConfig.xml` into the **`Machines`** namespace. The two you will read most are `Machines/AbstractMachine.xml` (the common base every machine derives from) and `Machines/MachineTypes.xml` (the node, axis, holder and parameter types).

A *concrete* machine, by contrast, is an individual file loaded on its own (below). As with operations, you do not edit `Supplement`: a custom machine is an individual file in the scanned machine folders (`$(SCHEMAS_FOLDER)` / `$(CUSTOM_SCHEMAS_FOLDER)`), normally produced by the machine-building tools (see §7). This chapter explains the descriptor syntax those machines are built from.

How a machine descriptor is loaded

Machines are loaded differently from operations, and the difference shapes how you author them.

- The type library loads once. `MachinesConfig.xml` loads the shared machine type library — the abstract machines, and the node, axis, holder and parameter building blocks (below). This is common to every machine.
- Each machine loads in isolation. A concrete machine is its own XML file. When the system needs a particular machine, it builds a *fresh* `Machines` namespace on top of the shared type library, includes that one machine file, and then applies the machine defaults (`Machines/MachinesDefaults.xml`, which pulls in any `*.usrdef`). Each machine gets its own private namespace.
- Identifiers are local to one machine. Because machines never share a namespace, two different machine files may safely reuse the same `SCType IDs`. You do not need globally-unique member names — only names unique *within* your machine.
- The library is lazy. Installations can hold thousands of machines, so the machine library only *scans* the predefined machine folders for lightweight metadata (enough to list and search them) and fully loads a machine's descriptor only when that machine is actually opened. Keep this in mind: a machine file is parsed on demand, in isolation, not all-at-once at startup.

The practical consequence for authoring: your machine file is a self-contained `<SCCollection>` in the `Machines` namespace that derives from a library type and overrides/extends it. It lives in a scanned machine folder, not in `Supplement`.

The building blocks

`MachineTypes.xml` defines the reusable types a machine is assembled from. There are only a handful to know; the node reference lists every property of each.

Type	Role
`TMachineNode`	Base for anything in the scheme tree — a *rigid* part (bed, column, table). Has an `ID`, a transform

Type	Role
	`Matrix`, visual properties and an optional `ImageFile` (`.osd`/`.stl` geometry).
`TMachineAxis`	A node that moves: `AxisType` (`Linear`/`Rotary`), a `Direction` (linear axes only), `Scale`, `RapidFeed`, and a `ParameterName` linking it to a state parameter.
`TMachineStateParameter`	The controllable value an axis is driven by (a position, a spindle, ⋯): `Address`, limits, control mode. Not a scheme node — axes link to it by name.
`TToolHolderNode` / `TWorkpieceHolderNode`	The sockets that carry the tool / the workpiece, with `SupportedToolTypes`. Specializations: `TMillToolHolder`, `TLatheCutterHolder`, `TJetCutterHolder`, ⋯

> Naming note. The type IDs `TToolHolderNode` and `TWorkpieceHolderNode` appear in the user interface as "Tool connector" and "Workpiece connector". They are the same thing — the socket where the tool, respectively the workpiece, attaches to the kinematic chain.

A few predefined axis and state-parameter types also exist (`TToolAxisX/Y/Z`, `TLatheSpindle`, `TRotaryTable`, `TAxisXPosition`, ⋯), but they are legacy and optional: introduced early to save typing through inheritance, they save little and force you to chase down which defaults they hide. Prefer declaring axes and parameters explicitly from `TMachineAxis` / `TMachineStateParameter`, so the values that matter are visible right in your machine file. (Template types such as the turret ones in §3 are the exception — there inheritance genuinely pays off.)

A complete simple machine

Concrete machines derive from `AbstractMachine` (or a closer base like `AbstractMillMachine`), which already provides every section a machine has:

Section	Contents
`DescriptionPlaceHolder`	Identity & metadata: `Name`, `Comment`, `Group`, `Developer`, `NCSystem`, postprocessor file (`SPPFile`), interpreter file, measurements system.

Section	Contents
`ControlData`	Control/behaviour parameters: arc support, rotary transformations, 5-axis compensation, local CS (ORIGIN) handling, tool-frame output format.
`MachineStateParameters`	The list of controllable parameters (the axes' state).
`Schema`	The kinematic tree itself.
`Coolants`	Available coolant tubes.
`Simulation`	Simulation method/resolution and collision settings.
`ToolChange`	Tool-change position, timing and output mode.
`MachineDimensions`	A tidy place to collect the scheme's key defining parameters.

The two sections you author for almost every machine are `MachineStateParameters` (the values the controller drives) and `Schema` (the geometry that moves with them). Here is a complete small 3-axis mill in which the table traverses in X while the tool head carries Y and Z — a common bed-type layout, and a good illustration that *both* branches can move:

```
<MachineStateParameters>
    <SCType ID="AxisXPos" Caption="X" type="TMachineStateParameter">
        <Address DefaultValue="X"/><Group DefaultValue="LinearAxis"/>
        <Min DefaultValue="-300"/><Max DefaultValue="300"/>
    </SCType>
    <SCType ID="AxisYPos" Caption="Y" type="TMachineStateParameter">
        <Address DefaultValue="Y"/><Group DefaultValue="LinearAxis"/>
        <Min DefaultValue="-200"/><Max DefaultValue="200"/>
    </SCType>
    <SCType ID="AxisZPos" Caption="Z" type="TMachineStateParameter">
        <Address DefaultValue="Z"/><Group DefaultValue="LinearAxis"/>
        <Min DefaultValue="0"/><Max DefaultValue="400"/>
<InitialValue DefaultValue="400"/>
    </SCType>
    <SCType ID="SpindlePos" Caption="Spindle" type="TMachineStateParameter">
        <Address DefaultValue="S"/><Group DefaultValue="RotaryAxis"/>
<AxisControl DefaultValue="Manual"/>
    </SCType>
</MachineStateParameters>
```

```
<Schema>
    <SCType ID="Base" Caption="Bed" type="TMachineNode" IsFloor="True">
        <ImageFile DefaultValue="Images\bed.osd"/>

        <!-- TOOL branch: Y carries Z carries the spindle carries the tool -->
        <SCType ID="AxisY" Caption="Axis Y" type="TMachineAxis">
            <ParameterName DefaultValue="AxisYPos"/>
            <AxisType DefaultValue="Linear"/>
            <Direction><X DefaultValue="0"/><Y DefaultValue="1"/><Z
DefaultValue="0"/></Direction>
            <SCType ID="AxisZ" Caption="Axis Z" type="TMachineAxis">
                <ParameterName DefaultValue="AxisZPos"/>
                <AxisType DefaultValue="Linear"/>
                <Direction><X DefaultValue="0"/><Y DefaultValue="0"/><Z
DefaultValue="1"/></Direction>
                <SCType ID="Spindle" Caption="Spindle" type="TMachineAxis">
                    <ParameterName DefaultValue="SpindlePos"/>
                    <AxisType DefaultValue="Rotary"/>
                    <SCType ID="Tool" Caption="Tool" type="TMillToolHolder">
                        <SpindleParamID DefaultValue="SpindlePos"/>
                    </SCType>
                </SCType>
            </SCType>
        </SCType>

        <!-- WORKPIECE branch: the table traverses in X (physically -X, see the note
below) -->
        <SCType ID="AxisX" Caption="Axis X" type="TMachineAxis">
            <ParameterName DefaultValue="AxisXPos"/>
            <AxisType DefaultValue="Linear"/>
            <Direction><X DefaultValue="-1"/><Y DefaultValue="0"/><Z
DefaultValue="0"/></Direction>
            <SCType ID="Table" Caption="Table" type="TMachineNode">
                <ImageFile DefaultValue="Images\table.osd"/>
                <SCType ID="Workpiece" Caption="Workpiece"
type="TWorkpieceHolderNode"/>
            </SCType>
        </SCType>
    </SCType>
</Schema>
```

Reading this tree: the bed is the fixed base (`IsFloor="True"`, see the kinematic scheme). Its tool branch is Y → Z → spindle → tool holder: move Y and everything below it moves, and the leaf `TMillToolHolder` is where the cutting tool attaches. Its workpiece branch is X → table → workpiece: the X axis carries the

table, and the `TWorkpieceHolderNode` on the table is where the part is held. These two leaves are the *Tool connector* / *Workpiece connector* of the UI. Each moving node names the state parameter that drives it through `ParameterName`, and all four parameters are declared in `MachineStateParameters`.

> Note the X axis's `Direction` of (−1, 0, 0). It sits in the workpiece branch, so its `Direction` is the *table's* physical travel — world −X. Moving the part in −X is equivalent to moving the tool in +X, so this produces the conventional machine X with a right-handed axis triple. (A tool-branch axis, by contrast, uses its travel direction directly.) This workpiece-branch sign rule is detailed in §4.

The rest of this chapter walks each section in turn.

Identity and metadata

`AbstractMachine` carries the header members every machine needs — `GUID`, `Priority`, `Name`, `Comment`, `Group`, `Image`, `Icon` — and the `DescriptionPlaceHolder` identity fields (`Developer`, `NCSystem`, postprocessor `SPPFile`, interpreter file, and the Measurements system that fixes the machine's units). A derived machine overrides these the same way an operation does:

```
<SCType ID="AbstractMillMachine" Caption="Abstract Mill
Machine" type="AbstractMachine">
    <GUID     DefaultValue="{FA164E91-DF7D-474B-B335-8EE7F2C34E41}"/>
    <Priority DefaultValue="1000"/>
    <Group    DefaultValue="Milling"/>
    ...
</SCType>
```

(real example: `Machines/AbstractMachine.xml`)

`Group` is a `TMachineGroup` enum: `Milling`, `Lathe`, `LatheMilling`, `JetCutter`, `WireEDM`, `Robot`, `Unknown`.

The Measurements system (metric vs. imperial) fixes the units of every `Matrix` translation and all referenced geometry — see Coordinate conventions and units.

Control parameters

`ControlData` is a deep section of switches that tell the engine and postprocessor what the machine can do. You typically override only the parts relevant to your machine. Every one of these parameters is explained in detail — with the XML fragment that overrides it — in Machine Setup parameters reference; this section is just an overview of what lives there:

- Arcs — whether arc output is allowed and in which planes, min length, max radius, spatial arcs.
- Rotary transformations — polar/cylindrical interpolation availability, TCPM support and defaults.

- 5-axis compensation — tool-tip / workpiece-zero / coordinate-system compensation for indexed and continuous modes.
- Local CS (ORIGIN) — how G54-style origins behave.
- Tool-frame output — the rotation/orientation format the postprocessor expects (normal vector, quaternion, Euler variants, …).

Many of these defaults are computed with the expression language so they stay consistent — for instance the rotation sequence derived from the chosen tool-frame format via a large `SWITCH` (see §5).

Machine state parameters

`MachineStateParameters` lists the quantities the controller drives. You add them by overriding the inherited section and declaring members of `TMachineStateParameter` (or a predefined specialization):

```
<MachineStateParameters>
    <SCType ID="AxisXPos" type="TAxisXPosition"/>
    <SCType ID="AxisYPos" type="TAxisYPosition"/>
    <SCType ID="AxisZPos" type="TAxisZPosition">
        <InitialValue DefaultValue="100"/>
    </SCType>
    <SCType ID="AxisSPos" Caption="Spindle Position" type="TMachineStateParameter">
        <Address  DefaultValue="S"/>
        <Group    DefaultValue="RotaryAxis"/>
        <Priority DefaultValue="1"/>
        <Order    DefaultValue="4"/>
    </SCType>
</MachineStateParameters>
```

(real example: `Machines/AbstractMachine.xml`)

The properties you set most often are `Address`, `Group`, and `Min`/`Max`/`Incr`; a few others control simulation and tool-change behaviour (`AxisControl`, `HasBrake`, `SupportShortestPathRotation`, `DesignTimeValue`, `ControlWithMap`, …). The complete property table, with notes on each, is in the node reference → `TMachineStateParameter`.

Two places to declare a state parameter

A `TMachineStateParameter` may be declared in either of two places:

- in the global **`MachineStateParameters`** section (as above), or
- inside the description of a node or axis in the `Schema`.

Declaring it on a node is what enables switchable (optional) nodes: when a node can be present or absent, its state parameter travels with it. The contents of `MachineStateParameters` are then composed

dynamically — parameters appear or disappear according to which nodes are actually loaded for the current machine configuration. Put a parameter in the global section when it always exists; put it on the node when it should come and go with that node.

The kinematic scheme

`Schema` is where the machine's geometry comes alive. It is a tree of nodes nested by containment: a child node sits on, and moves with, its parent. The nesting *is* the kinematic chain. A scheme always has two branches off the base — one that carries the tool and one that holds the workpiece — and both are mandatory (see the complete example above).

Key points when building a scheme:

- Each moving node binds to a state parameter through `ParameterName` (the spindle is just a rotary axis whose state parameter is in the `RotaryAxis` group, and the tool holder points at it with `SpindleParamID`). All the parameters a scheme names must exist in `MachineStateParameters`.
- `AxisType` is `Linear` or `Rotary`. **`Direction`** applies to linear axes only — there it is the direction of travel. For rotary axes `Direction` is *not used*: rotation is always about the Z axis of the axis's own local coordinate system, so orient the node (via its `Matrix`) to place that Z where the rotation axis should be.
- **`Scale`** is a dimensionless multiplier on the axis's state-parameter value (a length for a linear axis, an angle in radians for a rotary one); a negative `Scale` inverts the axis. See the node reference for the full `TMachineAxis` property list, including `RapidFeed`, `Channel` and `DesignTimeAxisValue`.
- The chain must end in a holder so the tool (and, on the workpiece side, the part) has a place to sit. A tool holder's `XAxisID`/`YAxisID`/`ZAxisID` and `ToolAxisID`/`ToolAxisValue` carry extra meaning — see §4 and the connector reference.
- Visual `Color`/geometry per node is set through the inherited `VisualProperties` (see the node reference and 3D models for machines).

One parameter, several nodes — the shared-axis trick

An axis lives in two places that play different roles: an axis node (`TMachineAxis`) in the `Schema` — the *geometry* (where it sits, its `Direction`, `Matrix`, 3D model) — and a state parameter (`TMachineStateParameter`) — the single *controllable value* (its `Address`, limits, control mode). They are linked by the node's `ParameterName` matching the parameter's `ID`.

The powerful consequence: several axis nodes can share one state parameter, so one controlled value drives several pieces of geometry at once. The classic case is a 3-jaw chuck — three jaws, each its own axis node pointing radially in its own rotated direction, all driven by a single `JawPos`:

```
<!-- one shared state parameter -->
<MachineStateParameters>
    <SCType ID="JawPos" Caption="Jaws" type="TMachineStateParameter">
```

```
        <AxisControl DefaultValue="Manual"/>
        <Min DefaultValue="0"/><Max DefaultValue="100"/>
<InitialValue DefaultValue="50"/>
    </SCType>
</MachineStateParameters>

<!-- three jaw nodes, each driven by the SAME JawPos but rotated to 0°, 120°, 240° -
->
<SCType ID="Chuck" Caption="Chuck" type="TMachineNode">
    <SCType ID="Jaw1" type="TMachineAxis">
        <ParameterName DefaultValue="JawPos"/><AxisType DefaultValue="Linear"/>
        <Direction><X DefaultValue="1"/><Y DefaultValue="0"/><Z
DefaultValue="0"/></Direction>
        <Matrix><SCType ID="T1" type="TRotateZ" DefaultValue="0"/></Matrix>
    </SCType>
    <SCType ID="Jaw2" type="TMachineAxis">
        <ParameterName DefaultValue="JawPos"/><AxisType DefaultValue="Linear"/>
        <Direction><X DefaultValue="1"/><Y DefaultValue="0"/><Z
DefaultValue="0"/></Direction>
        <Matrix><SCType ID="T1" type="TRotateZ" DefaultValue="120"/></Matrix>
    </SCType>
    <SCType ID="Jaw3" type="TMachineAxis">
        <ParameterName DefaultValue="JawPos"/><AxisType DefaultValue="Linear"/>
        <Direction><X DefaultValue="1"/><Y DefaultValue="0"/><Z
DefaultValue="0"/></Direction>
        <Matrix><SCType ID="T1" type="TRotateZ" DefaultValue="240"/></Matrix>
    </SCType>
</SCType>
```

Each jaw translates along its own local X (its `Direction`), but its `Matrix` turns that local frame to 0° / 120° / 240°, so the three jaws move radially in and out together whenever the single `JawPos` value changes. One parameter, three moving nodes.

Grounding the scheme — `IsFloor`

Mark the node that represents the ground (where gravity points) with the boolean attribute **`IsFloor="True"`** (as on the `Base` node in the example above). It does not affect the kinematics; it tells the graphics window how to build the standard views, so "up" looks natural to the user even when the machine's world Z does not point up — for instance a ceiling-mounted robot whose base Z points downward.

Common scheme errors

When a scheme is loaded the system validates it and reports problems. The messages you are most likely to meet:

- "Axes (X,Y,Z) must define the right coordinate system (repaired)" — the chosen axis travel directions do not form a right-handed triple. The system repairs it, but check the `Direction` vectors and the tool-/workpiece-branch signs (§4).
- **`Tool connector "…" has no correct X/Y/Z axis`** — a tool connector's `XAxisID` / `YAxisID` / `ZAxisID` does not name existing axes.
- "Possibly incorrect ToolAxisID ⋯" — the `ToolAxisID` of a tool or sub-machine does not point at a valid axis (usually it should be the spindle or turret axis).
- "The same machine axis is specified as a X, Y or Z axis more than once in the submachine" — a sub-machine lists one axis in two of its `XAxisID`/`YAxisID`/`ZAxisID` slots.
- "Incorrect redundant robot axis type" — an unsupported value in `RedundantAxis`.

> Advanced scheme topics — how to position nodes with matrices (parent- vs world-relative), build switchable/detachable nodes (turret blocks, attachable heads), author turrets, wire up tool/workpiece connectors, and set up control channels & sub-machines for multi-channel machines — are covered in Advanced machine topics.

Tooling, coolants and simulation

The remaining sections of the descriptor are smaller and mostly self-explanatory:

- **`Coolants`** — the coolant tubes available on the machine, each with the code the postprocessor outputs to switch it on/off.
- **`Simulation`** — the simulation method and resolution, and the collision settings used when verifying a project on this machine.
- **`ToolChange`** — the tool-change position, the timing law(s) (referenced by a connector's `ToolChangeTimeCalcLawIndex`) and the output mode for tool changes.

Machine dimensions

`MachineDimensions` is where the scheme's important driving parameters are gathered so the user sees them all in one compact place — for example the location of a positioner, a cell dimension, or any value that other nodes are parametrized from. A parameter reaches this section in one of two ways: declared directly as its child (for values that belong to base nodes), or attached to it with the `Parent` re-assignment attribute (for values that live on switchable/attachable nodes but should still appear here). See §8.4 for `Parent`.

Discovery

Machines are not registered anywhere — they are discovered by location. A machine file is found simply by being placed in one of the scanned machine folders (`$(SCHEMAS_FOLDER)` / `$(CUSTOM_SCHEMAS_FOLDER)`); the machine library picks it up during its folder scan and loads it on

demand (above). There is no registration record to write — unlike operations, machines need no registrator entry.

Checklist for a new machine

1. Decide the closest base (`AbstractMillMachine`, an abstract robot/lathe, …) or `AbstractMachine` directly.
2. Generate a `GUID`; set `Group`, `Name`, `Comment`, `Image`, `Icon`, postprocessor `SPPFile`.
3. Define the controllable quantities in `MachineStateParameters`.
4. Build the `Schema` tree: nest axes by their real kinematic order and end each branch in the appropriate holder; link each axis to its state parameter via `ParameterName`.
5. Set the relevant `ControlData` switches (arcs, rotary transforms, 5-axis, tool-frame output).
6. Put the file in a scanned machine folder (`$(SCHEMAS_FOLDER)` / `$(CUSTOM_SCHEMAS_FOLDER)`), not under `Supplement` — the library discovers it automatically, with no registration record needed; member `ID`s only need to be unique within this one file (above).

Next: Using XML properties from the CAM API

Machine node reference

This page is the property reference for every node type used in a machine's kinematic scheme, plus the coordinate conventions and units they are all expressed in. It is a look-up companion to Machine descriptors, which walks through how the nodes fit together; come here when you need the full property list of a particular node type.

The node types form a small inheritance tree:

- **`TMachineNode`** — the base of everything in the scheme. Every other node type below inherits its `ID`, `Matrix`, `VisualProperties` and `ImageFile`.
  - **`TMachineAxis`** — a node that *moves* (linear or rotary).
  - **`TToolHolderNode`** / **`TWorkpieceHolderNode`** — the sockets where the tool and the workpiece attach (the *Tool connector* / *Workpiece connector* of the UI).

A separate type, **`TMachineStateParameter`**, is not a scheme node at all — it is the controllable *value* an axis node is driven by. Axis nodes link to it by name.

Coordinate conventions and units

- World coordinate system. Looking at the front of an ordinary mill: Z points up, X to the right, Y away from the viewer. Build the scheme (and model the geometry) in this frame.
- Units. All `Matrix` translations — and all CAD geometry — are in the units of the machine's *Measurements* (see Identity & metadata): millimetres for a metric machine, inches for an imperial one. If the running unit system differs, the system tries to rescale the machine once at load, but that can misbehave for heavily parametrized schemes — so prepare the scheme directly in the units it will be used in.
- **`Direction`** need not be a unit vector. For an ordinary orthogonal axis it is a normalized travel vector. But on machines with non-orthogonal axes — e.g. a slanted Y support on a turn-mill — `Direction`'s magnitude is not 1; its components follow the sines/cosines of the tilt between the axes. The shipped *Mori Seiki* turn-mill (`Machines\LatheMilling\Turret\Mori Seiki\MoriSeiki.xml`) gives its Y axis `Direction = (√3, 1, 0)` — written exactly as `(1.7320508…, 1, 0)`, a 30° tilt with magnitude 2 — for a mathematically precise description.
- Tool-axis convention. The tool's Z runs from the cutting tip, along the tool's rotation axis, to its mounting point in the spindle/holder. For external (OD) turning tools the same holds per ISO 13399: Z from the tip to the holder mount, with the insert's working plane in the ZX plane; X is chosen so that a right-hand tool has most of its cutting geometry on the −X side (a left-hand tool on +X). The CAM system computes everything assuming this placement; only at final CLData output may the optional `AdditionalTransform` (tool-frame output) apply a last offset/rotation — normally zero for machines, and used on robots to flip Z so it points out of the flange (and out of the tool opposite the mount).

TMachineNode

`TMachineNode` is the base for anything in the scheme tree. Every node type inherits these four members:

Property	Default	Meaning
`ID`	`""`	The node's identifier, unique within this machine file.
`Matrix`	identity	Placement of the node relative to its parent (or to the world — see positioning nodes with matrices).
`VisualProperties`	—	How the node's 3D model is drawn (see below).
`ImageFile`	`""`	The node's 3D model, an `.osd` / `.stl` file (see 3D models for machines).

A bare `TMachineNode` is a rigid part of the structure (a bed, a column, a table) — it carries geometry and positions its children, but does not move on its own.

TMachineAxis

A `TMachineAxis` extends `TMachineNode` (so it has `ID`, `Matrix`, `VisualProperties` and `ImageFile` like any node) and adds the axis-specific properties below:

Property	Default	Meaning
`AxisType`	`Linear`	`Linear` or `Rotary`.
`ParameterName`	`""`	ID of the `TMachineStateParameter` that drives this axis.
`Direction`	`(0, 0, 1)`	Travel direction (linear axes only — see the note below).
`Scale`	`1`	Multiplier from the state-parameter value to

Property	Default	Meaning
		physical motion (see below); negative inverts direction.
`RapidFeed`	`10000`	Rapid velocity per minute (mm/min for a linear-mm axis, deg/min for a rotary-deg axis).
`Channel`	`-1`	Control channel (§6); `-1` = common.
`WorkpieceIndex`	`-1`	If ≥ 0, the axis is controlled only for that workpiece.
`DesignTimeAxisValue`	`0`	Axis value used at design time to compute node world matrices.

`Direction` — linear vs. rotary. For a linear axis `Direction` is the direction of travel. For a rotary axis `Direction` is *not used*: rotation is always about the Z axis of the axis's own local coordinate system, so orient the node (via its `Matrix`) to place that Z where the rotation axis should be.

`Scale` is a dimensionless multiplier applied to the axis's state-parameter value — the physical unit is carried by the *value*, not by `Scale`: a length for a linear axis (mm, or inch in imperial) and an angle in radians for a rotary axis. Use `Scale` to remap the value — for example an indexed turret stores a plain station *index* and a `Scale` of `360 / BlocksCount` turns each index step into one station's rotation. A negative `Scale` inverts the axis direction (and so also flips the base CS it contributes to — see §4).

`VisualProperties`

Every node (`TMachineNode` and its descendants — axes, holders, schema nodes) has a `VisualProperties` block that controls how its 3D model is drawn:

Property	Default	Meaning
`Visible`	`True`	Whether the node's model is shown; when off, the node is hidden.
`Color`	`(0.5, 0.5, 0.5)`	Node colour as `R` / `G` / `B`, each `0…1`.

Property	Default	Meaning
`Metallic`	`False`	Render with a metallic appearance.
`IsTransparent`	`False`	Make the node transparent (enables `Transparence`).
`Transparence`	`0.5`	Transparency, `0` = opaque ⋯ `1` = fully transparent.
`DisplayMode`	`Default`	Render mode — `Wire`, `Shade`, `EdgeShade`, or `Default`.
`VisMatrix`	identity	Matrix that places the node's 3D model independently of its kinematic `Matrix` (see §1).
`Use3DModelColors`	`False`	When `True`, draw the node with the colours stored in its 3D model instead of the `Color` above. (An attribute on `VisualProperties`.)

```
<VisualProperties Use3DModelColors="False">  <!-- an attribute on the element itself
-->
    <Visible DefaultValue="True"/>
    <Color><R DefaultValue="0.5"/><G DefaultValue="1"/><B DefaultValue="0.5"/>
</Color>
    <DisplayMode DefaultValue="Shade"/>
</VisualProperties>
```

> Preparing the 3D models themselves (the `.osd` / `.stl` files, where they live, units and alignment) is covered in 3D models for machines.

`TMachineStateParameter`

A `TMachineStateParameter` is a controllable quantity the controller drives — an axis position, a spindle, a jaw travel. Axis nodes link to it through `ParameterName`. It is declared either in the global `MachineStateParameters` section or on a node in the `Schema` (see machine state parameters).

Property	Default	Meaning
`Enabled`	`True`	Whether the parameter is active.
`ID`	`""`	Its identifier; axes and holders link to it by this name (often parametrized, e.g. `[Prefix]…`).
`Address`	`""`	The address letter the postprocessor outputs for it (e.g. `X`, `Z`, `C`, `S`).
`AxisControl`	`Continues`	How it is driven — `Continues` (continuous), `Indexed`, or `Manual` (see notes).
`Group`	`Other`	Classification — `LinearAxis`, `RotaryAxis`, or `Other`.
`Incr`	`0.001`	Smallest increment / resolution.
`Min` / `Max`	`-1E9` / `+1E9`	Travel limits.
`InitialValue`	`0`	The axis value when the machine is reset to its initial state in a project, before any operation runs.
`DesignTimeValue`	`0`	Zero-offset of the axis — the position at which its 3D model was drawn (see notes).
`HasBrake`	`False`	The axis has a brake. When set, an `AXESBRAKE On|Off` command is emitted into CLData around every change of this axis (for the postprocessor to release/apply the brake).
`SupportShortestPathRotation`	*computed*	Periodic rotary axes only — affects rapid-move simulation (see notes).

Property	Default	Meaning
`Priority`	`0`	Order of the parameter in the user's lists/windows only (no effect on output).
`Order`	`[Priority]`	Order in which axes are written into CLData commands.
`ParameterType`	`Geom`	Obsolete — has no effect now (it once grouped the display into geometrical vs. technological).
`ControlWithMap`	`False`	Resolve a redundant axis with the interactive *axis map* (see notes).
`ClampID`	*computed*	Id of a clamping fixture; referenced by part-handoff / takeover operations that clamp and unclamp it.

The ones you set most often are `Address`, `Group`, and `Min`/`Max`/`Incr`.

Notes on selected properties:

- **`AxisControl`** — *Continuous*: several continuous axes may change within one NC block. *Indexed*: cannot move in sync with other axes — it is positioned and locked in its own NC block(s) before the main cut. *Manual*: not program-controlled — the operator sets it by hand, so it may be changed only in Setup objects (not in the operations inside a setup), and a postprocessor may emit it as a comment/instruction for the operator.
- **`ControlWithMap`** — for redundant axes. It enables the interactive *axis map*: a graph of the redundant axis's value (vertical) against path length (horizontal) on which you draw how that axis should move along the toolpath, while watching the whole machine move and seeing node collisions as forbidden zones on the map. It is an interactive way to resolve inverse-kinematics redundancy. Enable it only for *extra* redundant axes that the virtual sub-machine does not already know about: the principal redundant external axes — e.g. a robot's 3 linear rail axes and 2 rotary positioner axes, or the C axis of a 5-axis machine (for singularity avoidance) — appear in the map automatically, without this flag.
- **`SupportShortestPathRotation`** — for periodic rotary axes whose range exceeds 180°. It does not change CLData (the target value, e.g. `A+370`, is still output); it only affects the simulation of rapid moves: when on, a rapid takes the shortest equivalent angle (`A+10` rather than `A+370`); when off, it makes the full turn. Feed moves always perform the full move.

- **`DesignTimeValue`** — use it when an axis's 3D model was drawn at a non-zero position: state that drawn position here, and the axis zero shifts by the opposite amount so the model lines up with axis value 0.

Tool and workpiece connectors

A connector is where a tool or the workpiece attaches to the kinematic chain — the *Tool connector* (`TToolHolderNode`) and *Workpiece connector* (`TWorkpieceHolderNode`) of the UI. Both derive from a common socket and share the spindle and clamping properties below. On machines (not robots) a tool connector additionally carries the properties that tell the kinematics solver how its tool is positioned and selected.

The tables here are the property look-up; the *behaviour* behind these properties — how the base coordinate system is derived, how a connector is made active, the workpiece placement caveat — is explained in §4 Tool connectors.

Tool-specific properties (`TToolHolderNode`)

Property	Default	Meaning
`XAxisID` / `YAxisID` / `ZAxisID`	`""`	Machine axes that position this tool's tip in space; they also define the connector's base coordinate system (§4).
`ToolAxisID`	`""`	The axis to move in order to make this connector active (magazine / turret indexing). May be left empty when an active sub-machine supplies the tool axis.
`ToolAxisValue`	`0`	The value of `ToolAxisID` that brings this station / tool in.
`ToolNumber`	`0`	The magazine position (tool number) the postprocessor outputs. `0` → not used; the NC program outputs the number from the cutting tool's own properties instead.

Property	Default	Meaning
`SupportedToolTypes`	—	The applications a tool in this connector can perform — `MillTool`, `LatheCutter`, `JetCutter`, `Punch`, `Wire`, `Cutter6D`, `Welder`, `AdditiveTool`, `Painter`, `HeatTreatment`, `Gripper`. Used to filter the operation/tool lists offered for the connector.
`Channel`	`-1`	The control channel this connector belongs to (§6).
`ToolChangeTimeCalcLawIndex`	`0`	Index of the tool-change-time law (defined in the machine's *Tool Change* section) that applies to this connector (`0` = the default law).

Spindle and clamping (shared by both connectors)

Property	Default	Meaning
`SpindleParamID`	`""`	ID of the machine state parameter driving the rotary spindle axis that spins the tool or the part. It is the *state-parameter* ID, not the axis node's `ID`.
`HolderType`	`Unknown`	`Unknown`, `LeftLatheSpindle` or `RightLatheSpindle`; marks a lathe spindle socket.
`DefaultClampID`	*computed*	The default clamp id of this socket, referenced by part-handoff / takeover operations; defaults from `HolderType` (left spindle → `1`, right → `2`, otherwise `-1`).

Workpiece connector (`TWorkpieceHolderNode`)

A workpiece connector is where the part is held — on a chuck, spindle or fixture. It carries the shared *spindle and clamping* properties above (most importantly `SpindleParamID` for a turning spindle, and `HolderType` / `DefaultClampID` for clamping), but none of the tool-specific positioning (`XAxisID`/`YAxisID`/`ZAxisID`), activation (`ToolAxisID`/`ToolAxisValue`) or `ToolNumber` / `SupportedToolTypes` fields — those describe how a *tool* is positioned and selected, which does not apply to a part holder.

> Placement caveat. Although `TWorkpieceHolderNode` inherits the node `Matrix`, that matrix is ignored for a workpiece connector — unlike for ordinary nodes, axes and tool holders. A workpiece connector always takes the placement of its parent node. So to offset where the part sits, add an extra parent node and put the placement (its `Matrix`) on *that* node. This is deliberate: the part's position is meant to be defined by the workpiece setup in the project (inside the CAM system), not fixed when the machine is created.

Back to the walkthrough: Machine descriptors · Machines index

Machine Setup parameters reference

> XML paths in this article refer to configuration files in the installed CAM system (`$(SCHEMAS_FOLDER)`), not manual attachments. Available files depend on the version and configuration.

This page explains the parameters shown on the Machine Setup tab of a machine — the *Description*, *Control Parameters* and the related sections — as you see them in the parameter inspector. For each parameter it describes what it does and when to change it, and gives the XML fragment that overrides its default in a machine descriptor.

These parameters are the human-facing side of the machine descriptor's `ControlData` and neighbouring sections (the XML structure is covered in Machine descriptors). Machine builders normally set them with the MachineMaker tool, but understanding them is essential when fine-tuning a machine or writing a postprocessor/interpreter, because they govern how the toolpath is turned into NC code and how the machine is simulated.

> Machine vs. robot. Some parameters and whole groups differ by kinematic type — a 5-axis mill exposes different options from a 6-axis articulated robot. Where it matters, the differences are called out below.

> Where values come from. Each parameter has a machine-level default set here; many of them simply seed the corresponding option that the operator then sees (and can change) inside an operation. The text notes when a Machine Setup value is "the default for the operation property".

How to override a parameter in XML

Each fragment below goes inside your machine type's body, overriding the value inherited from the base machine with the override form from §4.2 — you restate the member (and the section path that contains it) and set a new `DefaultValue`:

```
<SCType ID="MyMachine" Caption="My 5-axis mill" type="AbstractMillMachine">
    <ControlData>
        <LocalCS>
            <AutoLCSRotationLaw DefaultValue="SnapToToolCS"/>
            <!-- SnapToMachineCS, SnapToWorkpieceCS, ExcludeA, ExcludeB, ExcludeC -
->
        </LocalCS>
    </ControlData>
</SCType>
```

The fragments shown for each group below omit the outer `<SCType …>` for brevity — they start at the relevant section (`<ControlData>`, `<ToolChange>`, ⋯). The section nesting matters: a member must be wrapped in the same elements that contain it. Comments after a value list the other accepted enum options. Almost every parameter is declared in `MachineTypes.xml` and `Machines/AbstractMachine.xml`; complete, ready-made machines ship under `$(SCHEMAS_FOLDER)` and are worth studying as examples.

Description

General, mostly informational identity of the machine:

- Name, Group — display name and the machine category it belongs to.
- Developer — who authored the machine/scheme.
- NC System name — the control/CNC family, for information.
- Postprocessor file — the default postprocessor used to generate NC programs for this machine.
- Interpreter file — used to read an NC program back and reconstruct the toolpath inside the machine.
- Measurements — the unit system the machine was created in (mm / inch) and the derived units (linear, cutting speed, feedrate, feed/rev, feed/tooth, revolution). These are view-only on the tab.

```
<!-- machine type body (direct children) -->
<Name      DefaultValue="My 5-axis mill"/>
<Group     DefaultValue="Milling"/>  <!-- Unknown, Lathe, LatheMilling, JetCutter,
WireEDM, Robot -->
<Developer DefaultValue="Acme"/>
<NCSystem  DefaultValue="Heidenhain TNC640"/>
<SPPFile   DefaultValue="$(PROGRAM_PERSONAL)\Postprocessors\MyPost.sppx"/>
<Image     DefaultValue="Images\MyMachine.png"/>
<Icon      DefaultValue="Images\MyMachine_ico.bmp"/>
```

Tooling

For turret machines, this is where the turret block layout is configured.

Control Parameters

The bulk of the tab. These tell the engine and the postprocessor what the control can do and how output should be formed. Everything in this group lives under the `<ControlData>` element.

Arcs

Controls how circular moves are emitted into the NC program.

- Use arcs — master switch. If off, every arc that would appear in the toolpath is replaced by line segments, computed to the tolerance set in the operation.
- Use arcs in XY / YZ / ZX plane — restrict arc output to the planes the control actually supports; arcs in a disabled plane fall back to segments.
- Circles division — whether full circles are broken up: *Do not break*, *Into quadrants* (90°), or *Into halves* (180°).
- Minimal Arc Length / Maximal Arc Radius — guard against degenerate arcs. An arc shorter than the minimum, or with a radius larger than the maximum (almost a straight line), is emitted as segments instead.
- Spatial arcs — arcs that do not lie in an orthogonal XY/YZ/ZX plane, usually defined by three arbitrary points. Common on industrial robots. If off, such arcs become segments; if on, a dedicated multi-point arc command is produced for the postprocessor.
  - Supported — enables spatial-arc output.
  - Minimal distance between start and end points — a 3-point arc cannot be defined when start and end coincide (a full 360° arc). If the start/end distance is below this value, segments are emitted instead.
  - Reorientation mode — how the *tool orientation* is modelled while traversing a spatial arc: *Relative to base CS* (spherical interpolation of the orientation in the arc's base coordinate system) or *Relative to path* (the orientation is interpolated relative to the path tangent).
  - Consider middle point orientation — a 3-point arc carries position *and* orientation at start, middle and end. Off: the middle orientation is ignored and orientation is interpolated start→end (the middle point still shapes the arc in space). On: orientation is interpolated start→middle, then middle→end. Set it to match how the real control behaves.

```
<ControlData>
    <UseArc          DefaultValue="true"/>
    <UseArcInXY      DefaultValue="true"/>
    <UseArcInYZ      DefaultValue="false"/>
    <UseArcInZX      DefaultValue="false"/>
    <CirclesDivision DefaultValue="DoNotBreak"/>  <!-- Quadrants, Halves -->
    <MinArcLength    DefaultValue="0.05"/>
    <MaxArcRadius    DefaultValue="10000"/>
    <SpatialArcs>
        <Supported            DefaultValue="False"/>
        <MinDistance          DefaultValue="0.2"/>
        <ReorientationMode    DefaultValue="rmBaseCS"/>  <!-- rmPath -->
        <ConsiderMiddleOrient DefaultValue="False"/>
    </SpatialArcs>
</ControlData>
```

Singularities

A singularity is a pose where the joint values are mathematically undefined or ambiguous. The contents of this group depend on the current kinematics.

- Machine (rotary table). With a rotary-table scheme, when the tool axis becomes vertical the table rotation is undefined (infinitely many solutions). Such zones should be avoided.
  - Allowed axis deviation (e.g. *Wrist angle*) — how far the tool may be tilted away from the ideal orientation to skirt around a singularity zone instead of driving straight through it.
- Robot. A robot has more singularities (wrist, base/shoulder, elbow) because it has more joints. The per-joint angle values define where each singularity zone begins (the angle from which a joint is considered "in" the zone), and the *J2–J3 free area* / maps describe forbidden regions used to route around them.

```
<ControlData>
    <Singularities>
        <Wrist DefaultValue="0.01"/>   <!-- allowed axis deviation, degrees -->
        <!-- robot only, additional joints: -->
        <Elbow DefaultValue="0"/>
        <Base><Angle DefaultValue="5"/></Base>
    </Singularities>
</ControlData>
```

Flip Table / default machine configuration

When a pose can be reached in more than one way (e.g. tilt the head left or right — same tool/part relation, opposite axis signs), a flip chooses which solution to use. The Machine Setup only stores the default state of each flip; the operation itself exposes the actual flip controls.

- Machines: a single *Flip Table* default.
- Robots: a default state per joint flip — *Flip base (J1)*, *Flip elbow (J3)*, *Flip wrist (J5)*, plus external-axis flips/positions.

```
<ControlData>
    <DefaultFlips>
        <Flip5x DefaultValue="False"/>   <!-- machine: default table/wrist flip
(Boolean) -->
        <!-- robot only: FlipA1, FlipA3, FlipA5, ... -->
    </DefaultFlips>
</ControlData>
```

Origin list parameters

How work coordinate systems are numbered.

- Origin prefix (e.g. `G`), Initial number (e.g. `54`), Increment value (e.g. `1`) — produce `G54`, `G55`, `G56`, ⋯ as new origins are created. Robots use different conventions.
- Create new origin for part group — whether each part group gets its own origin.

```
<ControlData>
    <OriginParams>
        <Prefix                 DefaultValue="G"/>
        <InitialNum             DefaultValue="54"/>
        <IncStep                DefaultValue="1"/>
        <CreateNewOriginForPart DefaultValue="True"/>
    </OriginParams>
</ControlData>
```

Radius Compensation Parameters

- Sharp Corner — the angle threshold (e.g. `135°`) that separates *sharp* from *obtuse* corners when building the cutter-radius-compensated path. At a sharp corner an extra chord is inserted so the tool does not swing too far from the corner; at an obtuse corner the path simply runs to the intersection point.

```
<ControlData>
    <RadiusCompensationParams>
        <SharpCorner DefaultValue="135"/>   <!-- degrees -->
    </RadiusCompensationParams>
</ControlData>
```

Rotary Transformations

Polar and cylindrical interpolation support.

- Polar interpolation is available / Cylindrical interpolation is available — whether the control offers these interpolations at all.
- CNC supports polar interpolation / CNC supports cylindrical interpolation — if on, the control performs the transform itself and the CAM system works as if in plain Cartesian coordinates; if off, the CAM system *simulates* the interpolation and outputs the resulting rotary-axis values for the control to follow.
- Start from C=0 — legacy compatibility for old postprocessors that required the C axis to be reset to zero before enabling interpolation. Leave it off on new machines; the path is now tracked correctly without it.

```
<ControlData>
    <RotaryTrans>
```

```
        <PolarAvailable     DefaultValue="true"/>
        <CNCSupPolar        DefaultValue="true"/>
        <CylindAvailable    DefaultValue="true"/>
        <CNCSupCylind       DefaultValue="true"/>
        <OldPPCompatibility DefaultValue="false"/>  <!-- "Start from C=0"; keep
false -->
    </RotaryTrans>
</ControlData>
```

Tool frame output

How the tool orientation is written in each block of the NC program (mainly for 5-axis / multi-goto output).

- Format (the combo box) — the orientation format: *Normal vector (Nx, Ny, Nz)* (default for machines), the various *Euler angle* sequences (common on robots), *Quaternion*, or *Axis-angle*. Pick what the control expects.
- Additional transformation — an extra rotation of the tool's own coordinate system, mostly for robots. By default the tool Z points from the tip up toward the spindle; some robots are calibrated with Z pointing the other way, or with X along the tool. Three angles let you match the real tool frame.

> Known limitation — read before using on robots. There is currently one global `AdditionalTransform`, and it is applied to two different frames at once: the final *tool* coordinate system and the robot's *6th-axis flange* coordinate system. On real cells those two do not always coincide, and a cell may have several working organs while this setting stays single and global — so one transform cannot correct both independently. This is a known architectural limitation (a future version is expected to split the setting between the machine scheme and the cutting tool). If a robot's tool or flange orientation looks wrong and one `AdditionalTransform` cannot satisfy both, this is why — do not keep re-debugging it.

- Set machine state flags — output the machine frame state (zero / path / robot base) in every block. Off for ordinary machines (not used); typically on for robots, where per-point orientation is essential.
- Swap Tool and Workpiece frames — for robots that hold the part while the tool is stationary. Each block then expresses the part orientation relative to the tool (the inverse of the usual tool-relative-to-part matrix). If the control itself performs this inversion, leave it off; otherwise the CAM system inverts the matrix itself before passing the orientation to the postprocessor.
- Set tool contact surface normal vectors — for 5-axis tool compensation. On finishing passes the contact point and its surface normal are known; outputting that normal lets the control shift the tool along it by the difference between the real and the programmed tool (so a slightly different tool does not gouge). When enabled, a special **`TLCONTACT`** command is written into the CL data in every block that has surface contact, carrying that contact normal; the postprocessor must handle it (see the `TLCONTACT` command in your postprocessor manual).

```
<ControlData>
    <ToolFrameOutput>
        <Format DefaultValue="NormalVector"/>
            <!-- EulerZYX, EulerXYZ, FixedABC, Quaternion, AxisAngleDeg,
AxisAngleRad, ... -->
        <SetMachineStateFlags       DefaultValue="False"/>
        <SwapToolAndWorkpieceFrames DefaultValue="False"/>
        <SetContactNormal           DefaultValue="False"/>
        <AdditionalTransform>           <!-- extra tool-frame rotation (robots) -->
            <A DefaultValue="0"/>
            <B DefaultValue="0"/>
            <C DefaultValue="0"/>
        </AdditionalTransform>
    </ToolFrameOutput>
</ControlData>
```

Redundant axis

Selects the rule for resolving a redundant axis when solving the inverse kinematics (used only by schemes that actually have a redundant axis). Default *None*.

```
<ControlData>
    <RedundantAxis DefaultValue="None"/>
        <!-- KukaLBRiiwa, NITIProgress, NITIProgress3, NonSphericalWrist,
ScaraSimple -->
</ControlData>
```

Robot joint coupling

(Robot schemes.) Flags such as Joint 3 is linked with joint 2 (parallelogram) model mechanically coupled joints — on by default for parallelogram-type robots, off for others. When set, the coupled joint values are output as linked.

```
<ControlData>
    <IsParallelJoints23 DefaultValue="false"/>  <!-- joint 3 linked with joint 2 -->
    <IsConnected46Joints DefaultValue="False"/>
    <IsConnected56Joints DefaultValue="False"/>
</ControlData>
```

Indexed vs. continuous 5-axis machining

Two almost-identical groups, Indexed 5-axis machining and Continuous 5-axis machining, each containing the *same three compensation switches*. Which group an operation uses is chosen by the operation's Tool Center Point Management (TCPM) toggle: TCPM off → the operation uses the *indexed* group; TCPM on → it uses the *continuous* group.

The three compensation switches

They control how the tooltip and the workpiece coordinate system are recomputed as the tool and part move during 5-axis machining:

- 5 Axis tooling point compensation — when on, the tooltip stays rigidly attached to the tool (the control recomputes the tip position for every rotation). When off, the programmed point stays fixed while the tool turns — used in the rare case where a rotary head's kinematics are *not* known to the control (e.g. a manual head); then the CAM system computes the path itself. Normally leave it on.
- 5 Axis workpiece zero point compensation — whether the origin of the workpiece coordinate system moves with the part as axes rotate.
- 5 Axis coordinate system compensation — whether the axis orientation of the workpiece coordinate system rotates with the part.

Defaults differ by group: *indexed* = tooling-point on, workpiece compensation off; *continuous* = all three on (in TCPM the control compensates the whole kinematics, so the path is programmed as if the system were rigidly attached to the part).

```
<ControlData>
    <!-- used when the operation's TCPM is OFF (indexed): -->
    <Indexed5AxisCompensationMode>
        <IsToolTipCompensated        DefaultValue="true"/>
        <IsWorkpieceZeroCompensated  DefaultValue="false"/>
        <IsCoordinateAxesCompensated DefaultValue="false"/>
    </Indexed5AxisCompensationMode>
    <!-- used when the operation's TCPM is ON (continuous): -->
    <TCPM5AxisCompensationMode>
        <IsToolTipCompensated        DefaultValue="true"/>
        <IsWorkpieceZeroCompensated  DefaultValue="true"/>
        <IsCoordinateAxesCompensated DefaultValue="true"/>
    </TCPM5AxisCompensationMode>
</ControlData>
```

Local coordinate system (ORIGIN) — indexed mode

Indexed machining of a tilted feature needs an explicit *rotate the work coordinate system* command in the NC program (e.g. a plane/cycle command). These parameters govern it:

- Local coordinate system availability — *Unavailable* (the control has no such command — the operation then offers no Local CS option at all), *Off* (available but default off), or *Auto*. The chosen value becomes the default for the operation's Local CS property.
- Auto computes the local CS so its Z is along the tool axis while keeping the origin where the work zero (e.g. `G54`) was set.
- Auto LCS rotation law — Z along the tool leaves the rotation *about* Z undefined; this law fixes it: *Snap to Tool CS*, *Snap to Machine CS*, *Snap to Workpiece CS*, or *Exclude A/B/C* (force one spatial angle to zero so the result is just two rotations). The right choice usually depends on where the rotary axes sit in the kinematics.
- Local CS positioning mode — what the machine axes do when the CS-rotation command is output:
  - *Stay* — the CS rotates but no axis moves; the tooltip stays put, so its coordinates relative to the new CS change. You must add the axis-turn commands yourself.
  - *Turn* — additionally turns the rotary axis so the tool re-aligns with the new CS Z (e.g. on some controls a CS-rotate command is immediately followed by an "align rotaries" command). Most common — the rotate command alone is enough.
  - *Move* — like *Turn*, and also moves the linear axes so the tooltip keeps the same coordinates in the rotated CS. Usually undesirable.
- Euler angles type (rotation sequence) — the angle convention used to output the CS rotation (e.g. `XYZ`, `ZXZ`). Note that the CL data carries *both* the spatial (Euler) angles and the physical axis values; the postprocessor decides which to emit. The two can differ greatly on non-orthogonal kinematics (e.g. a 45°-skewed table), so confirm with the machine integrator which the control expects.
  - Rotations around movable axes — primed vs. unprimed convention (rotate about the already-rotated axes vs. the fixed axes).
  - Angles in degrees — degrees vs. radians.
- Move auto LCS with table — whether the origin of the auto CS rotates with the part (on) or stays at its original calibrated location (off).
- Rotatable Workpiece CS — whether the workpiece CS rotates with the part at setup/calibration time (when you define `G54` with an axis already rotated). Off for machines (the workpiece CS is always calibrated in a fixed orientation), on for robots (it is rigidly tied to the part).

```
<ControlData>
    <LocalCS>
        <DefaultState           DefaultValue="Auto"/>   <!-- Unavailable, Off, Auto
-->
        <PositioningMode        DefaultValue="Stay"/>    <!-- Turn, Move -->
        <IsSpatial              DefaultValue="True"/>
        <AutoLCSRotationLaw     DefaultValue="SnapToMachineCS"/>
            <!-- SnapToToolCS, SnapToWorkpieceCS, ExcludeA, ExcludeB, ExcludeC -->
        <TurnAutoLCSRotationLaw DefaultValue="ZAlongTurn_XAlongTool"/>
            <!-- ZAlongTurn_MinusXAlongTool, ZAlongTool_XbySnapRule -->
        <MoveAutoLCSWithTable   DefaultValue="true"/>
        <RotatableWCS           DefaultValue="false"/>   <!-- robots: true -->
```

```
        <EulerAnglesType>
            <RotationsSequence          DefaultValue="XYZ"/>  <!-- ZXZ, ZYX, XZY,
... -->
            <RotationsAroundMovableAxes DefaultValue="False"/>
            <AnglesInDegrees            DefaultValue="True"/>
        </EulerAnglesType>
    </LocalCS>
</ControlData>
```

Continuous 5-axis machining (TCPM)

- TCPM mode is available — whether the control offers continuous 5-axis / tool centre point management (an often separately-licensed option). If unavailable, the operation shows no TCPM toggle.
- TCPM mode default state — the default for the operation's TCPM toggle.
- Disable TCPM when rotate back — for rotary axes with a limited travel range that must be "rewound". Rewinding with TCPM active makes every axis move to hold the tooltip fixed — a large, risky motion. With this on, the system drops out of TCPM, turns just the one rotary axis back, then re-enables TCPM and continues.

These three live under `RotaryTrans` (the *Continuous 5-axis machining* group is only a UI grouping):

```
<ControlData>
    <RotaryTrans>
        <TCPMAvailable         DefaultValue="False"/>
        <TCPMDefault           DefaultValue="False"/>
        <DisableTCPMifOverturn DefaultValue="True"/>  <!-- "Disable TCPM when rotate
back" -->
    </RotaryTrans>
</ControlData>
```

Tool Change

- Go to tool change position — *Only if tool change is needed* (retract to the change position only when consecutive operations use different tools), *Always at the end of operation*, or *Never* (a legacy mode that simply swaps the tool with no retract, emulating the old "tool floating in space" behaviour from before machine simulation existed).
- Output mode — how the change-position coordinates are written:
  - *Machine coordinates (ISO G53)* — physical-axis moves (e.g. `G53 Z0`); safe, small numbers, independent of the work zero.
  - *Reference point (ISO G28)* — retract via the machine reference point; common on turn-mill / lathe centres; avoids stating an explicit change-position in the work CS.

    - *Tool tip coordinates* — the actual tip coordinates relative to the work CS; produces large numbers and requires the machine to be calibrated, or it may hit the limit switches.
- Tool change position — the position itself.
- Heavy axes — axes the collision-avoidance approach/retract planner should avoid moving where possible (manual/indexed/awkward axes such as a column). Lists machine node IDs.
- Tool change time calculation — how change time is estimated: *Undefined* (a built-in constant, for compatibility) or an explicit law stating the duration; several laws can be defined.

```
<!-- machine type body -->
<ToolChange>
    <Using      DefaultValue="Auto"/>     <!-- Anyway, DoNotUse -->
    <OutputMode DefaultValue="RefPoint"/> <!-- Tooltip, Machine -->
</ToolChange>
<ToolChangeMachineState DefaultValue=""/> <!-- tool change position -->
<Leads>
    <MotionPlannerOptions>
        <HeavyAxes DefaultValue=""/>       <!-- space-separated machine node IDs -->
    </MotionPlannerOptions>
</Leads>
```

Coolants

Which coolant channels the machine has — *Flood*, *Mist*, *Tool* (through-tool), and additional numbered tubes — each with a *Supported* flag and an optional switch time. Enabled channels appear on the operation's output tab.

```
<Coolants>
    <Coolant5>                          <!-- Coolant1..Coolant20 -->
        <Supported  DefaultValue="True"/>
        <SwitchTime DefaultValue="1"/>
    </Coolant5>
</Coolants>
```

Simulation

Parameters for the machining/stock simulation:

- Simulation method — *Voxel 5d* (default; fast, less precise — a voxel is a 3D pixel), *Voxel 3d*, or *Solid* (surface model; more precise and independent of voxel size, but slower on long paths and complex shapes). Voxel modes are a poor fit for very thin tools (e.g. EDM wire) where the tool size approaches the voxel size.
- Model resolution — the voxel count the stock is divided into (high / standard / low).

- Gouge detection tolerance — how precisely gouges into the part are detected.
- Revolution bodies simulation — when a chuck spins fast, its jaws become a body of revolution; on, that body is computed from the 3D jaw model for display and collision (can slow simulation); off, the jaws are shown/collided as-is.
- Check inappropriate tool and spindle direction — warns when the tool/spindle rotation is wrong (e.g. a right-hand tool with the spindle reversed), which would ruin the surface or crash the tool.
- Collisions to ignore — node pairs whose collisions should not be checked. Adjacent nodes are ignored automatically; list here only non-adjacent pairs that can never collide, to speed up checking.
- Show active part only — for multi-channel / Swiss-type machines that machine two part instances at once; hides the second part instance when it is in the way.

```
<Simulation>
    <Simulation_Method       DefaultValue="smVoxel5d"/>  <!-- smSolid, smVoxel -->
    <Simulation_Resolution   DefaultValue="srStandard"/> <!-- srLow, srHigh -->
    <GougeDetectionTolerance DefaultValue="0.05"/>
    <RevBodySim              DefaultValue="True"/>
    <CheckSpindleDirection   DefaultValue="true"/>
    <ShowActivePartOnly      DefaultValue="False"/>
</Simulation>
```

System settings → Main

- Rotation angles type (rotation sequence) — the default Euler-angle convention used wherever angles are entered (e.g. when creating a coordinate system). It is the default only; each case can still override it.

```
<SystemSettings>
    <Main>
        <RotationAnglesType>
            <RotationsSequence DefaultValue="XYZ"/>  <!-- ZXZ, ZYX, ... -->
        </RotationAnglesType>
    </Main>
</SystemSettings>
```

Machine state parameters, Schema, Machine dimensions

The remainder of the tab is the machine's own kinematics:

- Machine state parameters — the controllable quantities (Axis X/Y/Z/B/C positions, ⋯); see Machine state parameters.
- Schema — the kinematic node tree; see The kinematic scheme.

- Machine dimensions — cell-calibration values established when the cell is set up: e.g. the position of a rotary fixture/table relative to the cell's world coordinate system, or a tool's mounted position. Especially important for robot cells.

Back to Machine descriptors | guide index

Advanced machine topics

> XML paths in this article refer to configuration files in the installed CAM system (`$(SCHEMAS_FOLDER)`), not manual attachments. Available files depend on the version and configuration.

This page covers the parts of machine authoring that go beyond the basic kinematic scheme of Machine descriptors: how to position nodes with matrices, how to build switchable / detachable nodes (turret blocks, attachable heads, …), a full turret walk-through, and control channels & sub-machines for multi-channel machines.

Most machines are produced with MachineMaker (see Machine descriptors); the techniques here are for the special cases that are still authored, or fine-tuned, by hand. The building-block types are in `Machines/MachineTypes.xml`, the turret/chuck templates in `Machines/TurretTypes.xml`, and a multi-channel example template in `Machines/SwissTemplate.xml`. Complete machines ship under `$(SCHEMAS_FOLDER)` and are the best reference.

---

1 Positioning nodes with matrices

Every node in the `Schema` (a `TMachineNode` and its descendants) carries a **`Matrix`** that places it relative to its parent node. The whole kinematic chain is the product of these matrices down the tree. There are two ways to fill a `Matrix`, plus a separate matrix used only for the 3D model.

The matrix type (`TMatrix`) has a **`BaseCS`** selector — `Parent` (default) or `World` — and is built either from a *series of transformation steps* or from an *explicit basis* (`OrtX`/`OrtY`/`OrtZ` + `Move`).

Way 1 — relative to the parent CS (a series of moves and turns)

The usual way. Leave the base matrix at identity and add a series of translation and rotation steps, applied in declaration order. The step types are `TTranslateX` / `TTranslateY` / `TTranslateZ` and `TRotateX` / `TRotateY` / `TRotateZ` (degrees). Each step is a member you declare inside `Matrix`:

```
<SCType ID="MyNode" type="TMachineNode">
    <Matrix>
        <SCType ID="T1" type="TTranslateZ" DefaultValue="100"/>  <!-- then... -->
        <SCType ID="T2" type="TRotateX"    DefaultValue="90"/>
    </Matrix>
</SCType>
```

This is what the turret templates use, with the offsets driven by expressions (see §3):

```
<!-- TurretTypes.xml: a turret block, positioned on the head -->
<Matrix>
    <SCType ID="T1" type="TTranslateX" DefaultValue="-[TurretParameters.Radius]"/>
    <SCType ID="T2" type="TRotateZ"
            DefaultValue="-
[TurretParameters.RotationDirection]*360/[TurretParameters.BlocksCount]*
([Parameters.BlockNumber]-1)"/>
</Matrix>
```

`BaseCS` stays `Parent` (the default), so the steps are interpreted in the parent node's coordinate system. Order matters — `T1` is applied, then `T2`.

Way 2 — relative to the world CS (spatial angles)

When a node is easier to describe in the cell's world coordinate system, set `BaseCS` to `World` and give a single spatial transformation (`TComplexTransformation3d`): a translation plus three angles in a chosen convention (Euler / fixed-axis / quaternion / axis-angle).

```
<Matrix>
    <BaseCS DefaultValue="World"/>
    <SCType ID="T1" type="TComplexTransformation3d">
        <Translation><X DefaultValue="0"/><Y DefaultValue="0"/><Z
DefaultValue="300"/></Translation>
        <Rotation>
            <Convention DefaultValue="EulerZYX"/>  <!-- FixedXYZ, Quaternion,
AxisAngleDeg, ... -->
            <R1 DefaultValue="0"/>
            <R2 DefaultValue="45"/>
            <R3 DefaultValue="0"/>
            <R4 DefaultValue="0"/>   <!-- 4th value only for quaternion / axis-angle
-->
        </Rotation>
    </SCType>
</Matrix>
```

> You can also set the basis explicitly with `OrtX`/`OrtY`/`OrtZ` (the three axis vectors) and `Move` (the origin), but the two forms above are what you will use in practice.

The 3D-model matrix (`VisualProperties.VisMatrix`)

The node's matrix above defines its kinematic frame. The 3D model attached to the node (`ImageFile`, an `.osd` or `.stl`) is positioned by a *separate* matrix, `VisMatrix`, inside `VisualProperties`. Use it to align an imported model with the node frame without disturbing the kinematics:

```
<SCType ID="MyNode" type="TMachineNode">
    <ImageFile DefaultValue="Images\head.osd"/>
    <VisualProperties>
        <VisMatrix>
            <SCType ID="T1" type="TRotateZ" DefaultValue="180"/>
        </VisMatrix>
    </VisualProperties>
</SCType>
```

(Machine visualization is handled separately from kinematics; `VisMatrix` only affects how the model is drawn.)

---

2 Switchable / detachable nodes

Real machines have interchangeable parts — turret blocks, attachable milling heads, sub-spindle adapters, chuck variants. Within the machine-scheme XML there are two mechanisms to model "a node that may or may not be present", with different trade-offs.

> Scope. This section covers only the two *XML-scheme* mechanisms. Other kinds of detachable content are handled at the project level by separate, non-XML mechanisms and are unrelated to the scheme described here — for example fixtures (workpiece tooling, stored in their own binary container together with 3D models) and tool assemblies (built in their own data structures and plugged into the machine's magazine connectors). Don't reach for `TCaseNode` or node hiding to model those.

Mechanism A — `TCaseNode` (one branch loaded at parse time)

A node derived from `TCaseNode` holds several alternative child branches but exposes an **`ActiveNode`** naming the one that is "in". When the machine descriptor is parsed, only the active branch is loaded into the node tree; the others are skipped entirely.

```
<SCType ID="HeadBlockSelector" type="TCaseNode">
    <ActiveNode DefaultValue="EmptyBlock"/>   <!-- which child branch to load -->
    <Parameters>
        <SCType ID="BlockNumber" type="Integer" DefaultValue="1"/>
    </Parameters>
    <SCType ID="EmptyBlock"  Caption="Empty"  type="TAbstractTurretHeadBlock"/>
    <SCType ID="DrillBlock"  Caption="Drill"  type="TTurretDrillToolHolder"/>
    <!-- ...more alternatives... -->
</SCType>
```

- Pros: inactive branches consume no memory — essential when a machine has many optional blocks (a turret with dozens of stations, a tool magazine, …).
- Limit: the active branch is chosen once, at load/parse time. A project owns a single machine instance, so you cannot switch the active block *during* the project — only initialize it at the start. This is the classic mechanism for turrets (see §3).

`TCaseNode` carries an `ActiveNode` (the active child's ID) and an optional `Parameters` block; both are skipped by the loader when it descends into the chosen branch.

Mechanism B — hiding inactive nodes (switchable at runtime)

Here all nodes are loaded, but a node is shown/active only when it belongs to the currently selected tool or workpiece connector. Because nothing is unloaded, the active block can change mid-project. Three attributes drive it:

Attribute	Type	A node carrying it is active only when…
`VisToolHolderID`	string	its value equals the ID of the active tool connector
`VisWorkpieceHolderID`	string	its value equals the ID of the active workpiece connector
`HideIfInactive`	bool	the node lies on the branch of the active tool or workpiece (a simpler switch than spelling out IDs)

```
<!-- this adapter shows only when tool connector "Tool3" is active -->
<SCType ID="Adapter3" type="TMachineNode" VisToolHolderID="Tool3">
    ...
</SCType>

<!-- this branch shows only while it carries the active tool/workpiece -->
<SCType ID="OptionalHead" type="TMachineNode" HideIfInactive="True">
    ...
</SCType>
```

`HideIfInactive` is the easy equivalent of `VisToolHolderID` / `VisWorkpieceHolderID` when you do not want to hard-code connector IDs (which may change): instead of naming IDs, you just flag the node, and it is active whenever it is an ancestor of the current tool or workpiece connector.

Which to use

	`TCaseNode` (A)	Hiding (B)
Loaded into memory	active branch only	all branches
When the choice is made	once, at parse/load	any time, mid-project
Memory for many options	low	higher
Typical use	turret blocks, large optional sets	attachable heads/adapters switched during work

---

3 Turrets

A turret is the textbook use of §1 (parametric matrices), §2 Mechanism A (`TCaseNode` block selection) and parametrization (to keep node IDs unique across identical blocks). The shipped templates are in `Machines/TurretTypes.xml`; some older machines build a turret without the template but on the same principle (the template was added later).

The pieces

1. The turret head — a rotary, indexed axis. It holds a `TurretParameters` block (prefix, station count, radius, the IDs of the linear axes the turret rides on, rotation direction) and an indexed state parameter. `Scale` maps one index step to `360 / BlocksCount` degrees:

```
<SCType ID="TAbstractTurretHead" Caption="Turret head" type="TMachineAxis">
    <ID            DefaultValue="[TurretParameters.Prefix]TurretHead"/>
    <ParameterName DefaultValue="[TurretParameters.Prefix]TurretAxisPos"/>
    <AxisType      DefaultValue="Rotary"/>
    <Scale         DefaultValue="360/[TurretParameters.BlocksCount]"/>
    <SCType ID="TurretParameters" Caption="Turret parameters" type="ComplexType">
        <SCType ID="Prefix"           type="String"  DefaultValue=""/>
        <SCType ID="BlocksCount"      type="Integer" DefaultValue="12"/>
        <SCType ID="Radius"           type="Double"  DefaultValue="120"/>
        <SCType ID="XAxisID"          type="String"  DefaultValue="[Prefix]AxisX"/>
        <SCType ID="ZAxisID"          type="String"  DefaultValue="[Prefix]AxisZ"/>
        <SCType ID="RotationDirection" type="Integer" DefaultValue="1"/>
    </SCType>
    <SCType ID="TurretAxisPos" type="TMachineStateParameter">
        <ID          DefaultValue="[TurretParameters.Prefix]TurretAxisPos"/>
        <Address     DefaultValue="[TurretParameters.Prefix]T"/>
        <AxisControl DefaultValue="Indexed"/>
        <Group       DefaultValue="RotaryAxis"/>
```

```
    </SCType>
</SCType>
```

2. A block — placed on the head by a parametric matrix. Each station is rotated to its angular position and pushed out by the head radius (§1, Way 1):

```
<SCType ID="TAbstractTurretHeadBlock" Caption="Turret head
block" type="TMachineNode">
    <Matrix>
        <SCType ID="T1" type="TTranslateX" DefaultValue="-
[TurretParameters.Radius]"/>
        <SCType ID="T2" type="TRotateZ"
                DefaultValue="-
[TurretParameters.RotationDirection]*360/[TurretParameters.BlocksCount]*
([Parameters.BlockNumber]-1)"/>
    </Matrix>
</SCType>
```

3. A tool holder on the block — with parametric, unique IDs. Every block must produce unique node IDs and tool numbers; parametrization off `[Parameters.BlockNumber]` and `[TurretParameters.Prefix]` guarantees that:

```
<SCType ID="TTurretToolHolder" Caption="Tool holder" type="TToolHolderNode">
    <ID            DefaultValue="
[TurretParameters.Prefix]Tool[Parameters.BlockNumber]"/>
    <ToolAxisID    DefaultValue="[TurretParameters.Prefix]TurretHead"/>
    <ToolAxisValue DefaultValue="[TurretParameters.RotationDirection]*
([Parameters.BlockNumber]-1)"/>
    <ToolNumber    DefaultValue="[Parameters.BlockNumber]"/>
    <XAxisID       DefaultValue="[TurretParameters.XAxisID]"/>
    <ZAxisID       DefaultValue="[TurretParameters.ZAxisID]"/>
</SCType>
```

4. The block selector — a **`TCaseNode`**. Each station is a selector whose `Parameters.BlockNumber` distinguishes it and whose `ActiveNode` picks the fitted block type (empty, lathe cutter, driven drill, …). Only the active block per station is loaded:

```
<SCType ID="TAbstractTurretHeadBlockSelector" type="TCaseNode">
    <ActiveNode DefaultValue="EmptyBlock"/>
    <Parameters>
        <SCType ID="BlockNumber" type="Integer" DefaultValue="1"/>
    </Parameters>
```

```
    <SCType ID="EmptyBlock" Caption="Empty" type="TAbstractTurretHeadBlock"/>
    <!-- concrete machine adds: lathe cutter holder, drill holder, ... as
alternatives -->
</SCType>
```

Why parametrization is essential here

A turret has many identical stations. If their nodes had fixed IDs they would collide. By driving every `ID`, `ToolNumber`, `Address` and matrix offset from `[Parameters.BlockNumber]` (per station) and `[TurretParameters.Prefix]` (per turret, so two turrets on one machine don't clash), one template instantiates cleanly any number of times. This is the same `[...]` expression language as everywhere else (see the expression language); machine descriptors lean on it heavily.

> The chuck/jaw templates in the same file follow the identical pattern — a parametric `ChuckParameters` block and per-jaw matrices driven by `[JawIndex]`.

Putting it together — assembling a turret from the template

Now combine the template pieces into a complete, reusable turret. Everything derives from the `TurretTypes.xml` types, so the head's step angle, the block matrices and all IDs are produced automatically from the parameters.

1. Define the machine's blocks — each block derives from `TAbstractTurretHeadBlock` (inheriting its radial/angular positioning matrix) and holds the appropriate parametric holder:

```
<SCType ID="LatheCutterBlock" Caption="Lathe cutter holder"
type="TAbstractTurretHeadBlock">
    <SCType ID="ToolHolder" type="TTurretLatheCutterHolder"/>
</SCType>
<SCType ID="DrivenDrillBlock" Caption="Driven drill holder"
type="TAbstractTurretHeadBlock">
    <SCType ID="ToolHolder" type="TTurretDrillToolHolder"/>
</SCType>
```

> The holders above are the template's turret-ready holders, also declared in `TurretTypes.xml`: **`TTurretToolHolder`** (from the generic connector `TToolHolderNode`), **`TTurretLatheCutterHolder`** (from `TLatheCutterHolder`) and **`TTurretDrillToolHolder`** (from `TMillToolHolder`). Each one only *adds the parametric turret tool IDs* — `ID = [TurretParameters.Prefix]Tool[Parameters.BlockNumber]`, plus `ToolAxisID`, `ToolNumber` and the linked `XAxisID`/`YAxisID`/`ZAxisID` — on top of its connector type. That is what lets a block simply drop one in and inherit unique, prefixed names automatically.

2. Define the machine's block selector — derive it from the template selector `TAbstractTurretHeadBlockSelector` (a `TCaseNode` that already provides `ActiveNode`, `Parameters.BlockNumber` and an inherited `EmptyBlock` alternative) and add the blocks this machine supports:

```
<SCType ID="TMyBlockSelector" Caption="Turret block selector"
type="TAbstractTurretHeadBlockSelector">
    <SCType ID="LatheCutter" Caption="Lathe cutter" type="LatheCutterBlock"/>
    <SCType ID="DrivenDrill" Caption="Driven drill" type="DrivenDrillBlock"/>
    <!-- "EmptyBlock" is inherited from TAbstractTurretHeadBlockSelector -->
</SCType>
```

3. Build the turret as a type — derive it from `TAbstractTurretHead` (which gives the rotary, indexed axis and computes `Scale = 360 / BlocksCount` for you), set its `TurretParameters`, and hang the 12 station selectors. Each station's `BlockNumber` makes its IDs unique; its `ActiveNode` says which block is fitted there:

```
<SCType ID="TMyTurret" Caption="Turret" type="TAbstractTurretHead">
    <TurretParameters>
        <BlocksCount DefaultValue="12"/>   <!-- head Scale becomes 360/12 = 30° -->
        <Radius      DefaultValue="120"/>
    </TurretParameters>

    <SCType ID="T1"  Caption="Position 1"  type="TMyBlockSelector">
        <ActiveNode DefaultValue="LatheCutter"/>   <!-- station 1: a lathe cutter -
->
        <Parameters><BlockNumber DefaultValue="1"/></Parameters>
    </SCType>
    <SCType ID="T2"  Caption="Position 2"  type="TMyBlockSelector">
        <ActiveNode DefaultValue="DrivenDrill"/>   <!-- station 2: a driven drill -
->
        <Parameters><BlockNumber DefaultValue="2"/></Parameters>
    </SCType>
    <SCType ID="T3"  Caption="Position 3"  type="TMyBlockSelector">
        <ActiveNode DefaultValue="EmptyBlock"/>    <!-- station 3: empty (inherited
block) -->
        <Parameters><BlockNumber DefaultValue="3"/></Parameters>
    </SCType>

    <!-- ...stations T4 … T11, each a TMyBlockSelector with its own
         ActiveNode and BlockNumber = 4 … 11 ... -->

    <SCType ID="T12" Caption="Position 12" type="TMyBlockSelector">
        <ActiveNode DefaultValue="LatheCutter"/>
```

```
        <Parameters><BlockNumber DefaultValue="12"/></Parameters>
    </SCType>
</SCType>
```

That is the whole idea end-to-end:

- One selector type enumerates all possible blocks; twelve instances of it (`T1`⋯`T12`) are the physical stations on the head.
- Each station's **`ActiveNode`** picks the *one* block loaded there — the other alternatives cost no memory (§2 A).
- Each station's **`BlockNumber`**, together with the head's `TurretParameters`, parametrizes the block's matrix angle and its tool `ID`/`ToolNumber`, so the twelve identical stations produce twelve distinct, non-colliding tools.
- The configuration is fixed when the machine is parsed; to re-equip a station the user edits the machine, which is re-loaded.

Placing the turret(s) in the machine — why `Prefix` matters

`TMyTurret` is now a reusable type. Place it in the machine's `Schema` at the end of the slide that carries it, so it moves with those axes. With a single turret you can leave its `Prefix` empty:

```
<Schema>
    <SCType ID="AxisX" type="TToolAxisX">
        <SCType ID="AxisZ" type="TToolAxisZ">
            <SCType ID="Turret" Caption="Turret" type="TMyTurret"/>
        </SCType>
    </SCType>
</Schema>
```

But a lathe/turn-mill may carry two turrets (upper/lower), and a machine is loaded into a single namespace, so every node ID must be unique *within the machine* (how a machine is loaded). Two copies of `TMyTurret` would each emit `TurretHead`, `Tool1`⋯`Tool12`, `TurretAxisPos`, ⋯ — a collision. This is exactly why `TAbstractTurretHead` parametrizes every generated name with a **`Prefix`**: give each instance a different prefix and all its IDs become unique:

```
<Schema>
    <!-- lower turret on the lower X/Z slide -->
    <SCType ID="LowerX" type="TMachineAxis"> <ParameterName
DefaultValue="LowerAxisXPos"/>
      <SCType ID="LowerZ" type="TMachineAxis"> <ParameterName
DefaultValue="LowerAxisZPos"/>
        <SCType ID="LowerTurret" Caption="Lower turret" type="TMyTurret">
            <TurretParameters>
```

```
                <Prefix  DefaultValue="Lower"/>   <!-- -> LowerTurretHead,
LowerTool1, LowerTurretAxisPos -->
                <XAxisID DefaultValue="LowerX"/>
                <ZAxisID DefaultValue="LowerZ"/>
            </TurretParameters>
        </SCType>
      </SCType>
    </SCType>

    <!-- upper turret on its own slide -->
    <SCType ID="UpperX" type="TMachineAxis"> <ParameterName
DefaultValue="UpperAxisXPos"/>
      <SCType ID="UpperZ" type="TMachineAxis"> <ParameterName
DefaultValue="UpperAxisZPos"/>
        <SCType ID="UpperTurret" Caption="Upper turret" type="TMyTurret">
            <TurretParameters>
                <Prefix  DefaultValue="Upper"/>   <!-- -> UpperTurretHead,
UpperTool1, UpperTurretAxisPos -->
                <XAxisID DefaultValue="UpperX"/>
                <ZAxisID DefaultValue="UpperZ"/>
            </TurretParameters>
        </SCType>
      </SCType>
    </SCType>
</Schema>
```

Each instance reuses the same `TMyTurret` definition (all 12 stations and their blocks) but, thanks to its distinct `Prefix`, produces its own non-colliding IDs — `LowerTurretHead` / `LowerTool1…` vs `UpperTurretHead` / `UpperTool1…` — and references its own slide axes through the parametrized `XAxisID` / `ZAxisID`. (The single-turret case can leave `Prefix` empty.) On multi-channel machines the two turrets typically also sit on different `Channel`s — see §6.

---

4 Tool connectors — positioning axes and activation

A connector is where a tool or the workpiece attaches to the kinematic chain — the *Tool connector* (`TToolHolderNode`) and *Workpiece connector* (`TWorkpieceHolderNode`) of the UI (see building blocks). Both derive from a common socket and so share the spindle/clamping properties in Spindle and clamping. On machines (not robots) a tool connector additionally carries the properties that tell the kinematics solver how its tool moves and how it is selected.

> The connectors' complete property tables are gathered in the node reference → Tool and workpiece connectors; this section explains the *behaviour* behind those properties.

Positioning axes and the base CS — `XAxisID` / `YAxisID` / `ZAxisID`

`XAxisID`, `YAxisID`, `ZAxisID` name the machine axes that position this tool's tip in space. They give the solver a usable chain *even without an explicit sub-machine* (§5) — in effect each connector already describes a small "virtual sub-machine". They also define that chain's base coordinate system.

Explicit sub-machines later added `OriginG54BaseNode` / `OriginG54` to set this base CS directly. When those are not given, the base CS is derived:

- its origin coincides with the machine bed's world CS origin;
- its axis directions come from the travel directions of `XAxisID` / `YAxisID` / `ZAxisID`.

The result must be a right-handed triple, so set the axes' travel directions carefully. The branch an axis sits in matters:

- an axis in the tool-moving branch contributes its travel direction directly (positive);
- an axis in the workpiece-moving branch contributes the opposite direction (negative) — moving the part one way is equivalent to moving the tool the other way.

A negative axis `Scale` (node reference) flips a travel direction too, so it also affects this base CS.

Making a connector active — `ToolAxisID` / `ToolAxisValue`

This is the core mechanism behind tool magazines in general and turrets in particular. A connector states: *to make the tool in this connector active, move the axis `ToolAxisID` to the value `ToolAxisValue`.* It works for rotary magazines (a turret indexes to a station angle) and linear ones (a magazine slides to a position) alike.

```
<SCType ID="Tool5" type="TMillToolHolder">
    <ToolAxisID    DefaultValue="TurretHead"/>   <!-- axis that indexes the magazine
-->
    <ToolAxisValue DefaultValue="4"/>            <!-- value that brings this station
in -->
    <ToolNumber    DefaultValue="5"/>
</SCType>
```

In the turret template these are parametric — `ToolAxisID = [TurretParameters.Prefix]TurretHead` and `ToolAxisValue = [TurretParameters.RotationDirection]*([Parameters.BlockNumber]-1)` — so every station automatically points at the turret axis and its own index.

> **`ToolAxisID`** can be omitted. When an active sub-machine (§5) defines a tool axis, the system uses *that*; it falls back to the connector's own `ToolAxisID` only when the sub-machine has none. So a

> connector may leave `ToolAxisID` empty and let the active sub-machine supply it.

Tool number and supported applications

- **`ToolNumber`** — the magazine position of this connector, commonly known as the tool number; it is what the postprocessor outputs in the NC program. In the turret template it is parametrized to the station index. If it is left at `0` (not set), the connector number is not used and the NC program instead outputs the number taken from the cutting tool's own properties.
- **`SupportedToolTypes`** (*Supported applications*) — the kinds of machining a tool fitted in this connector can perform: `MillTool` (milling), `LatheCutter` (lathe cutting), `JetCutter` (jet cutting), `Punch`, `Wire` (wire EDM), `Cutter6D` (6D cutting), `Welder` (welding), `AdditiveTool` (additive), `Painter` (spray painting), `HeatTreatment`, `Gripper` (gripping). The system uses this set to filter the lists of operations and tools offered for the connector down to those its applications support.
- **`Channel`** — the control channel this connector belongs to (§6).
- **`ToolChangeTimeCalcLawIndex`** — index of the tool-change-time law (defined in the machine's *Tool Change* section) that applies to this connector (`0` = the default law).

Spindle and clamping (shared by both connectors)

Because tool and workpiece connectors derive from the same socket, these apply to both:

- **`SpindleParamID`** — the ID of the machine state parameter (machine state parameters) that drives the rotary spindle axis spinning the tool or the part. It is the *state-parameter* ID, not the axis node's `ID`.
- **`HolderType`** — `Unknown`, `LeftLatheSpindle` or `RightLatheSpindle`; marks a lathe spindle socket.
- **`DefaultClampID`** — the default clamp id of this socket, referenced by part-handoff / takeover operations; it defaults from `HolderType` (left spindle → `1`, right → `2`, otherwise `-1`).

Workpiece connectors

A workpiece connector (`TWorkpieceHolderNode`) holds the part (on a chuck, spindle or fixture) and carries only the shared spindle/clamping properties above — none of the tool-specific positioning, activation or `ToolNumber` / `SupportedToolTypes` fields, which describe how a *tool* is positioned and selected and do not apply to a part holder. Its full property list and the placement caveat (the node `Matrix` is ignored — a workpiece connector always takes its parent node's placement) are in the node reference → Workpiece connector.

---

5 Sub-machines (alternating work)

> Channels vs. sub-machines — the key difference. *Channels* (§6) describe parts of a machine that work simultaneously. *Sub-machines* describe alternative kinematic chains used one at a time

> (alternately) — they decide *which* axes drive a tool. They occur even on single-channel machines, so they are a separate concept.

The inverse-kinematics (IK) solver works with a fixed-shape chain: three linear axes plus two rotary axes (`X Y Z` + `RotaryAxis1` + `RotaryAxis2`). When a machine's node tree offers *more* than that — extra rotary axes, or several tool/workpiece branches (spindles, turrets) — the chain is redundant and the solver cannot choose. A sub-machine removes the ambiguity by naming the active branch: the tool node, the workpiece node, and exactly which linear and rotary axes participate.

The canonical example — a 6-axis mill (XYZ + ABC)

A common machining centre has linear `X Y Z` plus three rotary axes `A`, `B`, `C` — but the IK solver only ever drives two rotaries. Three rotaries are redundant. You resolve it by declaring two sub-machines, each pinning a usable *XYZ + two-rotary* chain:

- `XYZ + A + C`
- `XYZ + B + C`

The operation (or the user) then picks which sub-machine to use for a given job. Use the `SubMachine5x` variant, which adds the two rotary-axis fields:

```
<SubMachinesList>
    <SCType ID="XYZ_AC" Caption="A + C" type="SubMachine5x">
        <ToolNode DefaultValue="Spindle"/>   <!-- node carrying the tool -->
        <WrkNode  DefaultValue="Table"/>      <!-- node carrying the part -->
        <XAxisID  DefaultValue="AxisX"/>
        <YAxisID  DefaultValue="AxisY"/>
        <ZAxisID  DefaultValue="AxisZ"/>
        <R1AxisID DefaultValue="AxisA"/>      <!-- first rotary  -->
        <R2AxisID DefaultValue="AxisC"/>      <!-- second rotary -->
    </SCType>
    <SCType ID="XYZ_BC" Caption="B + C" type="SubMachine5x">
        <ToolNode DefaultValue="Spindle"/>
        <WrkNode  DefaultValue="Table"/>
        <XAxisID  DefaultValue="AxisX"/>
        <YAxisID  DefaultValue="AxisY"/>
        <ZAxisID  DefaultValue="AxisZ"/>
        <R1AxisID DefaultValue="AxisB"/>
        <R2AxisID DefaultValue="AxisC"/>
    </SCType>
</SubMachinesList>
```

Each sub-machine is one solvable XYZ + 2-rotary chain; together they cover the whole 6-axis machine without ever asking the solver to handle three rotaries at once. This machine is single-channel — sub-machines are needed regardless of channels.

A second example — twin-spindle / twin-turret mill-turn

A very common mill-turn centre has two tool carriers (an upper and a lower turret) and two part carriers (a left and a right spindle). Any turret can machine the part held by either spindle, so the usable tool + workpiece pairings are enumerated as sub-machines — plus one extra entry for spindle-to-spindle work:

- SubMachine 1 — UpperTurret → LeftSpindle
- SubMachine 2 — UpperTurret → RightSpindle
- SubMachine 3 — LowerTurret → LeftSpindle
- SubMachine 4 — LowerTurret → RightSpindle
- SubMachine 5 — RightSpindle *as the tool holder* → LeftSpindle *as the workpiece holder* — for axial drilling or pick-and-place / part transfer between the two spindles.

```
<SubMachinesList>
    <SCType ID="UpperOnLeft" Caption="Upper turret → Left
spindle" type="SubMachine">
        <ToolNode   DefaultValue="UpperTurret"/>   <!-- the turret HEAD, not a
single holder -->
        <WrkNode    DefaultValue="LeftSpindle"/>    <!-- the spindle node that holds
the part -->
        <XAxisID    DefaultValue="UpperX"/>
        <ZAxisID    DefaultValue="UpperZ"/>
        <ToolAxisID DefaultValue="UpperTurret"/>    <!-- the turret indexing axis -
->
        <Channel    DefaultValue="0"/>
    </SCType>
    <!-- ...UpperOnRight, LowerOnLeft, LowerOnRight — the other turret×spindle
pairs... -->
    <SCType ID="RightToLeft" Caption="Right spindle → Left
spindle" type="SubMachine">
        <ToolNode DefaultValue="RightSpindle"/>   <!-- the right spindle acts as the
tool carrier -->
        <WrkNode  DefaultValue="LeftSpindle"/>     <!-- the left spindle holds the
part -->
    </SCType>
</SubMachinesList>
```

> **`ToolNode`** / **`WrkNode`** name a *carrier* node, not a single holder. Give the nearest common parent of the holders involved — the turret head (parent of all its tool holders) or the spindle node — *not* an individual `TToolHolderNode` / `TWorkpieceHolderNode`. That is the whole point of (real)

> sub-machines: the branch's axes (`XAxisID` / `ZAxisID` / `ToolAxisID`) are stated once, on the sub-machine, instead of being repeated on every tool holder as the older "virtual sub-machine" (the connector `XAxisID`⋯ of §4) required. See the shipped `Machines\LatheMilling\MultiChannel\Puma MX2100ST\DOOSAN_PUMA_MX2100ST.xml` for a full real list (e.g. `ToolNode=AxisT` for the turret, `WrkNode=AxisC` for the spindle).

So each entry names which carrier holds the tool and which holds the part, giving the solver one unambiguous pair; the operator (or operation) picks the active sub-machine. The last entry shows that a spindle itself can be the tool carrier — e.g. the counter-spindle pushing the part onto a fixed drill, or taking the part over from the main spindle. Where the XYZ+ABC case used sub-machines to resolve redundant rotary axes, this case uses them to choose which of several tool/workpiece carriers form the active pair.

Declaration and fields

Sub-machines are listed in the machine's **`SubMachinesList`**; each entry is a `SubMachine` or a variant (the older array form `SubMachines` is obsolete).

Field	Meaning
`ToolNode` / `WrkNode`	The tool-carrying and workpiece-carrying carrier nodes (the nearest common parent of the holders — a turret head, a spindle), not an individual holder.
`XAxisID` / `YAxisID` / `ZAxisID`	The linear axes of this chain.
`ToolAxisID`	The tool / spindle axis.
`OriginG54BaseNode`	The node the sub-machine's base coordinate system is anchored to.
`OriginG54`	An arbitrary orientation (transform) for that base CS — commonly used on counter-spindle machines, where the base-CS Z must be inverted.
`ApproachRule` / `ReturnRule`	The general approach / return rule for this branch (see below).
`DetailedApproachRules` / `DetailedReturnRules`	Per-machining-mode variants of those rules, chosen automatically by operation type (see below).

Field	Meaning
`Channel`	The channel this sub-machine belongs to — relevant only on multi-channel machines (§6).

Sub-machine types. Which `SubMachine` type you use depends on the kinematics solver the branch needs, or on extra options specific to a kind of equipment. The plain `SubMachine` carries the common fields above; specialized types add the fields their solver requires. The set is open and will grow over time; at present two specializations exist beyond the base:

- **`SubMachine5x`** — for 5-axis machines. Adds `R1AxisID` / `R2AxisID`, the two rotary axes the 5-axis solver drives (used in the XYZ+ABC example above).
- **`TrevisanSubMachine`** — for machines with a U-axis turning function (the typical representatives are Trevisan-style machines). Adds the radial `UAxisID`.

Approach and return rules

`ApproachRule` / `ReturnRule` (and their detailed forms) describe how the machine moves between the tool-change point and the toolpath body for this sub-machine: the approach rule from the tool-change position to the *start* of the operation's toolpath, and the return rule from its *end* back to the tool-change position.

A rule is an ordered sequence of axis moves, written as groups separated by `;` that execute in turn; letters grouped together move together, and `Axis(value)` moves an axis to a given value. For example `ApproachRule = "BC; Y Z(10); X"` means: first turn B and C, then move Y and Z (to 10), then move X in.

A step can also be a command token rather than an axis move. In particular **`LCS`** switches the local coordinate system on or off at that point in the sequence — or, for the relevant 5-axis machining type, the TCPM mode — as seen in the `MillLCS` / `MillTCPM` variants below (`BC;Z;X;LCS;XY;Z`).

- **`ApproachRule`** / **`ReturnRule`** give a single general rule.
- **`DetailedApproachRules`** / **`DetailedReturnRules`** give per-machining-mode variants, so the right rule is selected automatically from the current operation's type:
  - `TurnRadial` / `TurnAxial` — radial / axial turning;
  - `MillRadial` / `MillAxial` — radial / axial ordinary 3-axis milling;
  - `MillLCS` — indexed 5-axis machining (local CS);
  - `MillTCPM` — continuous 5-axis machining (TCPM).

```
<SCType ID="MainMill" type="SubMachine">
    ...
    <ApproachRule DefaultValue="BC; Y Z(10); X"/>
    <ReturnRule   DefaultValue="Z(10); X; Z"/>
```

```
    <DetailedApproachRules>
        <TurnRadial DefaultValue="BS;Z;X"/>
        <TurnAxial  DefaultValue="BS;Z(10);X;"/>
        <MillRadial DefaultValue="BC;Z;X"/>
        <MillAxial  DefaultValue="BC;Z;X"/>
        <MillLCS    DefaultValue="BC;Z;X;LCS;XY;Z"/>
        <MillTCPM   DefaultValue="BC;Z;X;LCS;XY;Z"/>
    </DetailedApproachRules>
    <DetailedReturnRules> ... </DetailedReturnRules>
</SCType>
```

(real example: `Machines\LatheMilling\MultiChannel\Puma MX2100ST\DOOSAN_PUMA_MX2100ST.xml`)

Pick the type that matches the branch's kinematics; a single machine can mix types across its sub-machines.

6 Control channels (simultaneous work)

A channel is an independent stream of motion/NC code that runs at the same time as the others. On a multi-channel machine (twin-spindle / multi-turret lathes, turn-mill centres, Swiss-type lathes) different nodes, axes and state parameters belong to different channels, and some axes are shared. You assign membership with the integer **`Channel`** member on the relevant nodes/axes/state-parameters (and `ChannelWorkpiece` where the workpiece side differs):

```
<MachineStateParameters>
    <SCType ID="AxisX1Pos" type="TMachineStateParameter"><Channel DefaultValue="0"/>
... </SCType>
    <SCType ID="AxisX2Pos" type="TMachineStateParameter"><Channel DefaultValue="1"/>
... </SCType>
</MachineStateParameters>
```

`Channel="0"` and `Channel="1"` put the two cross-slides on separate channels; an axis left on the common channel is shared. Multi-channel machines are not only Swiss-type — see the shipped examples under `$(SCHEMAS_FOLDER)\LatheMilling\MultiChannel` (e.g. *IndexG160*, *Puma MX2100ST*, *PumaTT2000SY*). `SwissTemplate.xml` is one common, ready-made special case (a 2-channel Swiss lathe) that also shows turret/adapter `TCaseNode` selectors in context; shipped machines built on it — for example the Hanwha Swiss-type lathes under `$(SCHEMAS_FOLDER)\LatheMilling\SwissType` — are real examples of that template in use.

> In short: *channels* run simultaneously and say *which independent stream* a node belongs to; *sub-machines* are used alternately and say *which subset of nodes/axes forms one solvable tool+workpiece chain*. A single-channel machine can still need sub-machines (the XYZ+ABC mill above); a multi-channel turn-mill uses both — sub-machines within each channel.

Back to Machine descriptors | guide index

3D models for machines

A machine scheme is a tree of nodes (Machine descriptors); each node can carry a 3D model that is drawn — and collision-checked — at that node's position in the kinematic chain. This page explains how those models are referenced, prepared and aligned. The XML side is small (three things: `ImageFile`, `VisMatrix`, `Use3DModelColors`); the rest is about preparing the geometry correctly.

1 Attaching a model to a node

A node's model is named by its **`ImageFile`** property (inherited from `TMachineNode`), and positioned for display by **`VisMatrix`** inside `VisualProperties`:

```
<SCType ID="Column" Caption="Column" type="TMachineNode">
    <ImageFile DefaultValue="Images\column.osd"/>
    <VisualProperties>
        <VisMatrix>                          <!-- adjusts the model only, not the
kinematics -->
            <SCType ID="T1" type="TRotateZ" DefaultValue="90"/>
        </VisMatrix>
        <Color><R DefaultValue="0.4"/><G DefaultValue="0.4"/><B
DefaultValue="0.45"/></Color>
    </VisualProperties>
</SCType>
```

A node is drawn using its kinematic frame (the product of the `Matrix` chain from the root down to it, see §1), and then **`VisMatrix`** is applied on top, to the model only. So if a CAD model was exported in an awkward orientation, correct it with `VisMatrix` without disturbing how the node moves.

> One node = its own geometry. Each moving node must carry only the geometry that moves with it. A correctly prepared machine is therefore an assembly of separate models — one per node — not a single monolithic model. This is what lets the parts move correctly relative to one another.

2 Where model files live

- Model files normally sit next to the machine's XML file, or in an **`Images\`** subfolder for tidiness, and are referenced by relative paths (e.g. `Images\column.osd`). The same applies to the bitmap images shown in the machine library.
- A complete machine is therefore a folder of several files — the descriptor XML plus its models and images.
- Keep the folder self-contained and portable. Do not reference files in a neighbouring machine's folder — that breaks when the machine is moved or shared. (Machine schemes are stored, as folders, inside projects precisely so projects stay portable.)

- These folders do not have to live under the standard distribution aliases such as `$(SCHEMAS_FOLDER)`: the machine library can be told to scan any user folders for machines.

3 Units and coordinate system

- Units must match the machine descriptor. Model geometry must be saved in the same units as the machine's *Measurements* — millimetres for a metric machine, inches for an imperial one — exactly like the `Matrix` translations (see coordinate conventions and units). If the running unit system differs, the system tries to scale the machine once at load, but that can misbehave for heavily parametrized schemes — so prepare models in the units the machine will use.
- Coordinate system. Geometry is shown in the global CS it was drawn in, combined with the node-hierarchy matrices and then `VisMatrix`. In other words, model the part in the machine's world CS (Z up, X right, Y away — coordinate conventions), and use `VisMatrix` only to fix up a model that came out misaligned.
- Often it is a zero-position issue, not a **`VisMatrix`** one. A model frequently looks "off" simply because the axis it sits on was modelled at a non-zero position. Then you usually do not need `VisMatrix` — just set that axis's **`DesignTimeAxisValue`** (node reference) to the position at which the model was drawn, and the node (model *and* kinematics together) lines up correctly at design time. Reach for `VisMatrix` only when the model itself is rotated/offset relative to its node frame.

4 The OSD and STL formats

Two formats are accepted:

- OSD (*OpenGL Stream Data*) — the system's own format and the one to prefer. It is a triangulated mesh stored as a stream of OpenGL primitives/commands, and unlike STL it can also carry edges and surface normals (for nicer rendering) and extra snap/key points (e.g. circle centres from the original CAD that are otherwise lost in triangulation).
- STL — the common interchange mesh format; also supported, but without the extra edge/normal/snap information.

Producing an OSD

1. Import a CAD model (the system reads many common CAD formats) into the CAM system.
2. On the Model page, select the node (sub-folder) you want.
3. Use Save as and choose the OSD format.

The same export is available through the CAM API and CAM IPC (see the `cam-api-examples` repository), so model preparation can be scripted.

5 Visual vs. collision geometry — keep it light

Right now the same model is used both for display and for collision checking (this may change in future). That means model detail is a trade-off between looks and speed — favour speed: do not build schemes

from over-detailed CAD. Both the CAM system and MachineMaker provide a tool that interactively helps strip out invisible, internal or very small surfaces to simplify a model. (There is no fixed triangle-count guideline; simpler is better.)

This also interacts with the *Simulation* settings — *Revolution bodies simulation* and *Collisions to ignore* — described in Machine Setup → Simulation.

6 Colours — `Use3DModelColors`

`VisualProperties` accepts an optional boolean attribute **`Use3DModelColors`**:

- **`False`** (default) — the node is drawn with the `Color` set in its `VisualProperties` (the XML descriptor wins).
- **`True`** — the node is drawn with the colours stored in the 3D model itself.

```
<VisualProperties Use3DModelColors="True">
    <!-- this node keeps the colours baked into its .osd model -->
</VisualProperties>
```

7 Ready-made models

MachineMaker has an online library of devices, robots, machines and cells. It supplies prepared sets (models *plus* the extra descriptor information), not loose model files, and inserts them into the cell description for you — they cannot be loaded as individual files. So when you prepare a scheme by hand, you typically prepare the models individually for that machine, ending up with the self-contained, portable folder described in §2.

Back to Machine descriptors | guide index