CNC-Calc Post Processor Manual

Introduction

Before you start to write a complete post processor, make sure that you cannot make an existing post processor work by simply modifying the Globals section in an existing one. Please refer to the document "*CNC-Calc Post Processor_Basic configuration*" to see how the output can be modified by means of changes in the Globals section.

One of the new features in CNC-Calc version 7 is the addition of a Post Processor. This makes it possible for the end user to format the created NC program.

The Post Processors are written in JavaScript. JavaScript was selected because it makes it possible to define variables with both global and local scopes. It also gives the end user the possibility to declare and define functions that can be used in formatting and calculation. The writer of the post processor is in this way given access to all standard mathematical functions defined in standard JavaScript and all the standard features that this language contains.

It is beyond the scope of this manual to cover the JavaScript language, which is, however, described in a great number of books and on the World Wide Web. A good source of information is the site:

http://www.w3schools.com/js/

The best way to write a post processor is to use an existing one, for example one of those installed with CNC-Calc v7. Select the one that comes closest to the machine that the new post processor needs to support. In that way the framework is already there, and it is easier to just modify existing code and debug it.

CNC-Calc and the post processor cooperate to format and create the correct NC program. This cooperation is established via various functions that are called from CNC-Calc with the appropriate parameters. The parameters are then used to calculate and output one or more lines of code in the NC program.

The process of programming a new post processor can be roughly divided into 4 different areas:

1. Setting up global variables that are used by the core of the post processor and by the post processor developer alike. These global variables define which decimal point to use, what kind of end of line that should be used, how circular moves are written etc.

2. Creating the various variables with the correct formatting attached. For each value type handled by the post processor, there needs to be a defined format. This might seem like overkill, but since these variables do more than just format the values it is well worth the trouble. Things like modality and keeping track of the last value written are automatically handled by these variables, and along with that, complex formatting will be performed without the need of any additional code.

3. Writing the functions to handle the calls from CNC-Calc. For each kind of movement or action there exists a function in the post processor, which is called when CNC-Calc needs to create a program block for this kind of movement/action. This means that functions exist for linear moves, circular moves, drilling header, drilling footer etc. Most of these functions are quite simple, and when an existing post processor is used as template very few changes should be needed.

4. Creating and configuring the drill cycles needing support by CNC-Calc. Drill cycles have always been one of the more complicated topics with post processors, and it is also one of the more difficult tasks when writing a post processor for CNC-Calc. In the CNC-Calc post processor drill cycles are handled in four different sections:

• A list of structures defining what input fields should be enabled for input
• **onDrillingHeader**, which is a function called with a structure of parameters that contain the values entered in the UI
• **onDrillingPoint**, which is called once for each hole in the drilling operation
• **onDrillingFooter**, which is called once as the last function in the drill operation

As stated earlier, when you need to write a new post processor, it is to be preferred to find an existing post processor that is very close to the one needed. Make a copy of this and use the copy as a template, correcting only the sections that need to be changed. With luck, it might only be one or two sections that need correcting, and even if not so lucky you still start out with a functional post processor, which can then be gradually tweaked to fit what is needed.

1. The global variables and their use

At the very start of the Post Processor there is a section called **Globals**. In this section we have declared various variables that are used to define some of the rules used to format a valid NC program. These variables can always be used by the programmer, but some are also used ‘behind the scene’ by the post processor core.

Name	**Normal Values**	**Description**
decimalMark	‘.’ Or ‘,’	This variable defines the decimal mark to be used when decimal values are shown.
linebreak	"\n"	Defines the character sequence to be used to terminate the lines in the NC program.
variableDelimeter	" "	When more than one variable is shown in a single line, the **variableDelimeter** defines how they should be separated.
tolerance	0.02	The **tolerance** is normally used in the user program to define the smallest entity that should be handled. If for instance a very small, circular movement is made, the controller may mistake this as being a 360 degree movement.
showSequenceNumbers	true/false	If **showSequenceNumbers** is set to true, all blocks in the NC program will be formatted with a line number. This is used in the function **writeBlock** that can be modified by the programmer.
sequenceNumberStart	10	If **showSequenceNumbers** is set to true**, sequenceNumberStart** will define the starting number used for the first block. This is used in the function **writeBlock** that can be modified by the programmer.
sequenceNumberIncrement	5	**sequenceNumberIncrement** is only used if **showSequenceNumbers** is set to true. The **sequenceNumberIncrement** defines the jump in block numbers between blocks. This is used in the function **writeBlock** that can be modified by the programmer.
useRadius	true/false	Circular moves are normally written using the R (radius) or I, J, and K values (center coordinates). When **useRadius** is set to true, circular moves will use the R value. This is used in the function **onCircular** that can be modified by the programmer.

absoluteArcCenter	true/false	If **useRadius** is set to false, the circular moves will use the I, J, and K values. The value of **absoluteArcCenter** defines if these values should be given as absolute center coordinates or relative to the start of the move. This is used in the function **onCircular** that can be modified by the programmer.
xDiameterProg	true/false	This field is only used in post processors for lathes. The value of **xDiameterProg** defines if the X values should be output as diameter values. If not set to true, the X values will be output as radius.
iDiameterProg	true/false	This field is only used in post processors for lathes. If **useRadius** is set to false the circular moves will use the I, J, and K values. The value of **iDiameterProg** indicates how I values of circular moves should be formatted. If **iDiameterProg** is true the I values will be given as diameter values, otherwise they will be as radius. This is used in the function **onCircular** that can be modified by the programmer.

2. Creating an output format

One of the most important aspects of all data outputs is the correct formatting of the numerical values. The formats may vary depending on the values being handled. When you for example take a look at the feedrate F it typically has no decimals, as opposed to any of the coordinates X, Y or Z that usually have 3 decimals or more. Because of the importance of the formatting the post processor handles it in a way that offers functionalities suitable for most cases. In order to define a format we use the function **createFormat**, which takes a list of value pairs used to define how the format should look. In the table below the possible values and their default values are shown.

Name	**Default value**	**Description**
Prefix	“”	What should be written in front of the value
width	0	Indicates the width or the integer part of the value
zeropad	False	Zeropad relates to the integer part of the value. If a width of 4 is set and zeropad is set to true, 1 will be formatted as 0001.
decimals	0	Indicates the number of decimals with which the value should be formatted.

scale	1	The value is multiplied with this scale factor
forceDecimal	False	Forces the decimal part of the value to always be written with the number of decimals indicated in *decimals*. If decimals is set to 3 and forceDecimal is set to true, 3.0 will be formatted as 3.000
forceSign	False	Makes it possible to always force a sign. For example, 4.25 will be written as +4.25

Let us say we want to create the format to output the X, Y, and Z values. In this example we will not use the default value, but define all the possible fields in the **createFormat** format.

We would like to output an X value of 3 as X3.000; so let us look at the values we need to create this output:

Name	**Value**	**Description**
Prefix	“”	Here we could have used “X” in order to get the value prefixed with an ‘X’. The reason why it is left as “” is because we will use this format to output all the values for X, Y, and Z. Please refer to the next section to see how we handle the prefix.
width	0	Is set to 0. This does not mean that the width will be zero, but it will be set to the actual width of the integer part of the value being formatted.
zeropad	False	Is set to False. Again, this is to ensure that the integer part is as wide as the actual width of the integer part of the value being formatted.
decimals	3	The value will be formatted with max. 3 decimals.
scale	1.0	Since we do not want to scale the value, the scale will be set to 1.0
forceDecimal	True	We set the forceDecimal to true so all values will be formatted with exactly 3 decimals (4 becomes 4.000).
forceSign	False	We do not want to have the sign shown in front of positive values.

The final definition of our format will look like this:

xyzFormat = *createFormat*({prefix : ””, width : 0, zeropad : false, decimals : 3, scale : 1.0, forceDecimal : true, forceSign : false});

3. Using a format to create a variable

When we have created a format we will then use the function *createVariable* to tie this format to one or more variables. As the format above was to be used to define the format for X, Y, and Z, we gave it the name xyzFormat. The prefix field is set to “” to indicate that the format should have no prefix. Now we need to tie this format to the X, Y, and Z variables. To do this, we will use the function *createVariable*. When we create a variable in this way we get more than the formatting defined in the format we attach. We also get modality handled automatically, and the member function *getLast***()** returns the last value used.

The function *createVariable* has two parameters. The first is a list of value pairs and the second has a format like the one created above. In the table below the possible values and their default values are shown.

Name	**Default Value**	**Description**
prefix	“”	That, which should be written in front of the value.
force	False	This relates to the subject of modality. When force is set to false it will only be written if it has changed since the last time it was used. On the other hand, if force is set to true it will always be written.

Now we will create the variables with the function *createVariable***.** As mentioned earlier, we will not use the default values, but declare them all instead.

xOutput = *createVariable*({prefix: "X" , force: false }, **xyzFormat**);
yOutput = *createVariable*({prefix: "Y" , force: false }, **xyzFormat**);
zOutput = *createVariable*({prefix: "Z" , force: false }, **xyzFormat**);

We have now created 3 variables: **xOutput**, **yOutput,** and **zOutput**. They all use the previously defined format, and since *force* is set to false they are all modal. These variables have to be handled in a special way in order to get and set their values.

The function call:

var **xString** = *xOutput.format*(**vx**);

will assign to the variable **xString** a string that is formatted using the defined **xyzFormat** on the value **vx**. After this call the **xOutput** will remember the value of vx, and if we call it next time with the same value it will return an empty string because **xOutput** was declared with *force* set to false and is therefore modal.

The function call:

var **lastX** = *xOutput.getLast*(**vx**);

will assign the last numerical value, with which *xOutput.format* was called. So if the last call was *xOutput.format*(**vx**), then **lastX** would be assigned the value of **vx**.

This section has explained how we create format and output variables, and how these are used. To see more examples of their use, please refer to one of the post processors installed with CIMCO Edit/CNC-Calc 7.

4. The Post Processor functions

Besides Helper functions used by other functions, most functions are called directly from CNC-Calc in order to generate the NC program. In the following, we will take a closer look at each of the functions currently available in the post processor.

Some functions are called at 'fixed' places in the post processing, while others are called when some kind of movements are encountered. The functions called at fixed places will in the following be named *Framework functions,* and these will be described here.

4.1. Framework Functions

onInit() This function will be called before any other function in the JavaScript. It is normal to set up all formatting and output variables in this function. This will ensure that they have the correct set-up when used later in the post processor. Other variables needing a default value can also be handled here. Since this function does not return any value, no CNC program can be generated here.

onOpen(**isClipboard**) is called at the start of an actual program. This function makes it possible to create and return NC lines at the start of the program. Program starts might be different if they are exported to the Editor rather than the clipboard. Therefore, the function is called with a *Boolean parameter* that indicates the kind of export being requested. It would be practical to save the state of this variable for future use in for instance the **onClose()** function.

onClose() is the last function called in the post processing of an operation. This would be the place to return "M30" and other strings that should be inserted at the end of a program. If we are post processing a program snippet with **export to clipboard**, then M30 should probably not be inserted. To prevent this we can look at the **isClipboard** parameter that was passed to the *onOpen* function, and then make the right output on the basis of this.

4.2. Movement Functions

The movement functions are functions that handle the actual movements on both lathes and mills. They are called from CNC-Calc whenever a movement is encountered, and they create the actual text output, which is returned to CNC-Calc.

Common for all movement functions, they are called with the movement type as the last parameter. This movement type is in turn used to get the right feedrate set by CNC-Calc based on the parameters used in the given operation. To resolve the movement to a given feedrate, call the function *getFeedrate*(**movement**) and this should produce a correctly formatted feedrate.

Regarding compensation, much the same approach should be used. None of the movement functions has compensation as a parameter. Instead, when the compensation changes, the function *onCompensation*(**comp**) is called from CNC-Calc. This is done to retain a compensation change to be captured from those movement functions that actually handle compensation (on some controllers a compensation change in a circular movement will trigger an alarm). So the *onCompensation* function retains the change, which is then retrieved and handled by the movement functions with a call to the function *getCompensation*(), which again returns the correct G-code or other essential control text.

Milling and turning have different movement functions. These movement functions return a correctly formatted movement string for linear or circular movements. Their declarations look like the following:

Milling:

function *onLinear*(**vx**, **vy**, **vz**, movement)
function *onCircular*(**cx**, **cy**, **cz**, **ex**, **ey**, **ez**, **isCW**, **movement**)

Turning:

function *onLinear*(**vx**, **vz**, movement)
function *onCircular*(**cx**, **cz**, **ex**, **ez**, **isCW**, **movement**)

The coordinates **vx**, **vy**, and **vz** are of the type ‘double’ and together they define the end point of the linear movement.

For the circular movement, **cx**, **cy**, and **cz** are all ‘double’ and they constitute the absolute coordinates for the center of the movement.

The coordinates **ex**, **ey**, and **ez** define the end point of the circular movement.

With these values it is possible to format the correct output to be returned to CNC-Calc. You might need to create a value relative to the start point. This is, however, not included in the call, but can be found if an output variable was used to format the given coordinate. This variable retains the last formatted value, which can be retrieved using the function call to the method *output.getLast*(). With the above knowledge and by studying the post processors that were installed together with CNC-Calc, it should be possible to write the correct movement functions.

5. Creating and handling canned drill operations in Milling

In the drilling operations for Milling, special steps must be taken in order to handle the posting of the holes correctly:

First, we must make sure that it is possible to input the correct drilling parameters for a given operation in the UI. To do this, the **MillDrillingFields** list in the post processor needs to be filled out correctly. The **MillDrillingFields** list contains structures of the type shown below. The first field is a string, but the other fields are all Booleans, and they define if a given field in the UI should be enabled or disabled.

Field Name	**Description**
name	This is the name of the drilling operation. Any name may be used as long as it is unique. It is used in the functions *onDrillingHeader*, *onDrillingPoint* and *onDrillingFooter* to identify the type of drill cycle being posted.
isLonghand	Orders the operations in the UI. All Canned cycles are grouped together and the same goes for longhand cycles.
plungingFeedrate	Enable / disable the **plungingFeedrate** field.
retractFeedrate	Enable / disable the **retractFeedrate** field. This field is only used if the **plungingFeedrate** field is enabled. Otherwise it is always disabled.
firstDepth	Enable / disable the **firstDepth** field.
degressionType	Enable / disable the **degressionType** field.
degression	Enable / disable the **degression** field.
peckingRetract	Enable / disable the **peckingRetract** field.
minimumDepth	Enable / disable the **minimumDepth** field.
firstFeedrateFactor	Enable / disable the **firstFeedrateFactor** field.
upperDwell	Enable / disable the **upperDwell** field.
lowerDwell	Enable / disable the **lowerDwell** field.
pitch	Enable / disable the **pitch** field.
tappingSpeed	Enable / disable the **tappingSpeed** field.
retractSpeed	Enable / disable the **retractSpeed** field.

After these structures to handle the UI are filled out correctly, we can start looking at the definition of the functions to handle the canned operations. Canned drill cycles are generated using three different functions:

function *onDrillingHeader*(**drillHeader**) is called before any hole coordinates are passed to the post processor. The drillheader is a structure that contains all the parameters from the drilling parameters UI. These parameters are:

Name	**Description**
name	The name of the selected drilling operation. This is the name that was defined in the MillDrillingFields structure and is used to identify the operation.
isLonghand	Indicates if this is a longhand or canned operation. This parameter makes it easier to organize these two types into two different sections of code.
retractPlane	The retract plane is the height to which the drill is retracted between each individual hole. This value is always absolute.
referencePlane	This is the value of the absolute reference plane. If either the safe distance or end depth is given as relative coordinates, they are relative to this plane.
safeDistanceIsAbsolute	The value of this Boolean indicates, whether the safe distance is given as an absolute value or relative to the reference plane.
safeDistance	The value of the safe distance, where the transition from rapid move to feed is performed.
endDepthIsAbsolute	This Boolean indicates, whether the safe distance is given as an absolute value or relative to the reference plane.
endDepth	This is the depth of the hole.
firstDepth	When pecking is used, this is the depth of the first peck.
firstFeedrateFactor	The feedrate could be set to a different value for the first peck. This factor determines what it is.
degressionType	If some kind of degression should be used between pecks, this contains the type (as enumerator) and it could be set to none, value or factor.
degression	If the degression was set to value or factor, this value is used to calculate the new depth.
peckingRetract	This is the amount that the drill should retract between pecks.
minimumDepth	If degression is used, the minimum depth can set a lower limit of the peck depth.
usePlungingFeedrate	This Boolean indicates if a plunging feedrate should be used or not.
plungingFeedrate	Is the value of the plunging feedrate.
useRetractFeedrate	Indicates if retract feedrate should be used or not. Notice, that if no plunging feedrate is used, it has no meaning to define a retract feedrate.
retractFeedrate	This is the retract feedrate.

useTopDwelling	If top dwelling is used, this Boolean is set to true, otherwise false.
topDwelling	The value of the top dwelling. Usually this will be the time in seconds.
useBottomDwelling	Used to indicate if bottom dwelling should be used.
bottomDwelling	The bottom dwelling time. Usually given in seconds.
pitch	If this is a threading operation, this parameter contains the pitch of the thread.
useTappingSpeed	Used to indicate, if a special speed should be used for tapping.
tappingSpeed	If different speeds are used for tapping and retract, this parameter contains the tapping speed.
useRetractSpeed	Used to indicate, if a special speed should be used for retract. This only has meaning, if tapping speed is used.
retractSpeed	This parameter contains the retract speed.

Normally, the *onDrillingHeader* can be set up to do a few things. Firstly, it should always save drillHeader in a global variable so it can be accessed from functions *onDrillingPoint* and *onDrillingFooter*. Secondly, it could calculate the values used to generate the header of the canned cycle, but this can also be done in the *onDrillingPoint* function as long as the header was saved here.

Function *onDrillingPoint*(x, y) is called for each hole that needs to be posted. The first call to this function should write the header of the canned operation. The reason for this is that if no holes were selected and the header was written in *onDrillHeader*, then the header would be invalid.

Function *onDrillingFooter*() is called as the last function after all holes have been posted. In this function the termination of the canned operation should be inserted into the NC program. The function returns the valid termination, if one exists.

6. Creating and handling canned drill operations in Turning

As with Milling, in the drilling operations for Turning, special steps must be taken in order to handle the posting of the holes correctly:

Instead of the **MillDrillingFields** for Milling, the **TurnDrillingFields** list in the post processor is used to configure the way the UI is shown. In the **TurnDrillingFields** the entry fields in the UI are either enabled or disabled depending on the parameters, with which the user should be concerned. Below is a table showing the configurable fields for the drill operations in Turning.

Field Name	Description
name	The name of the drilling operation. Any name may be used as long as it is unique. It is used in the function onDrilling to identify the type of drill cycle that is being posted.
isLonghand	Is used to order the operations in the UI. All Canned cycles are grouped together and the same goes for longhand cycles.
plungingFeedrate	Enable / disable the **plungingFeedrate** field.
retractFeedrate	Enable / disable the **retractFeedrate** field. This field is used only if the **plungingFeedrate** field is enabled. Otherwise it is always disabled.
usePecking	Enable / disable the **usePecking** field.
firstPeckDepth	Enable / disable the **firstDepth** field.
subsequentPeckDepths	Enable / disable the **degressionType** field.
peckClearance	Enable / disable the **degression** field.
peckRetract	Enable / disable the **peckingRetract** field.
useUpperDwell	Enable / disable the **upperDwell** field.
uselowerDwell	Enable / disable the **lowerDwell** field.
pitch	Enable / disable the **pitch** field.

When these structures have been filled out for all the supported drilling operations we can write the turning functions to support these. As opposed to drilling in milling, where there can be multiple holes in one operation, center drilling drills only one hole in the center of the part. Therefore, the drilling for turning is simpler than it is for milling and there is only one function, called once for each drilling operation. That function is described below:

Function *onDrilling(drillHeader)* is the only function called for a drilling operation and it contains all the parameters needed to generate the NC code for this operation.

Name	Description
Name	The name of the selected drilling operation.
isLonghand	Indicates if this is a longhand or canned operation. It makes it easier to separate these two types into two different sections of code.
startDepth	The absolute height of the start of the material.
endDepth	The absolute depth of the hole.
safeDistance	The value of the safe distance, where the transition from rapid move to feed is performed. This is an absolute value.
retractDistance	The retract plane is the height to which the drill is retracted after the operation is performed.
usePlungingFeedrate	This Boolean indicates, if a plunging feedrate should be used or not.
plungingFeedrate	Is the value of the plunging feedrate.
useRetractFeedrate	Indicates if retract feedrate should be used or not. Notice that if no plunging feedrate is used, it has no meaning to define a retract feedrate.
retractFeedrate	This is the retract feedrate.
Pitch	If this is a threading operation, this parameter contains the pitch of the thread.
UsePecking	If the hole is going to be drilled with a pecking operation, this parameter is set to true.
firstPeckDepth	When pecking is used, this is the depth of the first peck.
subsequentPecksDepths	After the first peck has been made, this is the depth of all subsequent depths.
peckClearance	This is the distance over the previous depth, where a transition from rapid to feed is made.
PeckRetract	This is the amount that the drill should retract between pecks.
useUpperDwell	If top dwelling is used, this Boolean is set to true, otherwise false.
upperDwell	The value of the top dwelling. Usually, this will be the time in seconds.
useLowerDwell	Used to indicate, if bottom dwelling should be used.
lowerDwell	The bottom dwelling time. Usually given in seconds.
useDrillTipCompensation	In some cases, where we want to drill completely through a part, we can add the length of the tip to the depth of the hole (the depth is here the thickness of the part).
drillDiameter	The diameter of the drill. This diameter is used to calculate the length of the drill tip.
drillTipAngle	This is the angle of the tip used to calculate the tip length.

7. Setting up the Backplot to show the Post Processed file correctly

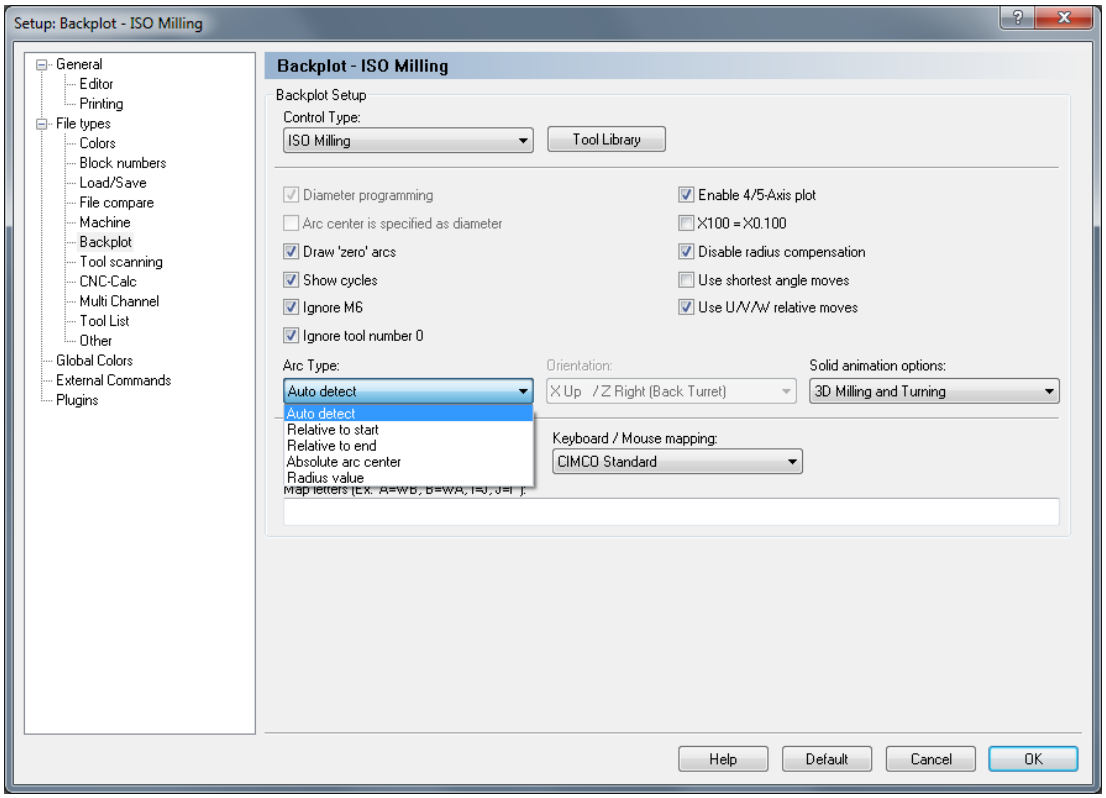

Figure 1: ISO Milling Backplot Set-up

In Figure 1 we see the Backplot Configuration for ISO Milling and we will now see how we need to change the setup to reflect the variables in the Globals section.

The only part of the Milling setup that can be influenced by the variable in the Globals section is the *Arc Type* drop-down menu. The *Arc Type* is the drop-down menu that is shown open in Figure 1. The choices in this drop-down menu depend on the Globals variables **useRadius** and **absoluteArcCenter**. In the table below the values of the variables and the corresponding Arc Type are shown.

Variable values	**Correct Arc Type Selection**
useRadius = true absoluteArcCenter = false	Radius Value
useRadius = true absoluteArcCenter = true	Radius Value
useRadius = false absoluteArcCenter = false	Relative to Start
useRadius = false absoluteArcCenter = true	Absolute Arc Center

In Turning there are two more values in the Globals section that will influence the setup of the Backplot. Below in Figure 2 is shown the Backplot Configuration for ISO Turning.

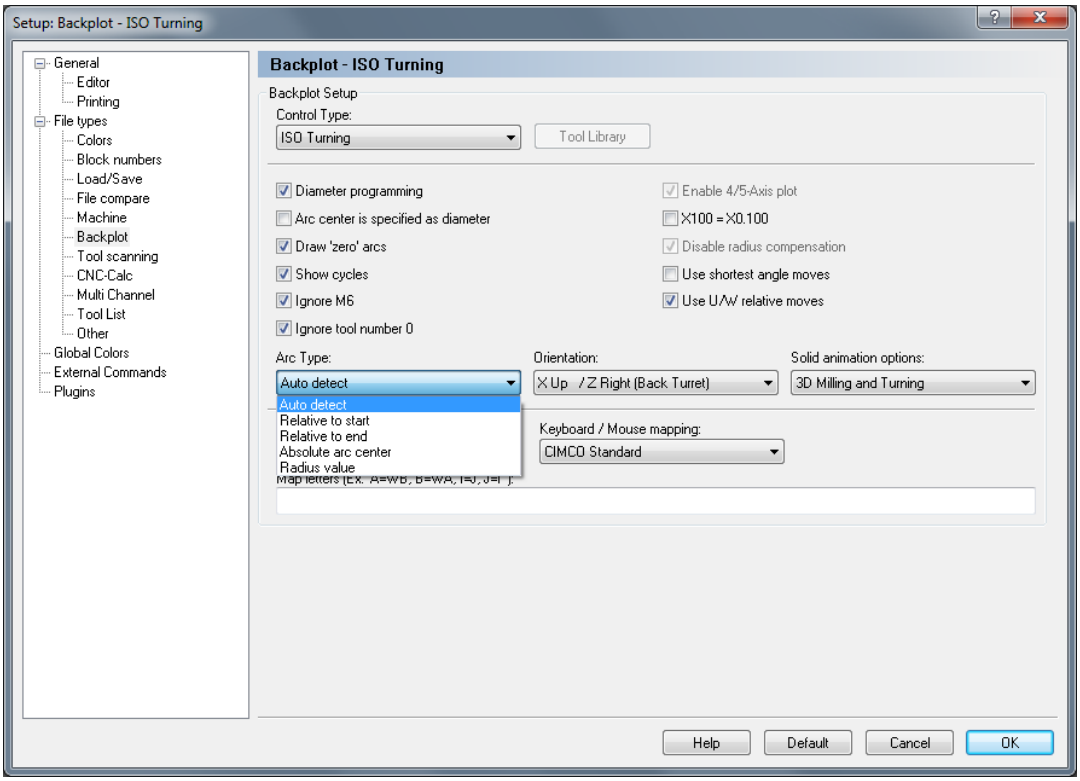

Figure 2: ISO Turning Backplot Set-up

The *Arc Type* in Turning should be selected from the Drop-down menu based on the Globals variables **useRadius** and **absoluteArcCenter**, exactly as described under Milling. When we look at Turning we also have to consider the possibilities regarding diameter programming. In the post processor the diameter programming is handled by the Global variables **xDiameterProg** and **iDiameterProg**. These variables correspond to the fields *Diameter programming* and *Arc Center is specified as diameter*. In the following table you can see how the values of the Global variables correspond to the fields in the Turning setup.

Variable Values	**Correct Backplot Setup**
xDiameterProg = false iDiameterProg = false	*Diameter programming* is unchecked *Arc Center is specified as diameter* is unchecked
xDiameterProg = false iDiameterProg = true	*Diameter programming* is unchecked *Arc Center is specified as diameter* is unchecked
xDiameterProg = true iDiameterProg = false	*Diameter programming* is checked *Arc Center is specified as diameter* is unchecked
xDiameterProg = true iDiameterProg = true	*Diameter programming* is checked *Arc Center is specified as diameter* is checked

If the *Arc Type* drop-down menu and the checkboxes *Diameter programming* and *Arc Center is specified as diameter* are checked respectively unchecked as described above, the Backplot should show the correct tool paths.