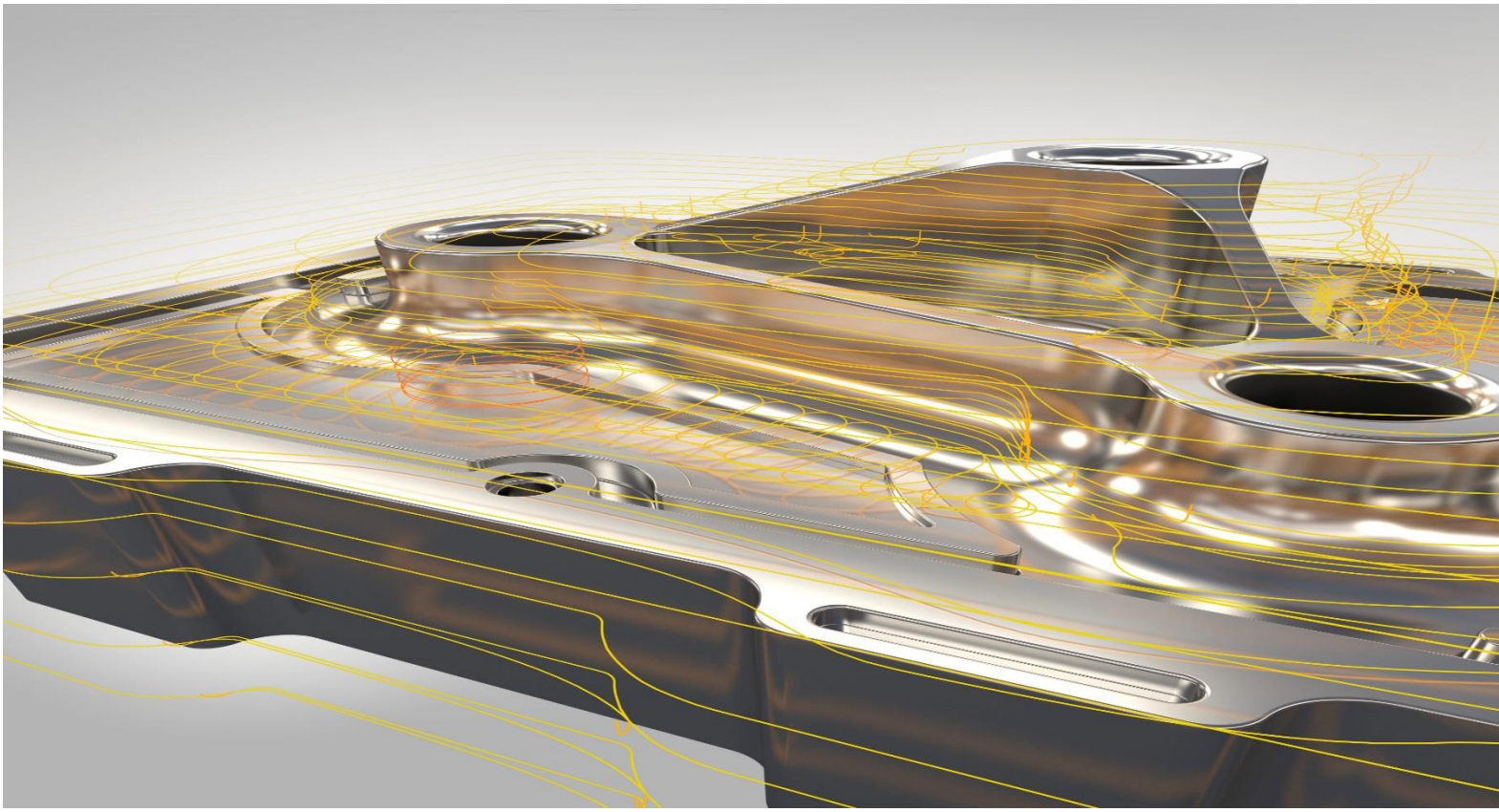
AUTODESK.



Post Processor Training Guide

For use with Fusion CAM, Inventor HSM, HSMWorks

CAM Post Processor Guide 8/28/19

Table of Contents

Table of Contents

Table of Contents

Table of Contents

1 Introduction to Post Processors

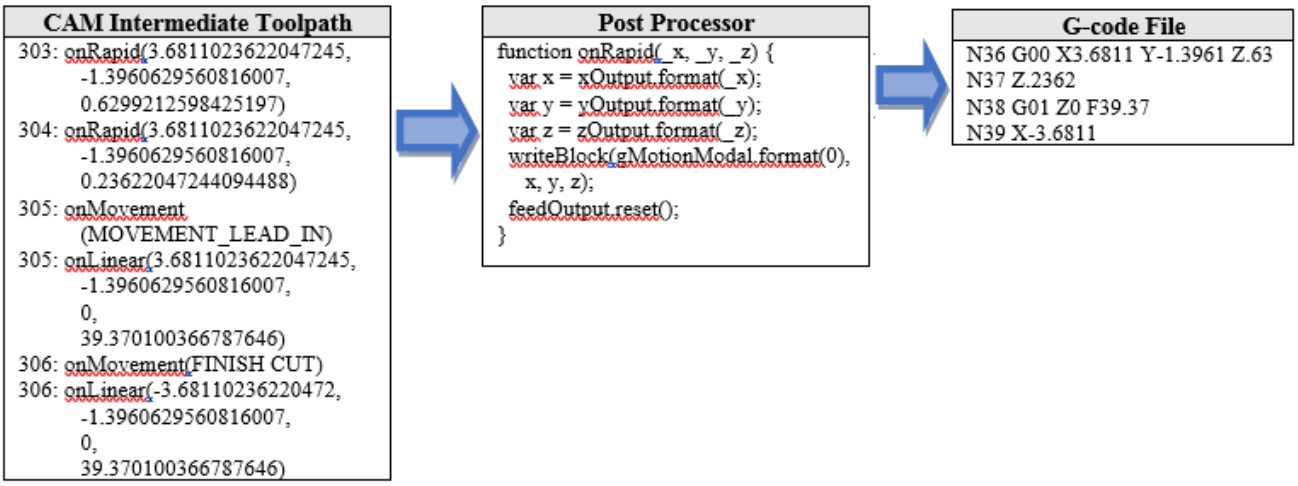
1.1 Scope

This manual is intended for those who wish to make their own edits to existing post processors. The scope of the manual covers everything you will need to get started; an introduction to the recommended editor (Autodesk Fusion 360 Post Processor Editor), a JavaScript overview (the language of Autodesk post processors), in-depth coverage of the callback functions (onOpen, onSection, onLinear, etc.), and a lot more information useful for working with the Autodesk post processor system.

It is expected that you have some programming experience and are knowledgeable in the requirements of the machine tool that you will be creating a post processor for.

1.2 What is a Post Processor?

A post processor, sometimes simply referred to as a "post", is the link between the CAM system and your CNC machine. A CAM system will typically output a neutral intermediate file that contains information about each toolpath operation like tool data, type of operation (drilling, milling, turning, etc.), and tool center line data. This intermediate file is fed into the post processor where it's translated into the language that a CNC machine understands. In most cases this language is a form of ISO/EIA standard G-code, even though some controls have their own proprietary language or use a more conversational language. All examples in this manual uses the ISO/EIA G-code format.

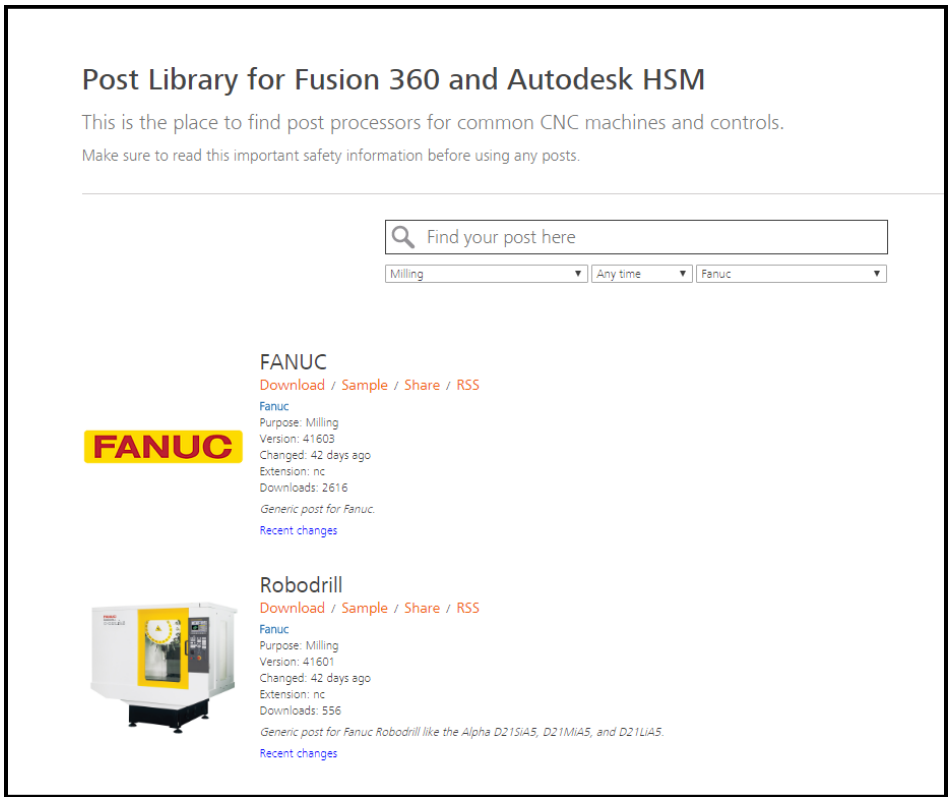


If you would like a bit more information on the G-code format the CNC Handbook contains a lot of useful information including a further explanation of the G-code format in Chapter 5 CNC Programming Language.

Though most controls recognize the G-code format the machine configuration can be different and some codes could be supported on one machine and not another, or the codes could be interpreted differently, for example one machine may support circular interpolation while another requires linear moves to cut the circle, which is why you will probably need a separate post processor for each of your machine tools.

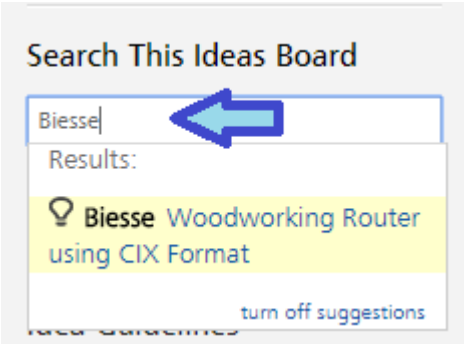
1.3 Finding a Post Processor

The first step in creating a post processor is to find an existing post that comes close to matching your requirements and start with that post processor as a seed. You will never create a post processor from scratch. You will find all the generic posts created by Autodesk on our online Post Library. From here you can search for the machine you are looking for by the machine type, the manufacturer of the machine or control, or by post processor name.

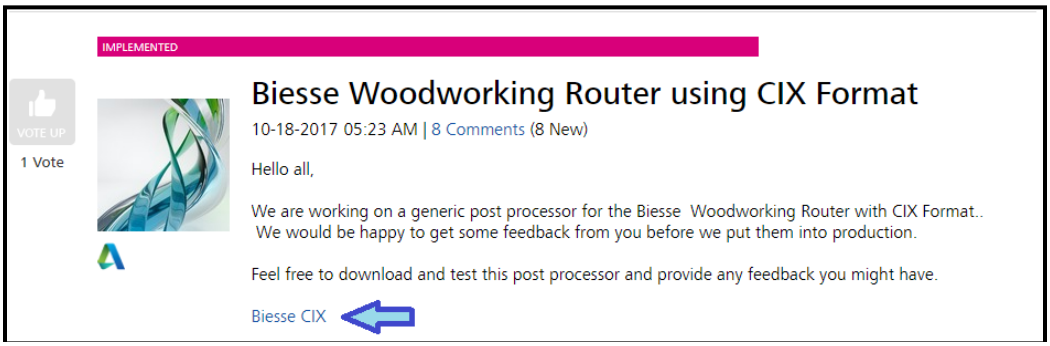


Other places to check for a post processor include the HSM Post Processor Forum or HSM Post Processor Ideas.

It is possible that Autodesk has already created a post processor for your machine, but has not officially released it yet. These posts are considered to be in Beta mode and are awaiting testing from the community before placing into production. You can visit the HSM Post Processor Ideas site and search for your post here. This site contains post processor requests from users and links to the posts that are in Beta mode. You can search for your machine and/or controller to see if there is a post processor available.



Searching For a Post Processor on Ideas or the Forum



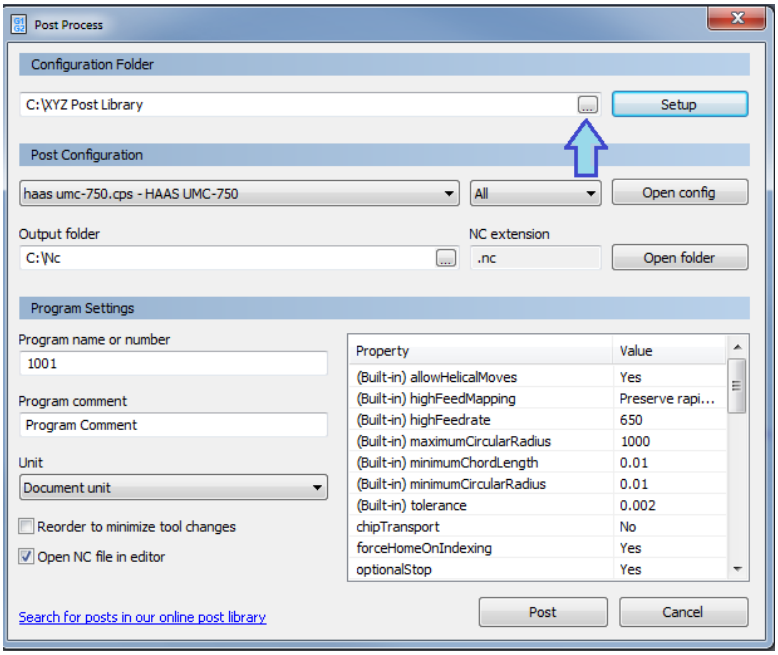
Beta Post Processor Found on HSM Post Processor Ideas

If your post processor is not found, then you should search the HSM Post Processor Forum using the same method you used on the HSM Post Processor Ideas site. The Post Processor Forum is used by the HSM community to ask questions and help each other out. It is possible that another user has created a post to run your machine.

You should always take care when running output from a post processor for the first time on your machine, no matter where the post processor comes from. Even though the post processor refers to your exact name, it may be setup for options that your machine does not have or the output may not be in the exact format that you are used to running on the machine.

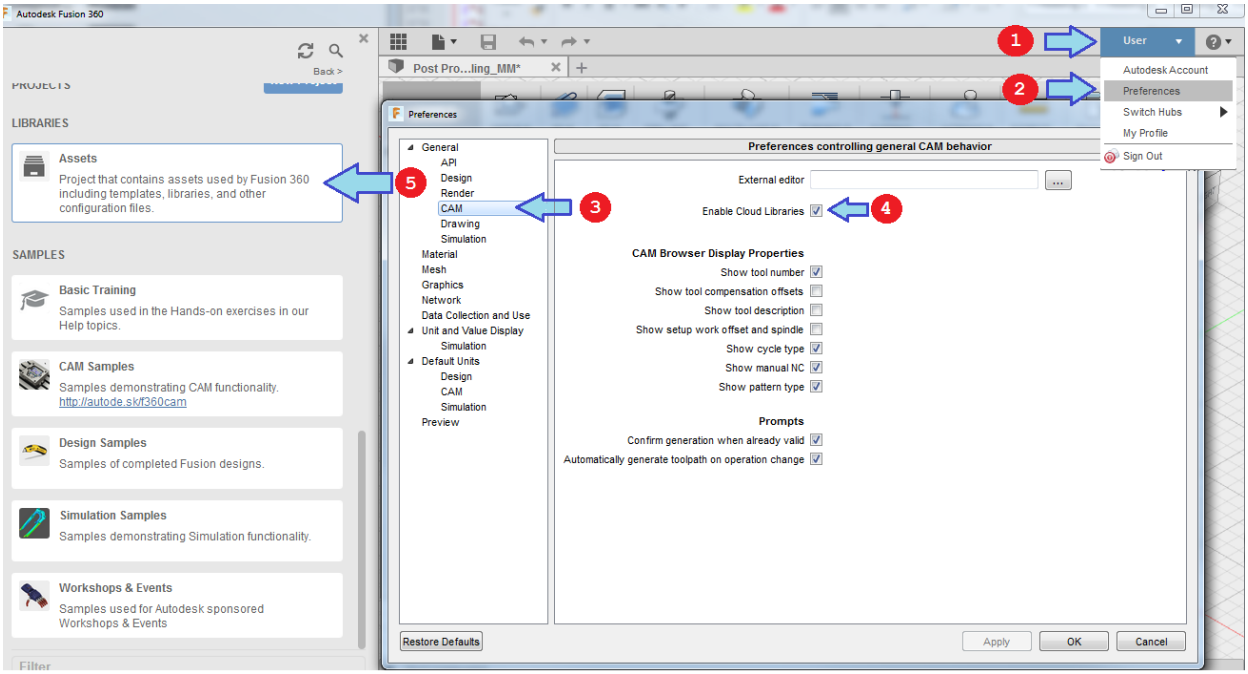
1.4 Downloading and Installing a Post Processor

Once you find the post processor that closely matches your machine you will need to download it and install it in a common folder on your computer. If you are working on a network with others then this should be in a networked folder so everyone in your company has access to the same library of post processors.

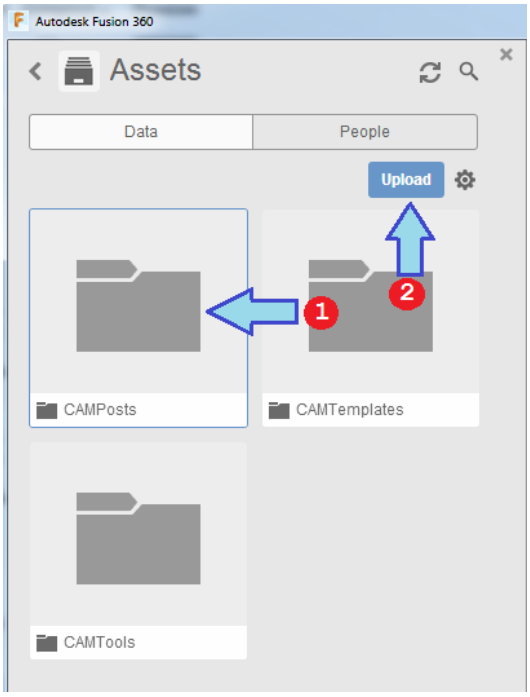


Selecting the Local Post Processor Folder

When using Fusion 360 it is recommended that you enable cloud posts and place it in your Asset Library. This way post processors, tool libraries, and templates will be synched across devices and users at a company.

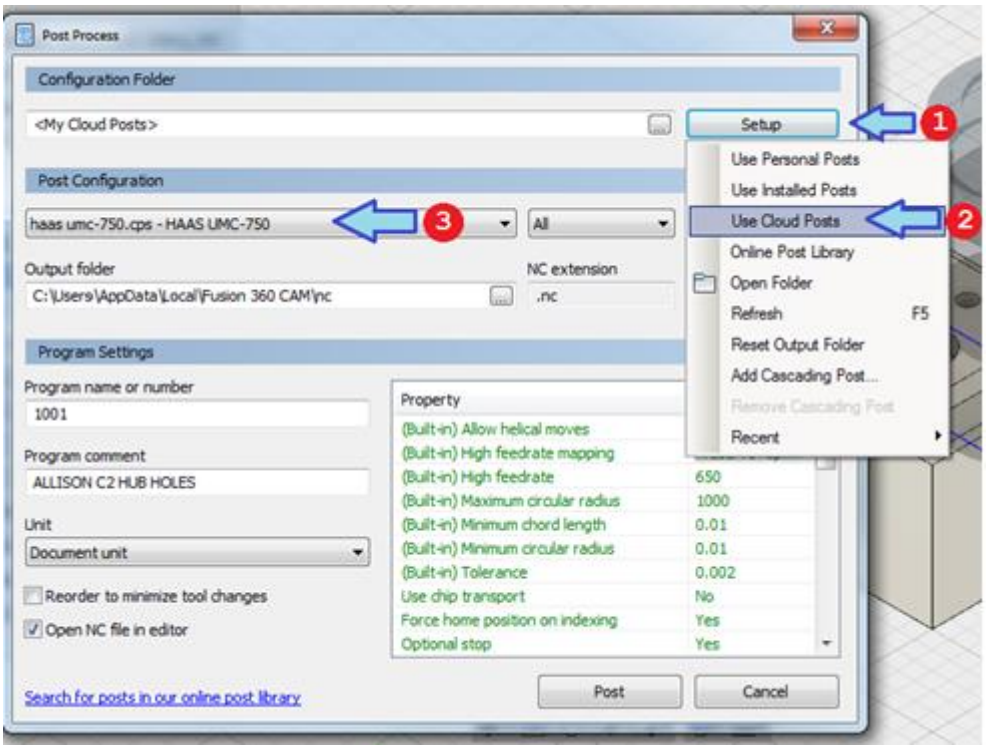


Enabling Cloud Post Processors in Fusion 360



Double Click the CAMPosts Folder and then Press the Upload Button

Once you have uploaded your post(s) to the Cloud Library you can access these from Fusion 360. You do this by pressing the Setup button in the Post Process dialog and selecting your post from the dropdown menu.



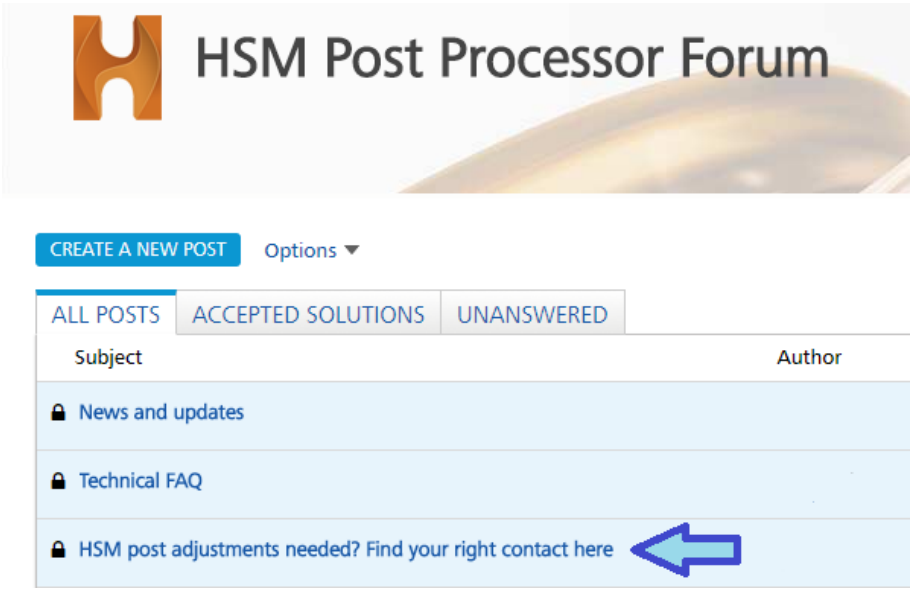
Selecting Your Post from the Cloud Library

In all cases you will want to avoid placing posts in the production install folder as these can be overwritten when HSM is updated. Downloading your posts to a separate folder means that you can reduce your list of post processors that show up in the Post Process dialog to those that you use in your shop.

1.5 Creating/Modifying a Post Processor

Once you find a post processor that is close, but not exact to the requirements of your machine you will need to make modifications to it. The good news is, all of posts are open source and can be modified without limitation to create the post you need. You have a few options for making the modifications.

1. Make the modifications yourself using this manual as a guide and by asking for assistance from the HSM community on the HSM Post Processor Forum.
2. Visit HSM Post Processor Ideas and create a request for a post processor for your machine. Other users can vote for your request for Autodesk to create and add your post to our library.
3. Contact one of our CAM partners who offer post customization services. These partners can be found on the HSM Post Processor Forum at the top of the page.



Finding HSM CAM Partners

No matter which method you decide to use to create your post processor, you should have enough information available to define the requirements, which includes as much of the following as you can gather.

1. A post processor (.cps) that will be used as the seed post.
2. Sample NC code that has run on your machine.
3. The machine/control make and model.
4. The type of machine (mill, lathe, mill/turn, waterjet, etc.).
5. The machine configuration, including linear axes, rotary axes setup, etc.
6. A programming manual for your machine/control.

1.6 Testing your Post Processor – Benchmark Parts

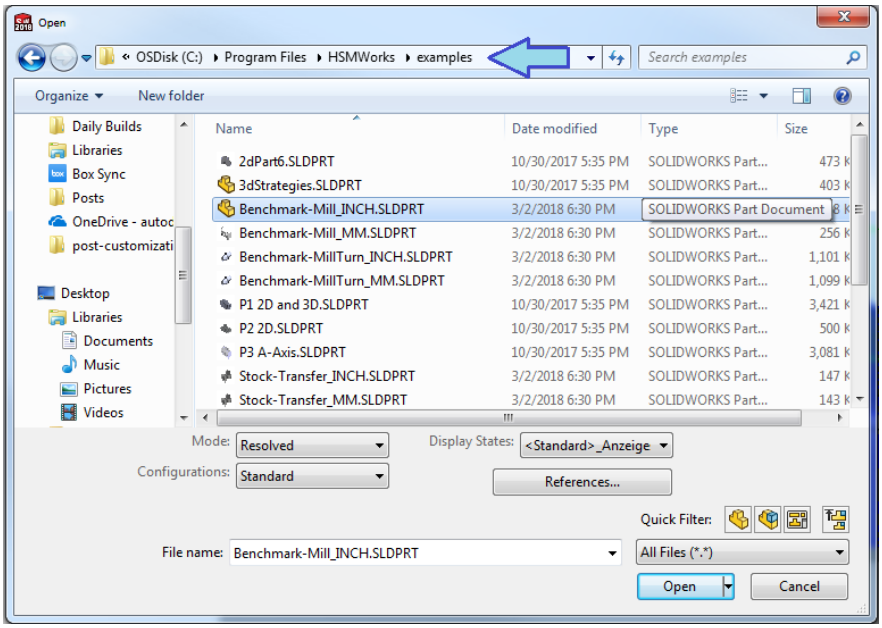
When testing your post processor, you will need a part with cutting operations to post against. We have created standard benchmark parts for this specific purpose. These parts cover the most common scenarios you will come across when testing a post processor and are available for HSMWorks, Inventor HSM, and Fusion 360 CAM. They are available in both metric and inch format for all three CAM systems. There are five different benchmark parts.

- Milling
- Turning and Mill/Turn
- Stock Transfers
- Waterjet-Laser-Plasma
- Probing

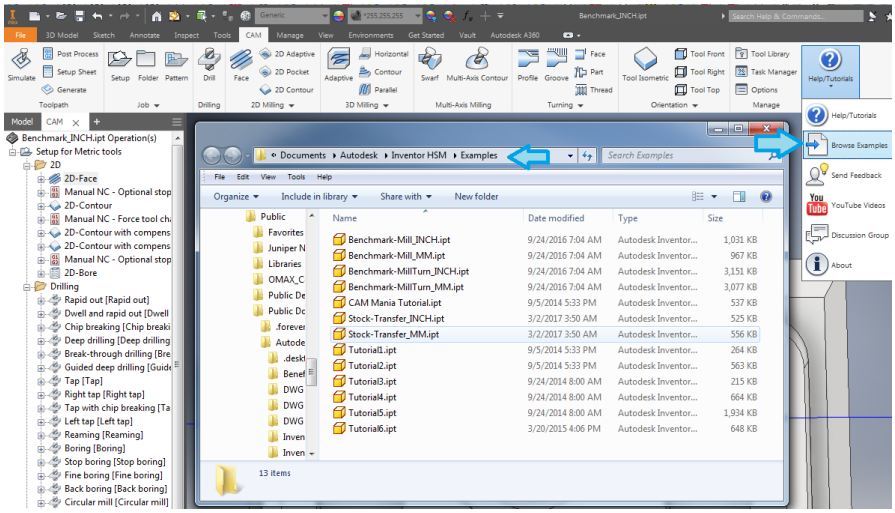
You can visit the Autodesk Manufacturing Lounge for more information on the benchmark parts.

1.6.1 Locating the Benchmark Parts

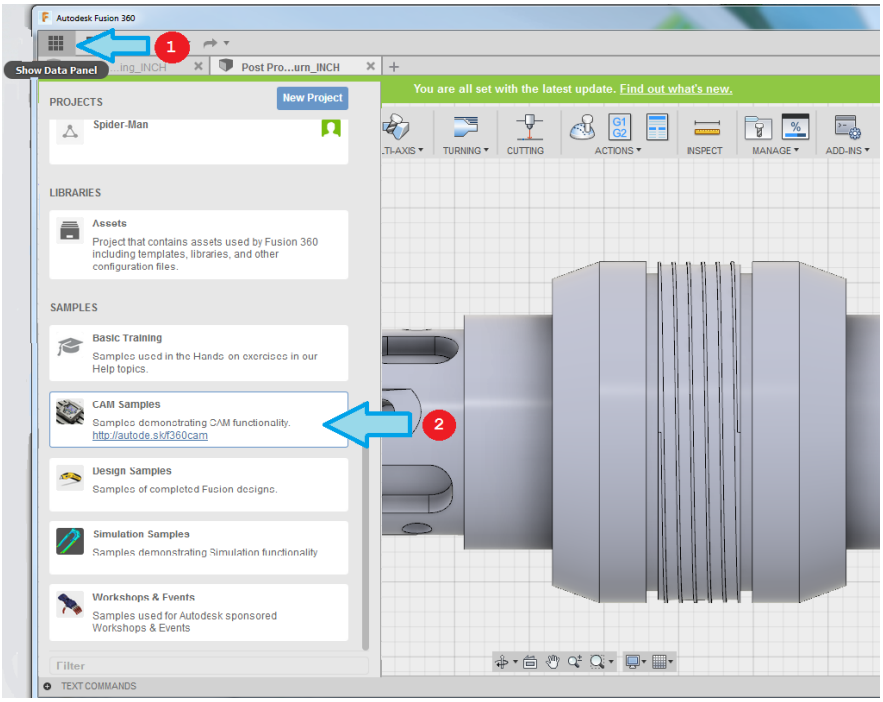
The benchmark parts are available to all users of Autodesk CAM and can be accessed in the Samples folder for each product.



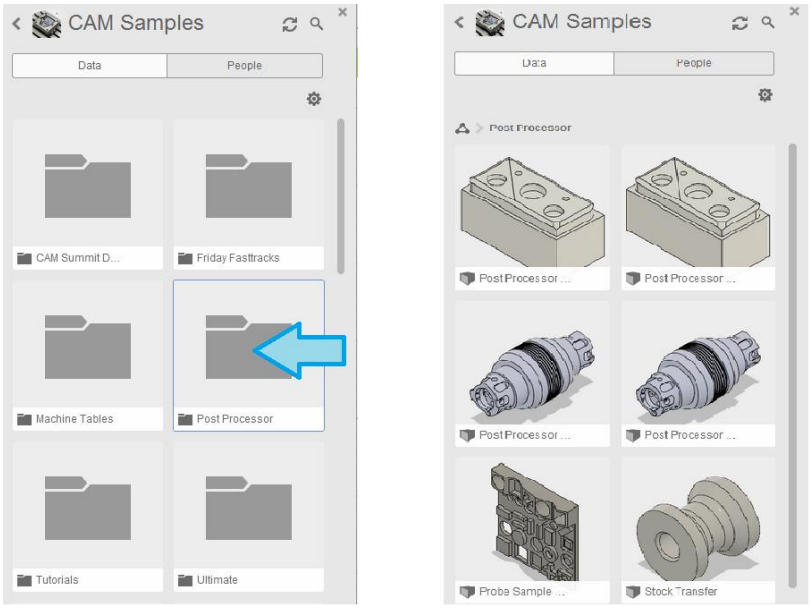
HSMWorks Sample Parts
C:\Program Files\HSMWorks\examples



Inventor HSM Sample Parts
C:\Users\Public\Public Documents\Autodesk\Inventor HSM\Examples



Fusion 360 CAM
Select the Data Panel and Double Click on CAM Samples

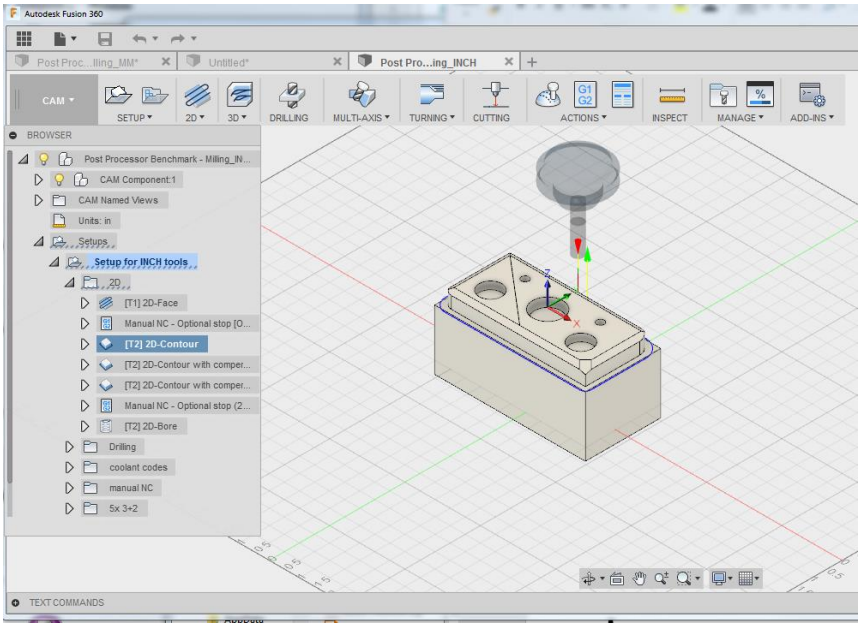


Fusion 360 CAM (continued)
Double Click on Post Processor to Display the Sample Parts

1.6.2 Milling Benchmark Part

The milling benchmark parts include the following strategies.

- 2D
- Drilling
- Coolant codes
- Manual NC commands
- 3+2 5-axis
- 5-axis simultaneous

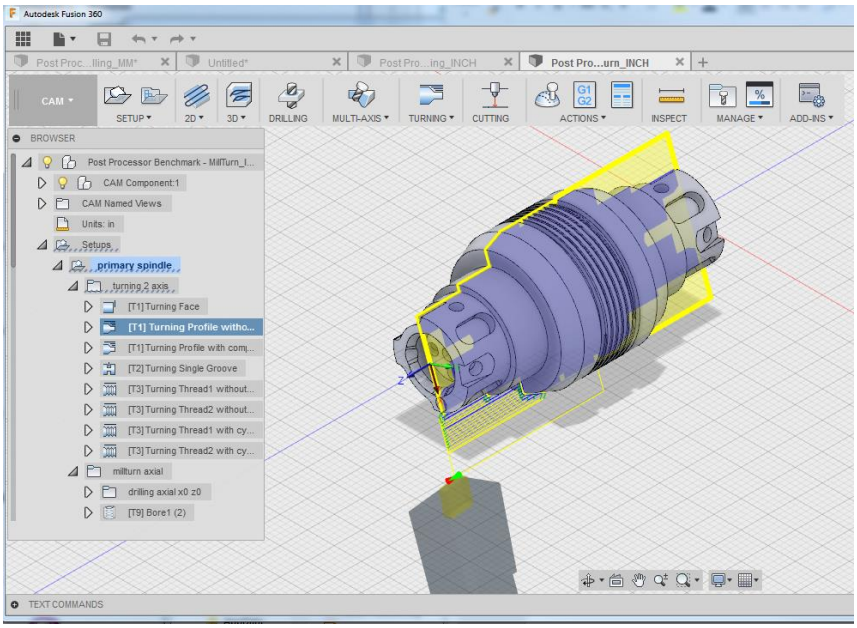


Mill Benchmark Part

1.6.3 Mill/Turn Benchmark Part

The mill/turn benchmark parts contain the following strategies.

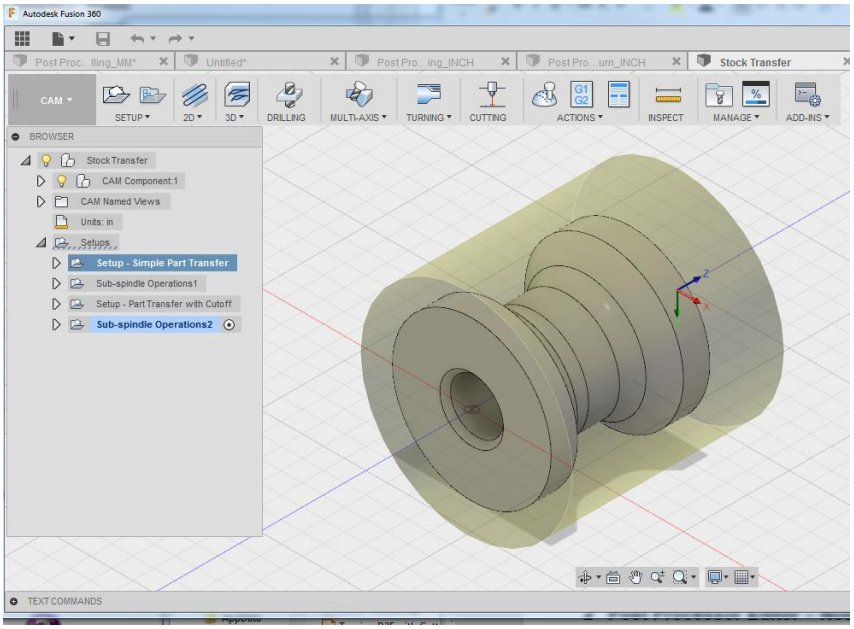
- Primary and Secondary spindle operations
- Turning
- Axial milling
- Radial milling
- 5-axis milling



1.6.4 Stock Transfer Benchmark Part

The stock transfer benchmark part contains the following strategies.

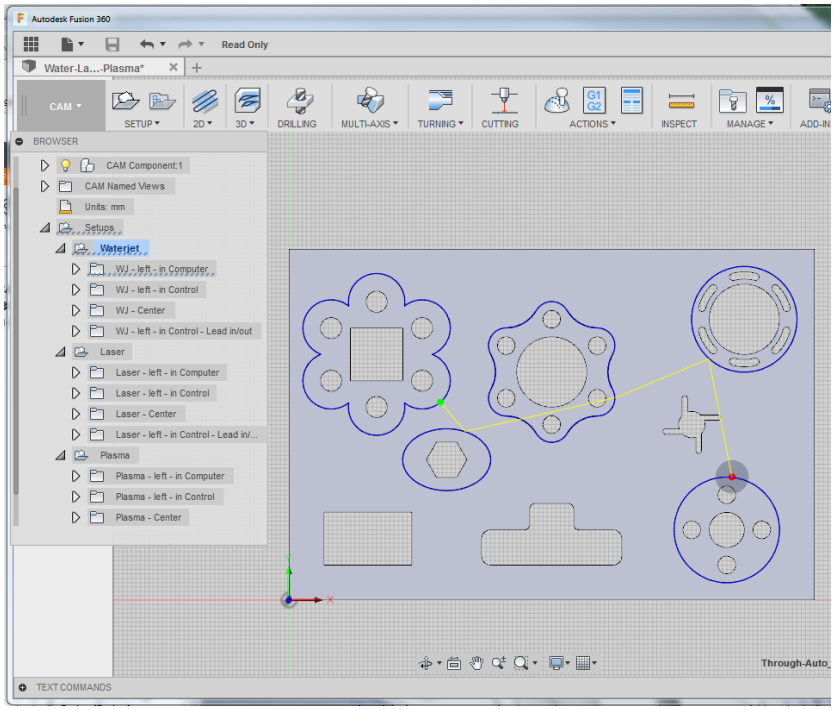
- Primary and Secondary spindle operations
- Simple part transfer
- Part transfer with cutoff



Stock Transfer Benchmark Part

The Waterjet-Laser-Plasma benchmark part contains the following strategies.

- Waterjet
- Laser
- Plasma
- Lead in/out
- Radius compensation

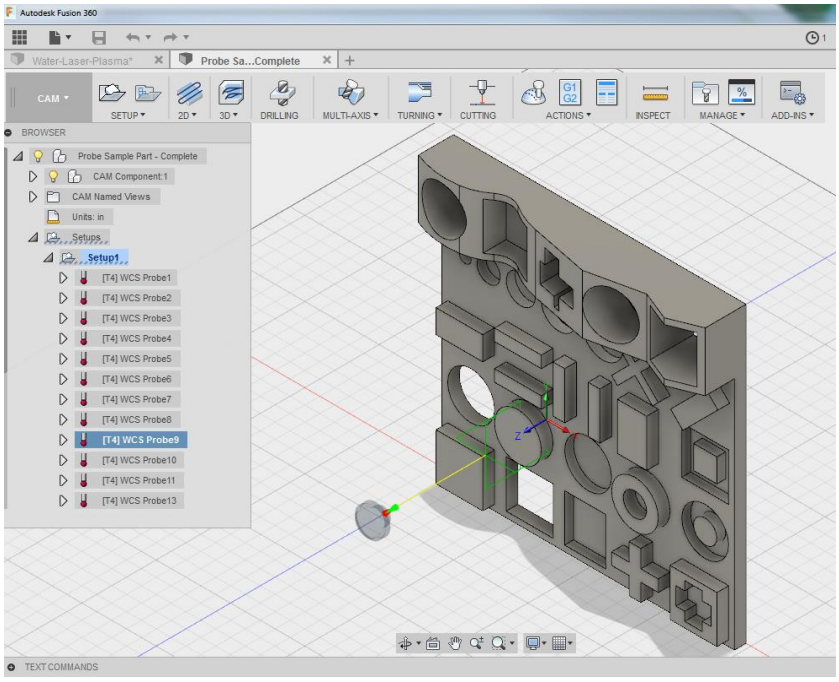


Waterjet-Laser-Plasma Benchmark Part

1.6.5 Probing Benchmark Part

The Probing benchmark part contains the following strategies.

- Various probing cycles



Probing Benchmark Part

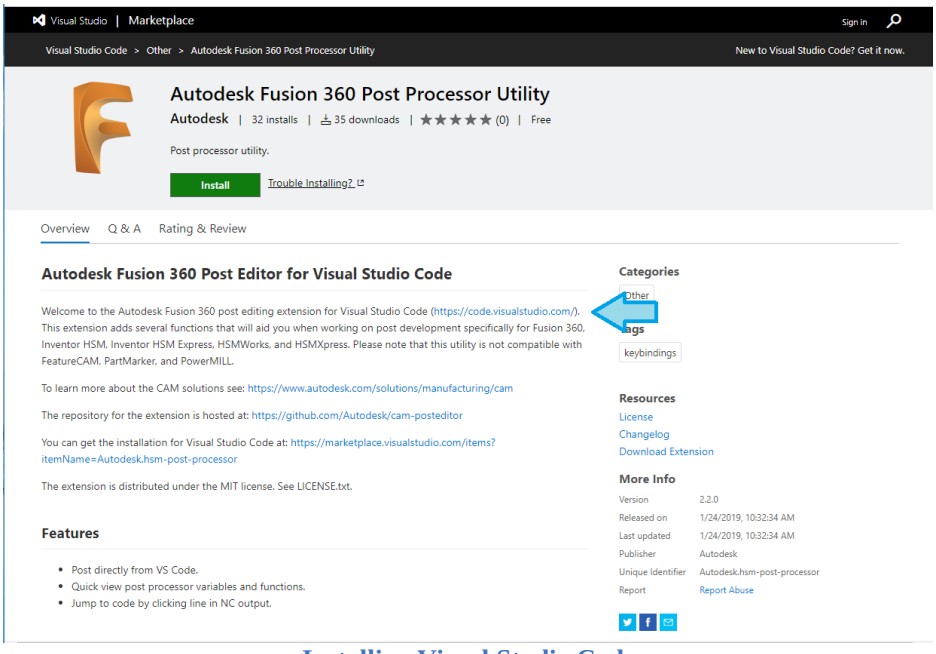
2 Autodesk Post Processor Editor

Since Fusion 360, Inventor HSM, and HSMWorks post processors are text-based JavaScript code, they can be edited with any text editor that you are familiar with. There are various editors in the marketplace that have been optimized for working with programming code such as JavaScript. We recommend Visual Studio Code with the *Autodesk Fusion 360 Post Processor Utility* extension. Using this editor provides the following benefits when working with Autodesk post processors.

- Color coding
- Automatic closing and matching of parenthesis and brackets
- Automatic indentation
- Intelligent code completion
- Automatic syntax checking
- Function List
- Run the post processor directly from editor
- Match the output NC file line to the post processor command that created it

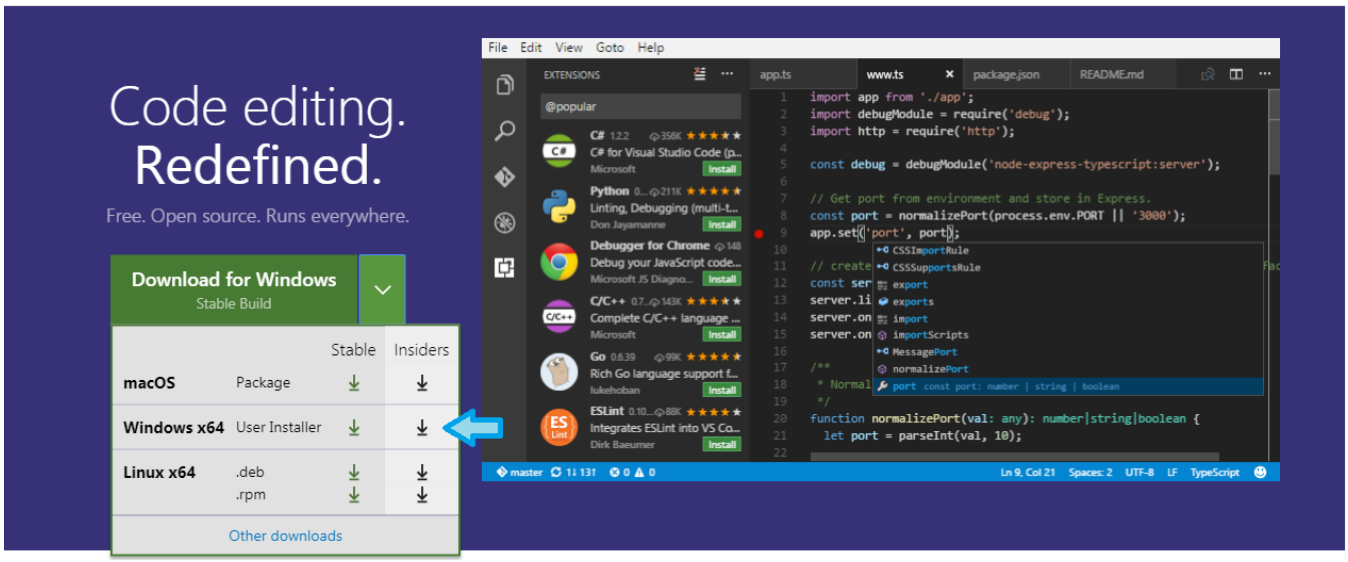
2.1 Installing the Autodesk Post Processor Editor

Before you can use the VSC editor you will need to install it. The easiest way is to visit the Autodesk Fusion 360 Post Processor Utility page in the Visual Studio Marketplace, where you can download VSC and then the Autodesk Fusion 360 Post Processor Utility extension. Please note that the Visual Studio Code site changes quite frequently, so the directions/pictures in this section might not be exactly what you see on the screen, but the installation steps should still be similar.



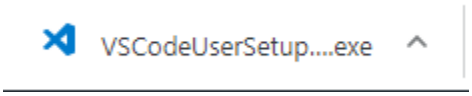
Installing Visual Studio Code

This link will take you to the Visual Studio Code installation page. Select the correct version for your operating system.



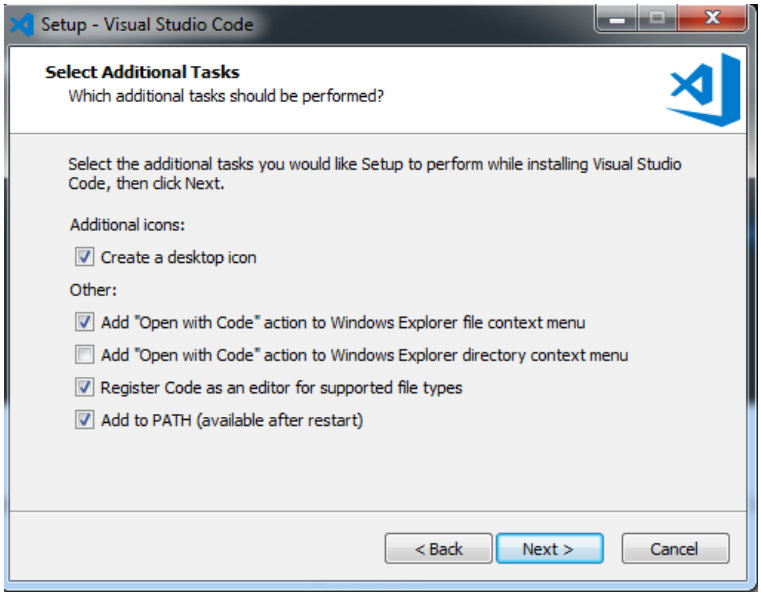
Installing the Windows Version of Visual Studio Code

This will download an installation program that you can run to do the actual install. Left click on the installation program to execute it.



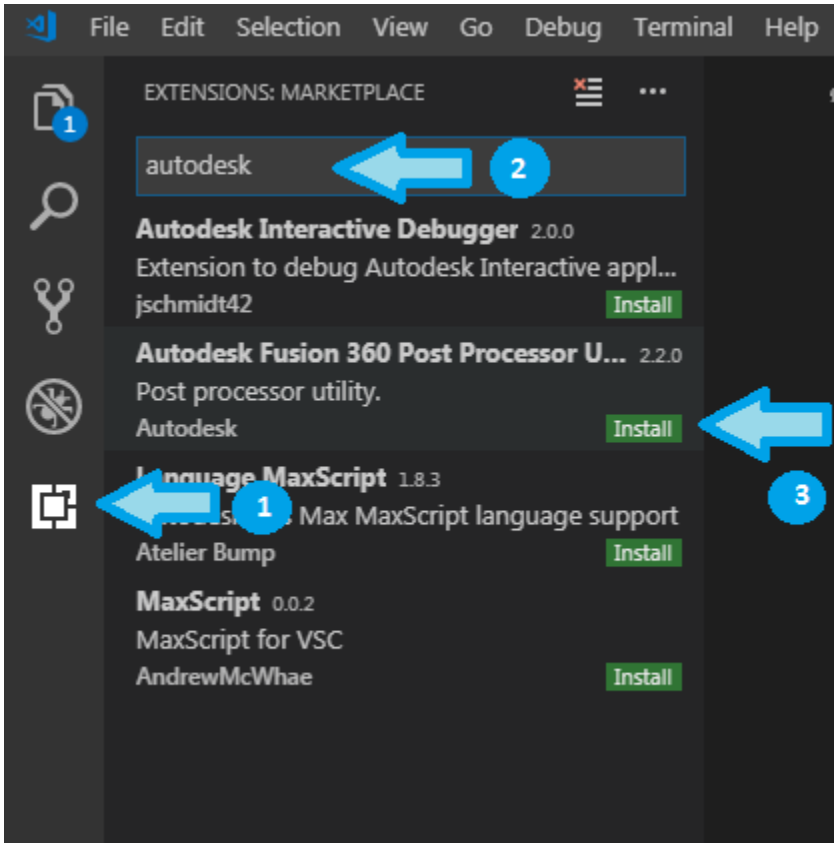
Click the Executable to Install VSC

Follow the instructions displayed on the screen to finish the installation. You should select the defaults for all questions, though you may want to make this the default code editor and add it to the Windows Explorer file context menu.



Selecting Installation Options

You can choose to startup the Visual Studio Code editor automatically after it is installed. Once the editor is opened you can install the Autodesk Fusion 360 Post Processor Utility by opening the *Extensions* view in the left pane and searching for *Autodesk*. Select the Autodesk Fusion 360 Post Processor Utility to install it.



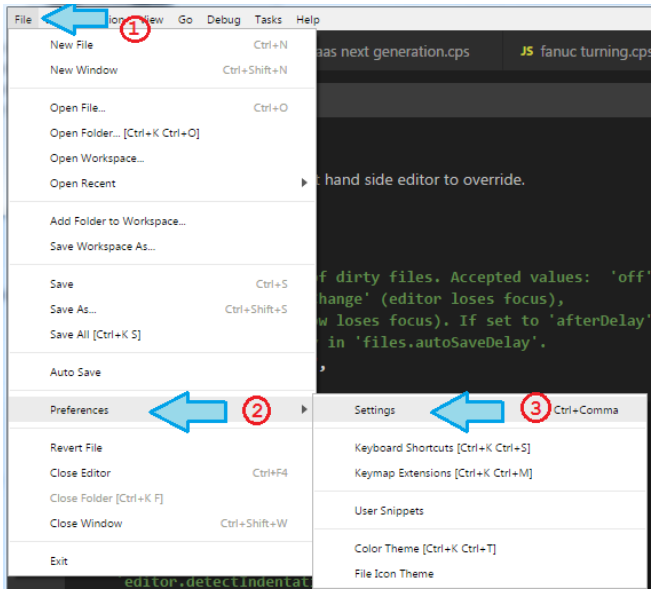
Downloading the Autodesk Fusion 360 Post Processor Extension

Installing the Autodesk Fusion 360 Post Processor Extension

After installing the Autodesk Fusion 360 Post Processor Utility extension you will want to exit the VSC editor and then restart it so that the extension is initialized. You are now ready to start editing Autodesk post processors.

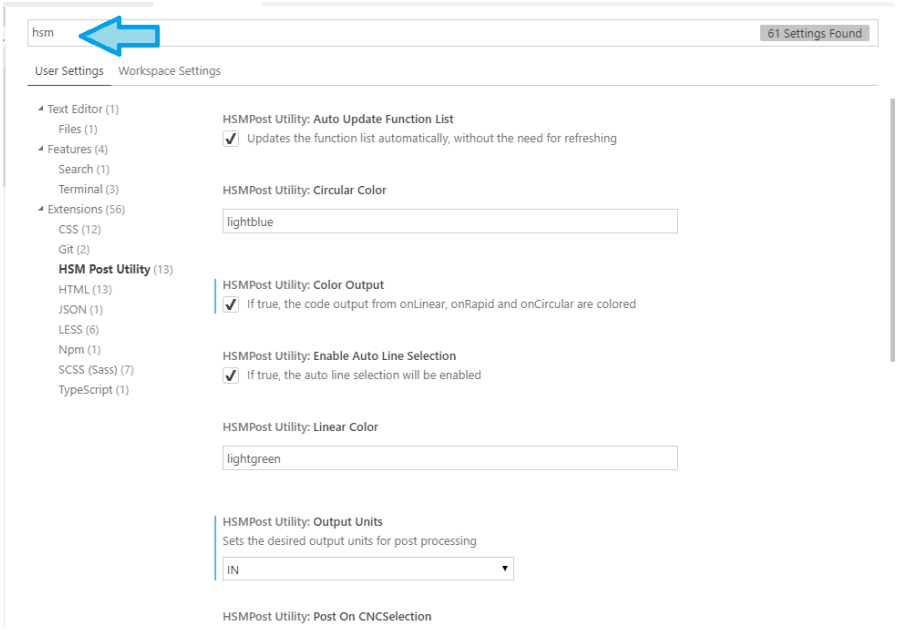
2.2 Autodesk Post Processor Settings

After installing the Autodesk Post Processor editor you will want to setup the editor to match your preferences. Open the settings file by selecting *File->Preferences->Settings*. This section will describe some of the most popular settings, but feel free to explore other settings at your leisure to find any that you may want to change. The User Settings can also be displayed by using the *Ctrl+Comma* shortcut.



Displaying the Editor Settings

The settings will be displayed in a separate tab. You can now search for individual settings using the Search bar. To display the Autodesk Fusion 360 Post Processor Utility settings type in *hsm* in the search bar.



Modifying the Editor Settings

There is a description that explains the setting making it easy for you to make the changes.

The following table provides a list of some of the more common settings and their descriptions.

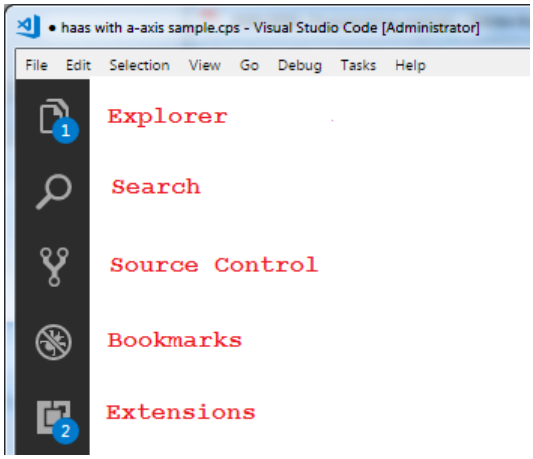
Setting	Description
Editor > Minimap	Controls if the minimap is shown. The minimap is a small representation of the entire file displayed on the right side of the window and allows you to easily scroll through the file.
Editor: Font Size	Size of the editor font.
Editor: Font Weight	Weight (thickness) of the editor font.
Editor: Detect Indentation	Automatically detects the *editor.tabSize* and *editor.insertSpaces* settings when opening a file.
Editor: Insert Spaces	When checked, spaces will be inserted into the file when the *tab* key is pressed.
Editor: Tab Size	Sets the number of spaces a tab is equal to. The standard setting for Autodesk post processors is 2.
Editor > Parameter Hints	Enables a pop-up that shows parameter documentation and style information as you type.
Editor: Auto Closing Brackets	Controls if the editor should automatically close brackets after opening them.
Extensions: Auto Check Update *or* Auto Updates	Automatically (check for) update extensions.
Files: Associations	Associates file types with a programming language. This must have *"*.cps": "javascript"*

Setting	Description
	set in it to enable the automatic features of the editor in Autodesk post processors.
Workbench: Color Theme	Defines the color theme for the editor. This setting can be changed using the *File->Preferences->Color* theme menu.
HSMPost Utility: Auto Update Function List	Updates the function list automatically, without the need for refreshing.
HSMPost Utility: Sort Function List Alphabetically	When checked the function list will be sorted. Unchecked will display the function names in the order that they are defined.
HSMPost Utility: Color Output	When checked, rapid, feedrate, and circular blocks will be displayed in color.
HSMPost Utility: Rapid Color	Color for rapid move blocks.
HSMPost Utility: Linear Color	Color for feedrate move blocks.
HSMPost Utility: Circular Color	Color for circular move blocks.
HSMPost Utility: Enable Auto Line Selection	Enables the automatic selection of the line in the post processor that generated the selected line in the output NC file.
HSMPost Utility: Output Units	Sets the desired output units when post processing
HSMPost Utility: Shorten Output Code	Limits the number of blocks output when posting, making it easier to navigate.
HSMPost Utility: Post On CNCSelection	When checked, post processing will occur as soon as a CNC file is selected.
HSMPost Utility: Post On Save	Automatically run the post processor when it is saved, only if the NC output file window is open.

Commonly Changed User Settings

2.3 Left Side Flyout

On the left side of the editor window is a tab that will open different flyout dialogs. The features contained in the flyout dialogs are quite beneficial while editing a post processor and are explained in this section. The *Source Control* flyout is not used when editing post processors and will not be discussed.



Left Side Flyout Dialog

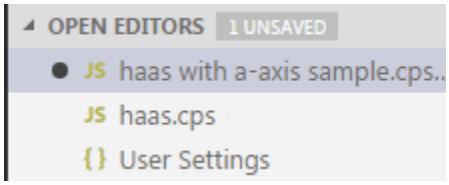
2.3.1 Explorer Flyout

The Explorer flyout contains expandable lists that are used to display the open editors, folders, variables, functions, and CNC selector. The arrow ► at the left of each entry is used to expand or collapse the list.

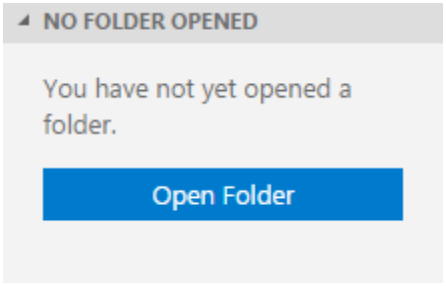
List	Description
OPEN EDITORS	Lists the files that are open in this instance of the VSC editor. Any files that have been changed, but not been saved will be marked with a bullet (•). The number of changed files that have not been saved is displayed in the Explorer icon.
NO FOLDERS OPEN	You can open a folder for quick access to all of the post processors in the folder. Expanding the folders will display the *Open Folder* button that can be used to open a folder. Clicking on a file in the open folder will automatically open it in the editor. Take note that if a folder is opened, then all opened files in the editor will first be closed and you will be prompted to save any that have been changed.
OUTLINE	Lists the functions defined in the post processor and the variables defined in each function. Expanding the function by pressing the arrow ► to the left of the function name will display the variables defined in the function. You can select any of the variables to go to the line where it is defined.

List	Description
CNC SELECTOR	Contains the Autodesk intermediate files (*.cnc) that are available to the post processor from the VSC editor. This list is further explained in the *Running/Debugging the Post* section of this chapter.
FUNCTION LIST	Expanding the function list will display the functions defined in the active post processor. The functions will either be listed in alphabetical order or by the order they appear in the post processor depending on *the HSMPost Utility: Sort Function List Alphabetically* setting. You can select on a function in this list and the cursor will be placed at the beginning of this function in the editor window and while traversing through the post processor the function that the cursor is in will be marked with an arrow ►, making it easy for you to determine what function the active line is in.
POST PROPERTIES	Contains the Property Table for the post processor, similar to the Property Table displayed when running the post from CAM. This list is further explained in the *Running/Debugging the Post* section of this chapter.
VARIABLE LIST	Lists the variable types supported by the post processor, such as Array, Format, Vector, etc. It does not contain a list of variables defined in the post processor. Expanding the variable type by pressing the arrow ►to the left of it will display the functions associated with the variable type.

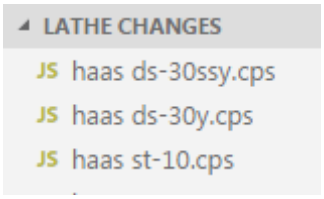
Explore Flyout Selections



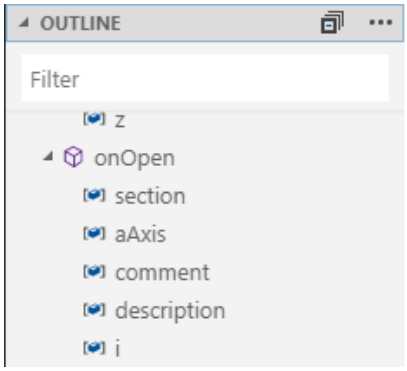
Open Editors



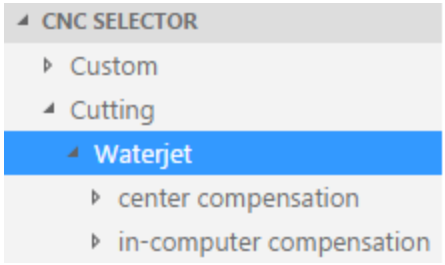
Opening a Folder



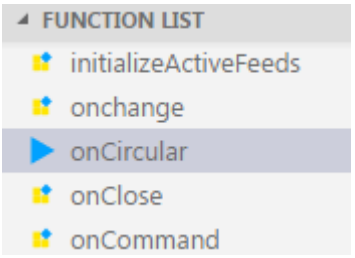
Open Folder File List



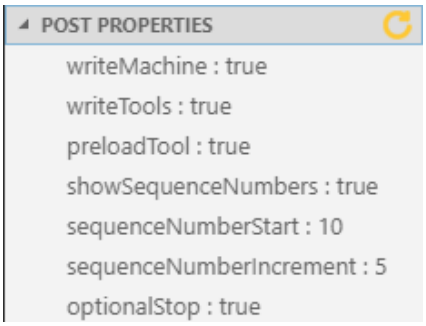
Outline



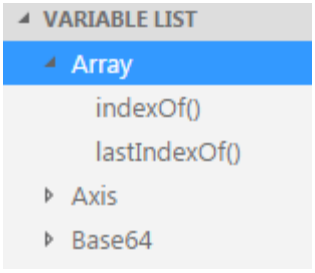
CNC Selector



Function List



Post Properties

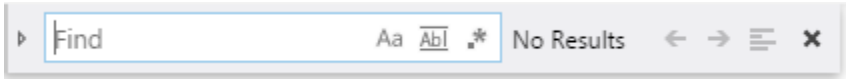


Variable List

2.3.2 Search Flyout

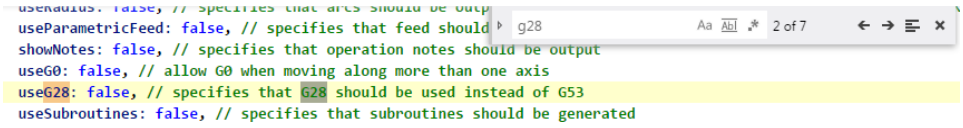
Search

You can search for a text string in the current file or in all of the opened files. To search for the text string in the current file you should use the Find popup window accessed by pressing the *Ctrl+F* keys.



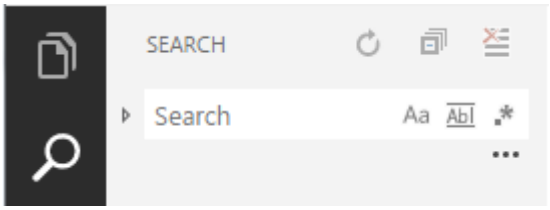
Ctrl+F Find Popup – Search for a Text String in the Current File

As you type in a text string the editor will automatically display and highlight the next occurrence of the text in the file. The number of occurrences of the text string in the file will be displayed to the right of the text field. You can use the *Enter* key to search for the next occurrence of the string or press the arrow keys to search forwards → and backwards ← through the file. If you use the *Enter* key, then the keyboard focus must be in the *Find* field.



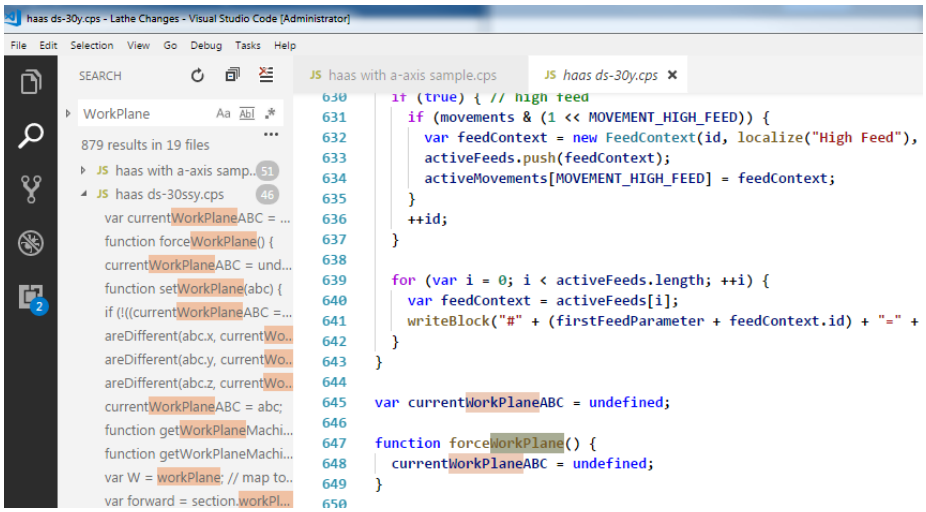
Using the Find Popup to Search for Text Strings

The *Search* flyout searches for a file in the opened files and in the files located in an open folder (refer to the *Explorer* flyout to see how to open a folder). The *Search* dialog will be displayed when you press the *Search* button.



Search Flyout – Search for a Text String in Multiple Files

Entering a text string to search for and then pressing the *Enter* key will display the files that contain the text string and the number of instances of the text string in each file. You can expand the file in the list by pressing the arrow key ► and each instance of the text string found in the selected file will be displayed. Clicking on one of the instances causes the editor to go to that line in the file and automatically open the file if it is not already opened. If you don't make any changes to the file and then select the text string in another file, then the first file will be closed before opening the next file. An unchanged file opened from the *Search* flyout will have its name italicized in the editor window.



Searching for a Text String in the Opened Files

There are options that are available when searching for text strings. These options are controlled using the icons in the *Search* dialog and *Find* popup.

Icon	Description
	When enabled, the case of the search string must be the same as the matching text string in the file.
	When enabled, the entire word of the matching text string in the file must be the same as search string. When disabled, it will search for the occurrence of the search string within words.
	When enabled, the '.' character can be used as a single character wildcard and the '*' character can be used as a multi-character wildcard in the search string.

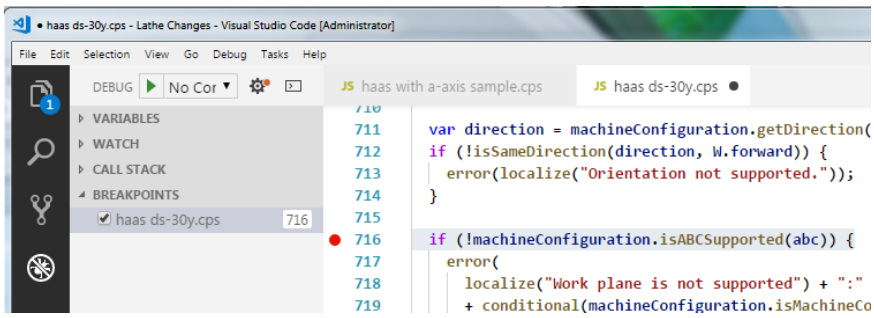
Icon	Description
	Search forward in the file. In the *Find* popup only.
	Search backward in the file. In the *Find* popup only.
	Searches for the text string only in the selected text in the file. In the *Find* popup window only.
	Closes the *Find* popup window.
	Refresh the results window. In the *Search* flyout only.
	Collapse all expanded files in the results window. In the *Search* flyout only.
	Displays fields that allow you to include or exclude certain files from searches. In the *Search* flyout only.
	Displays the *Replace* field, allowing you to replace the *Search* text with the *Replace* field text.
	Replaces the current (highlighted) occurrence of the *Search* text with the *Replace* field text. Hitting the *Enter* key while in the *Replace* field performs the same replacement. In the *Find* popup window only.
	Replaces all occurrences of the *Search* text with the *Replace* field text. When initiated from the *Search* flyout, all occurrences of the text in all files listed in the *Results* window will be replaced.

Search and Replace Options

2.3.3 Bookmarks Flyout

Okay, so the *Bookmarks* flyout is actually a Breakpoints flyout, but since JavaScript does not have an interactive debugger we are going to use it for adding bookmarks to the opened files. Placing the cursor to the left of the line number where you want to set a bookmark will display a red circle and then clicking at this position will add the bookmark.

To see the active bookmarks you can open the Bookmarks flyout and expand the *BreakPoints* window. You can then go directly to a line that is bookmarked by selecting that line in the Bookmarks flyout. Bookmarks set in all opened files will be displayed in the flyout and the file that the bookmark is set in will automatically be made the active window when the bookmark is selected.

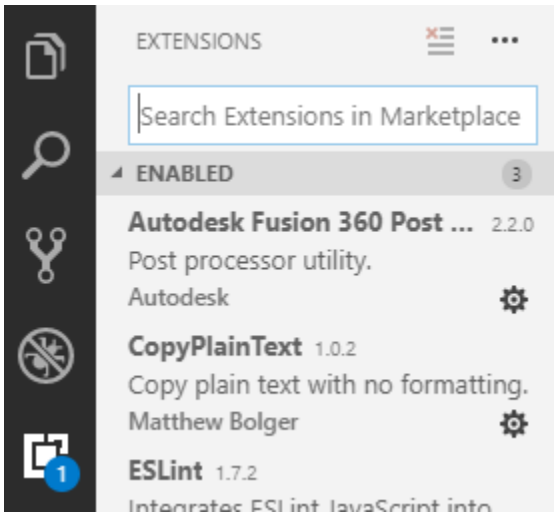


Using the Bookmarks Flyout

2.3.4 Extensions Flyout

Extensions

Visual Studio Code is an open source editor and there are many extensions that have been added to it by the community. For example, the *Autodesk Fustion 360 Post Processor* Utility is an extension to this editor. By opening the Extensions flyout you can see what extensions you have installed and what extensions have updates waiting for them.

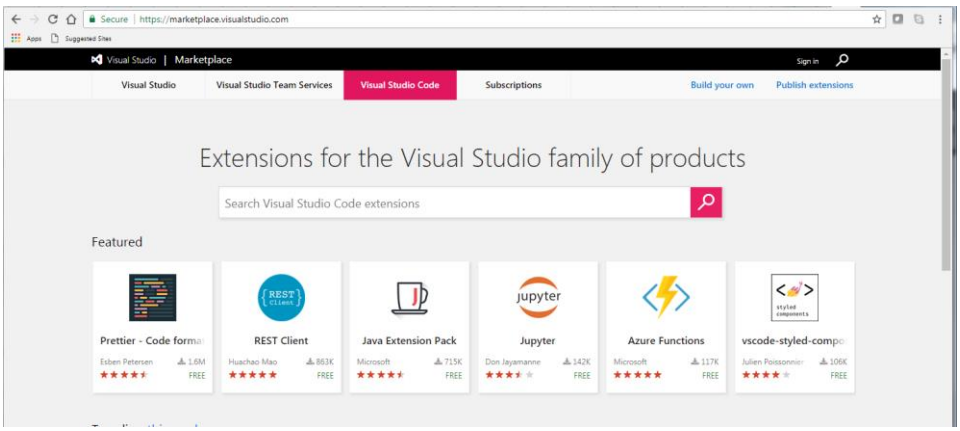


Viewing Installed Extensions

If there is an *Update to x.x.x* button displayed with the extension you can press this button to install the latest version of the associated extension.

You can search the Visual Studio Marketplace for extensions that are beneficial for your editing style by typing in a name in the *Search Extensions in Marketplace* field. For example, if you want a more dedicated way to set bookmarks you can type in bookmark in this field and all extensions dealing with adding bookmarks will be displayed. You can press the green *Install* button to install the extension.

You can also search for extensions online at the Visual Studio Marketplace.



2.4 Autodesk Post Processor Editor Features

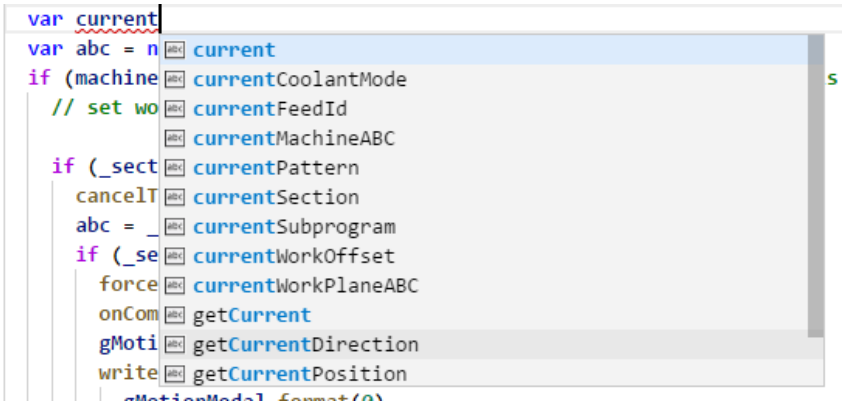
The Autodesk Post Processor editor has features to enhance the ease of editing of post processor JavaScript files. One example is the color coding of the text, variables are in one color, functions in another, JavaScript reserved words in yet another, and so on. The colors of each entity is based on the *Workbench Color Theme* setting.

This section will go over some of the more commonly used features. You are sure to discover other features as you use the editor.

2.4.1 Auto Completion

As you type the name of a variable or function you will notice a popup window that will show you previously used names that match the text as it is typed in. Selecting one of the suggestions by using the arrow keys to highlight the name and then the tab key to select it will insert that name into the spot where you are typing.

If the *Editor: Parameter Hints* setting is set to true, then when you type in the name of a function, including the opening parenthesis, you will be supplied the names of the function's arguments for reference.

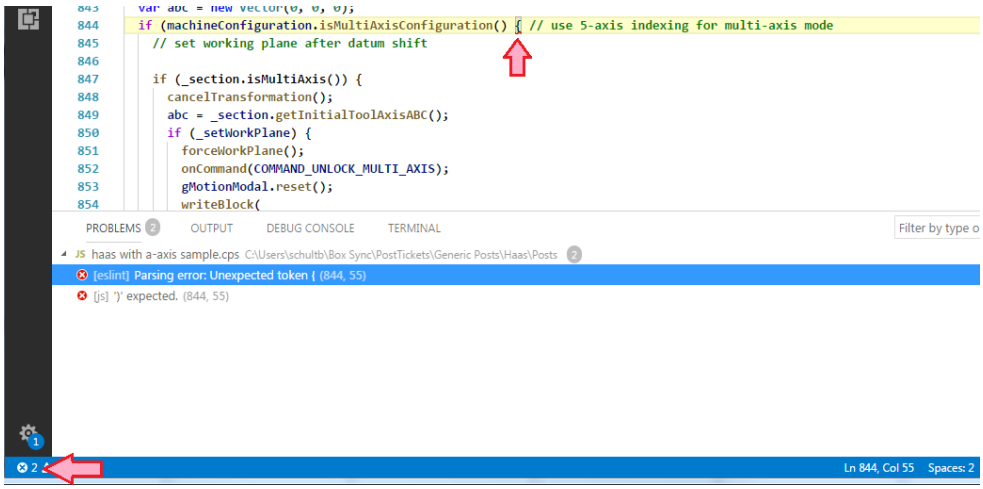


Using Auto Completion

2.4.2 Syntax Checking

If you have a syntax error while editing a file, the editor is smart enough to flag the error by incrementing the error count at the bottom left of the window footer and marking the problem in the file with a red squiggly line. You can open the Problems window by selecting the X in the window footer to see all lines that have a syntax error. Clicking on the line displaying the error will then take you directly to that line, so that you can resolve the error.

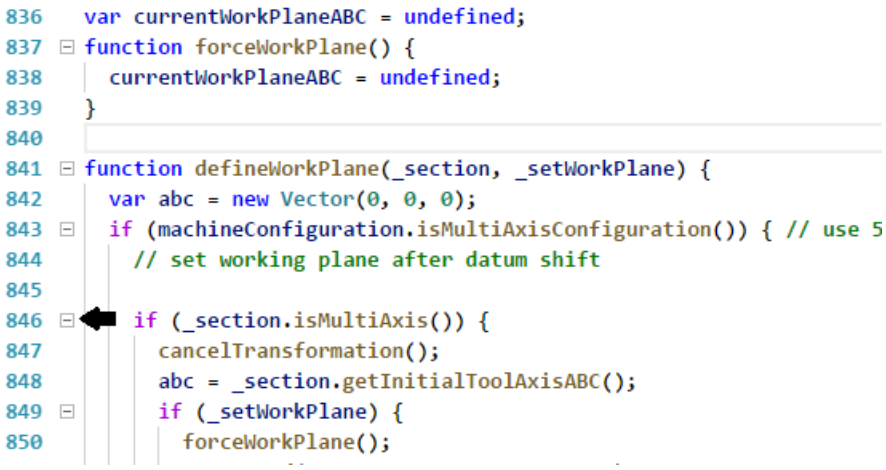
You can close the window by pressing on the X in the window footer or the X at the top right of the Problems window.



Displaying Syntax Errors

2.4.3 Hiding Sections of Code

You can hide code that is enclosed in braces {} by positioning the cursor to the right of the line number on the line with the opening brace and then pressing the [-] icon. The code can be expanded again by pressing the [+] icon. Note that the icons will not be displayed unless the cursor is placed in the area between the line number and the editing window.



Hiding Sections of Code

2.4.4 Matching Brackets

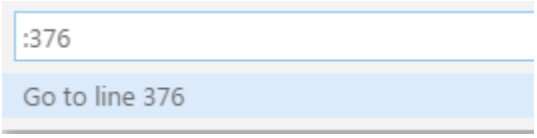
If you place the edit cursor at a parenthesis (()), bracket ([]), or brace ({}) the editor will highlight the selected enclosure as well as the opening/closing matching enclosure character. If there are multiple enclosure characters right next to each other, then the enclosure following the edit cursor will be selected. If the enclosure character does not highlight, then this means that there is not a matching opening/closing enclosure.

```
writeBlock(
  gMotionModal.format(0),
  conditional(machineConfiguration.isMachineCoordinate(0), "A" + abcFormat.format(abc.x)),
  conditional(machineConfiguration.isMachineCoordinate(1), "B" + abcFormat.format(abc.y)),
  conditional(machineConfiguration.isMachineCoordinate(2), "C" + abcFormat.format(abc.z))
);
```

Matching Parenthesis

2.4.5 Go to Line Number

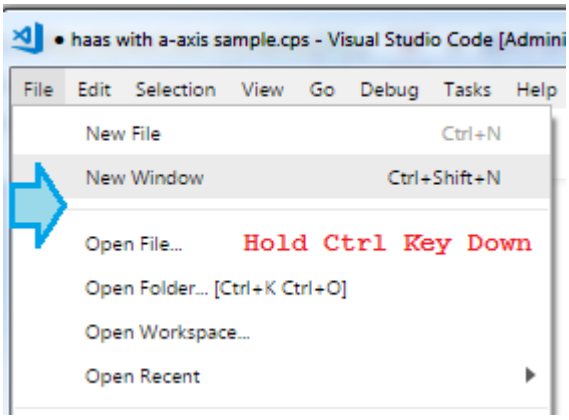
You can go to a specific line number in the file by pressing the *Ctrl+G* keys and then typing in the line number.



Go to Line Number

2.4.6 Opening a File in a Separate Window

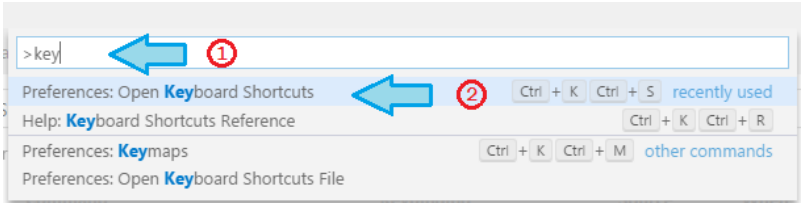
You can open a file in the current window by selecting the *File->Open File…* menu from the task bar or by pressing the *Ctrl+O* keys. You can open the active file in a separate VSC window by pressing the *Ctrl+K* keys and then pressing the *O* key. The file will be opened in the a new window and remain open in the active window. You can also open a new VSC window by selecting the *File->New Window* menu or by pressing the *Ctrl+Shift+N* keys.



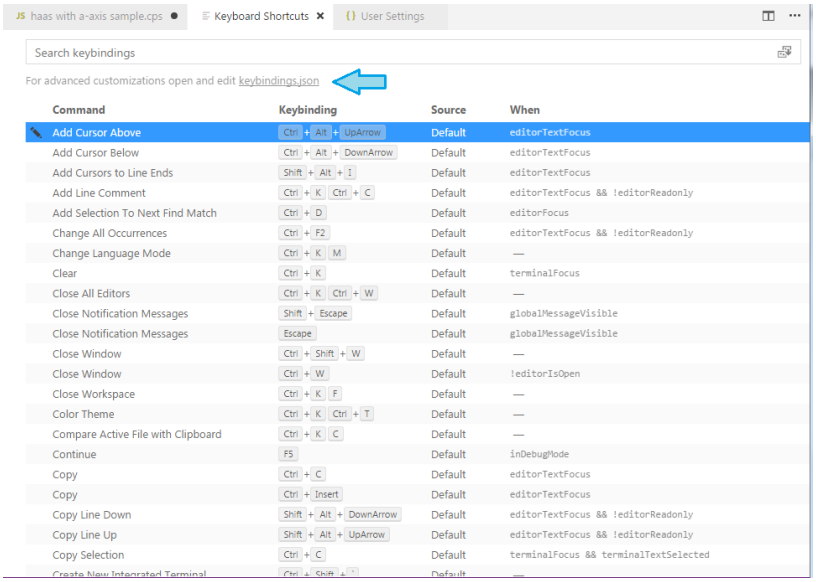
Open Separate VSC Window

2.4.7 Shortcut Keys

You can display the assigned Shortcut Keys by pressing the *F1* key and then typing in *key* to display all commands referencing the key string. Select the *Preferences: Open Keyboard Shortcuts* menu. You can also press the *Ctrl+K Ctrl+S* keys in sequence to display the Shortcut Keys window.



Display the Shortcut Keys



Shortcut Key Assignments

Modifications and/or additions to the Shortcut Key assignments can be made by selecting the *keybindings.json* link at the top of the page. This will open a split window display that displays the default Shortcut Keys in the left window and the user defined Shortcut Keys in the right window. Use the same procedure as modifying a setting to modify a Shortcut Key, by copying the binding definition from the left window into the right window and making the desired changes. Be sure to save the *keybindings.json* file after making your changes.

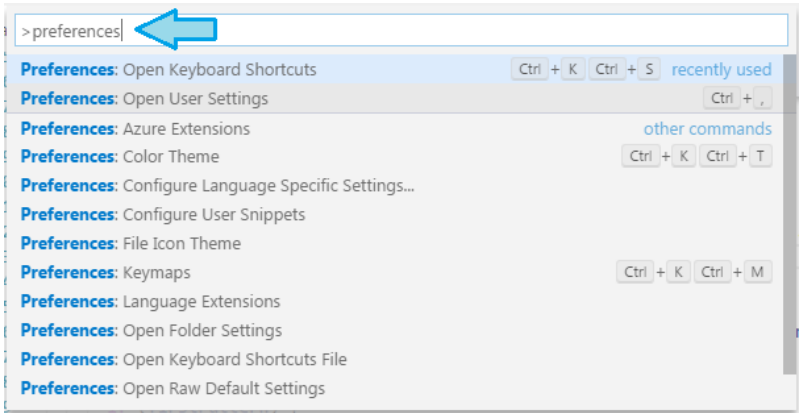
The format of the keystrokes that represent a single Shortcut is defined in the following table.

Shortcut	Sample	Description
key	F1	Press the single key.
key+key	Ctrl+Shift+Enter	*key* is the name of the key to press. The + character means that the keys must be pressed at the same time. The + key is not pressed.
key key	Ctrl+K Ctrl+S	The keys should be pressed in sequence, one after the other. Each key can be a combination of multiple keys that are pressed at the same time as explained above. Unless *Shift* is part of the key sequence, then lower case letters are being specified.

Shorcut Key Syntax

2.4.8 Running Commands

The commands accessible by shortcut keys or the menus can be found and run from the command popup dialog and are accessed in the editor by pressing the *F1* key. Once the command popup is displayed you can search for commands by typing in text in the search line. The commands that match the search will be displayed along with the Shortcut Keys that are assigned to the commands. Select on the command to run it.



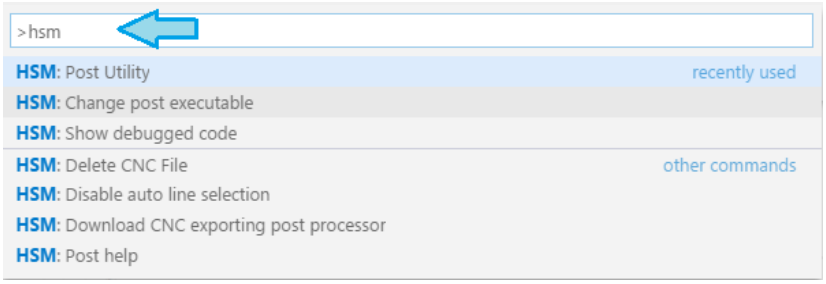
Running a Command

2.5 Running/Debugging the Post

The *Autodesk Fusion 360 Post Processor Utility* extension allows you to run the post processor that you are editing directly from the editor and to debug the post by matching the output lines in the NC file with the code line that generated the output. You can run the post against the provided intermediate files generated from the Benchmark Parts or you can create your own intermediate file to run the post against.

2.5.1 Autodesk Post Processor Commands

There are built-in commands that pertain to running the post processor. These commands are accessed by pressing the F1 key and typing HSM in the search field.



Displaying the Autodesk Post Processor Commands

The following table describes the available commands.

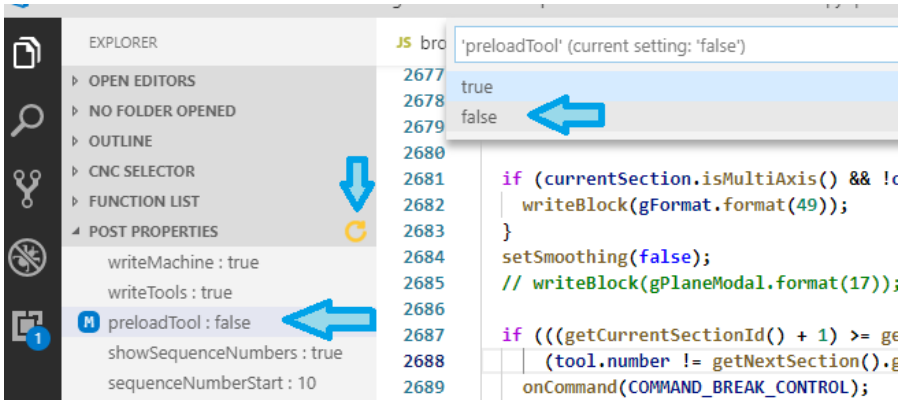
Command	Description
Post Utility	Displays a menu where you can post process the selected intermediate (CNC) file against the open post processor, select a new CNC file, or display the *Autodesk Post Help* window. You can also use the shortcut *Ctrl+Alt+G* to run the post processor.
Change post executable	Sets the location of the post processor engine executable.
Show debugged code	Displays the entry functions that are called and the line numbers that generated the block in the output NC file. This is the same output that is displayed when you call the *setWriteStack(true)* and *setWriteInvocations(true)* functions.
Delete CNC file	This command cannot be run from the Commands menu. Right clicking on a CNC file in the *CNC Selection* list and selecting *Delete CNC File* will delete the file and remove it from the list.
Disable auto line selection	Disables the feature of automatically displaying the line in the post processor that generated the selected line in the NC output file.
Download CNC exporting post processor	Downloads the *Exporting Post Processor* used for generating your own CNC files for testing.
Post help	Displays the online *AutoDesk CAM Post Processor Documentation* web page.

The Autodesk Post Processor Commands

2.5.2 The Post Processor Properties

You can display the properties associated with the open post processor by opening the Explorer flyout and expanding the Post Properties list. Clicking on a property will prompt you to change the property. The  symbol will be displayed next to the property if it has been changed from the default value.

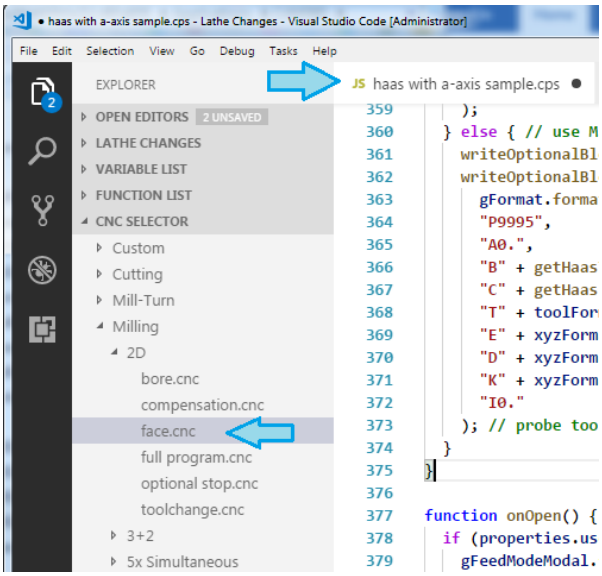
If you add a new property to the post or for some reason the properties don't display, you can press the yellow refresh symbol in the Post Properties header to refresh the displayed properties.



Modifying the Post Properties

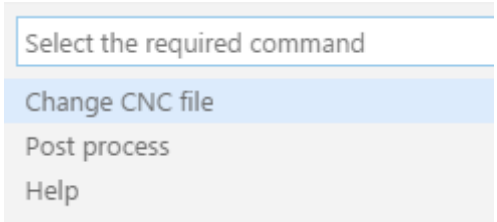
2.5.3 Running the Post Processor

To run the post processor that is open in the editor you can use the *Ctrl+Alt+G* shortcut or run the *Post Utility* from the Command window as described in the previous section. First you will need to select the intermediate CNC file to run the post against. You select the CNC file by opening the *Explorer* flyout and expanding the *CNC Selector* list until you find the desired CNC file.



Post the Selected CNC File Against the Active Post

You can also select the CNC file from the Post Utility menu.



Select the CNC File or Post Processor Using the Post Utility Command

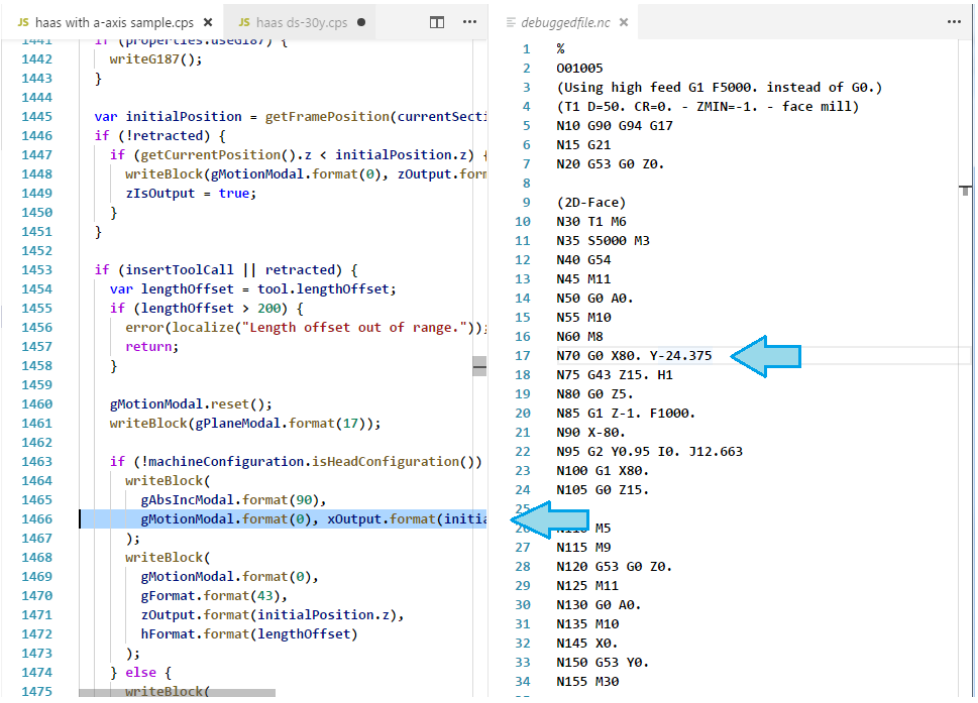
If running a post processor for the first time in the editor it is possible that the location of the post engine executable (post.exe) is not known. In this case you will see the following message displayed.

You can press the *Browse…* button to search for *post.exe*. The executable will be in one of the following locations depending on the version of HSM being run.

HSM Version	Post Executable Location
Fusion	C:\User*username*\AppData\Local\Autodesk\webdeploy\production*(id)*\Applications\CAM360 *username* is your user name that you logged in as. *(id)* is a unique and long name that changes depending on the version of Fusion that you have installed. You will usually select the folder with the latest date.
Inventor	C:\Program Files\Autodesk\InventorHSM *yyyy* *yyyy is* the version number (year) of Inventor.
HSMWorks	C:\Program Files\HSMWorks

Post Executable Locations

Once you have posted against the CNC file, the output NC file or Log file will be displayed in the right panel of the split screen. When the *HSMPostUtility: Enable Auto Line Selection* setting is true, then clicking twice on a line in the output NC file will highlight the line in the post processor that generated the output. The second click must be on a different character on the same output line to highlight the line. Then, by clicking on a different character in the same line you will be walked through the stack of functions that were called in the generation of the output.

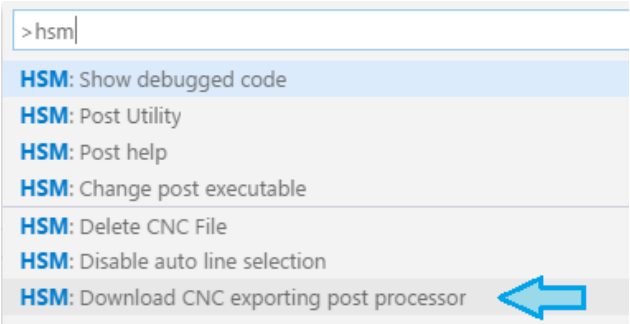


Output NC File, Click Twice on Output Line to See Code that Generated Output

2.5.4 Creating Your Own CNC Intermediate Files

The Autodesk Post Processor extension comes with built-in CNC intermediate files that are generated using the HSM Benchmark Parts. These can be used for testing most aspects of the post processor, but there are times when you will need to test specific scenarios. For these cases you can create your own CNC file to use as input.

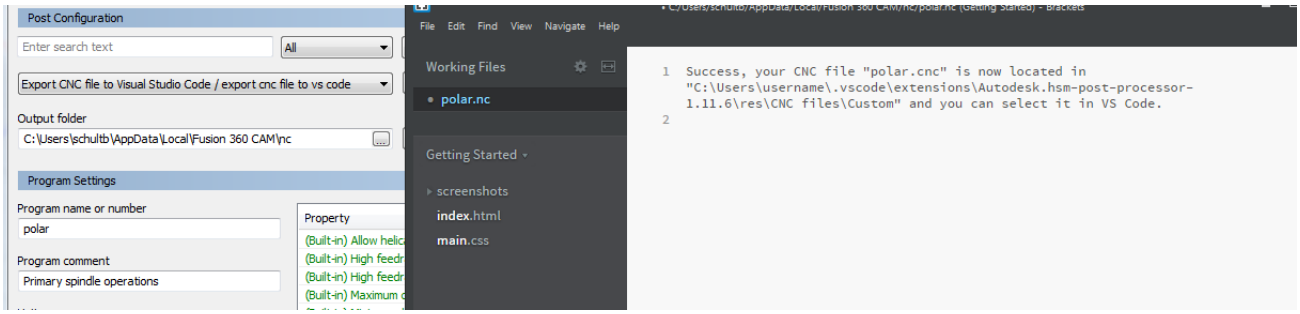
First you will need to download the *export cnc file to vs code.cps* post processor. You can do this by running the *Download CNC exporting post processor* command.



Download the CNC Exporting Post Processor

A file browser will come up that allows you to select the folder where you want to download the post. Follow the directions in the *Downloading and Installing a Post Processor* section for installing a post processor on your system.

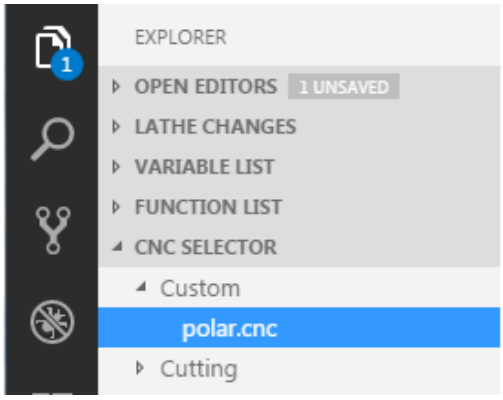
Once the post processor is installed you will want to post process the operations you want to use for testing. The CNC exporting post processor is run just like any other Autodesk post processor, except it will not generate NC code, but will rather create a copy of the CNC file from the Autodesk CAM system in the *Custom* location of the *CNC Selector* folder. Most posts use a number for the output file name, it is recommended that you give the CNC file a unique name that describes the operations that were used to generate it.



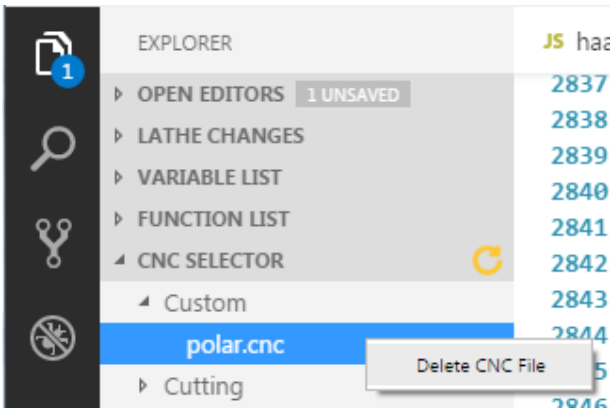
Create a Custom CNC Intermediate File

Once you click the yellow refresh button you should see the CNC file in the *Custom* branch of the *CNC Selector* list and can use it when post processing from the VSC editor. If you decide that you no longer

need a custom CNC intermediate file you can delete it by right clicking on the CNC file and select *Delete CNC File*.



Using a Custom CNC Intermediate File



Deleting a Custom CNC Intermediate File

3 JavaScript Overview

3.1 Overview

Autodesk post processors are written using the JavaScript language. It resembles the C, C++, and Java programming languages, is interpreted rather than being a compiled language, and is object-orientated. JavaScript as it is used for developing post processors is fairly simple to learn and understand, but still retains its complex nature for more advanced programmers.

This chapter covers the basics of the JavaScript language and conventions used by Autodesk post processors. There are many web sites that document the JavaScript language. The ELOQUENT JAVASCRIPT site has a nicely laid out format. If you prefer a hard copy JavaScript guide, then the *JavaScript the Definitive Guide, Author: David Flanagan, Publisher: O'Reilly* is recommended. Whichever manual you use, you will want to focus on the core syntax of JavaScript and ignore the browser and client-side aspects of the language.

The Autodesk post processor documentation is provided as the *post.chm* file with HSMWorks and Inventor HSM or you can visit the Autodesk CAM Post Processor Documentation web site. You will find that the *post.chm* version of the documentation is easier to view, since it has a working Index.

3.2 JavaScript Syntax

JavaScript is a case sensitive language, meaning that all functions, variables, and any other identifiers must always be typed exactly the same with regards to lower and uppercase letters.

```
currentCoolant = 7;
currentCoolant = 8;
currentcoolant = 9;
```

Case Sensitive Definition of 3 Different Variables

JavaScript ignores spaces and new lines between variables, operators, names, and delimiting characters. Variable and function names cannot have spaces in them, as this would create separate entities. Commands can be continued onto multiple lines and are terminated with a semicolon (;) to mark the end of the logical command. If you are defining a string literal within quotes, then the literal should be defined on a single line and not on multiple lines. If a text string is too long for a single line, then it should be concatenated using an operation.

```
message = "The 3 inch bore needs to be probed prior to starting " +
  "the next operation.";
```

Breaking Up a Text String onto Multiple Lines

There are two methods of defining comments in JavaScript. You can either enclose comments between the /* and */ characters, which will treat all text between these delimiters as a comment, or place the // characters prior to the comment text.

The /* *comment* */ format is typically used as the descriptive header of a function or to block out multiple lines of code. Any characters on the line that follow the // characters are treated as a comment, so you can have a single comment line or add a comment to the end of a JavaScript statement.

```
/**
  Output a comment.
*/
function writeComment(text) {
  writeln(formatComment(text)); // write out comment line
}
..
/*
  switch (unit) {
  case IN:
    writeBlock(gUnitModal.format(20));
    break;
  case MM:
    writeBlock(gUnitModal.format(21));
    break;
  }
*/
```

Comment Lines

Using indentation for function contents, if blocks, loops and continuation lines is recommended as this makes it easier to visualize the code. Tab characters, though supported by JavaScript, are discouraged from being used. It is preferred to use virtual tab stops of two spaces for indenting code in post processor code. Most editors, including the Autodesk Post Processor Editor can be setup to automatically convert tab characters to spaces that will align each indent at two spaces. Please refer to the Post Processor Editor chapter for an explanation on how to setup the Autodesk recommended editor.

```
function test (input) {
```

```
  // indent 2 spaces inside of function
  if (input == 1) {
    writeBlock(  // indent 2 more spaces in if block or loop
      gAbsIncModal.format(90),  // indent 2 more spaces for continuation lines
      gMotionModal.format(0)
    );
  }
}
```

Indent Code 2 Spaces Inside Function, If Block, Loop, and Continuation Line

3.3 Variables

Variables are simply names associated with a value. The value can be a number, string, boolean, array, or object. Variables in JavaScript are untyped, meaning that they are defined by the value that they have assigned to them and the value type can change throughout the program. For example, you can assign a number to a variable and later in the program you can assign the same variable a string value. The *var* keyword is used to define a variable.

If a variable is not assigned a value, then it will be assigned the special value of *undefined*.

```
var a;              // define variable 'a', it will have the value of undefined
var b = 1;          // assign a value of 1 to the variable 'b'
var c = "text";     // assign a text string to the variable 'c'
c = 2.5;            // 'c' now contains a number instead of string
```

Variable Definitions

While you can include multiple variable declarations on the same *var* line, this is against the standard used for post processors and is not recommended. You can also implicitly create a variable simply by assigning a value to the variable name without using the *var* keyword, but is also not recommended. When declaring a new variable, be sure to not use the same name as a JavaScript or Post Kernel keyword, for example do not name it *var*, *for*, *cycle*, *currentSection*, etc. Refer to the appropriate documentation for a list of keywords/variables allocated in JavaScript or the Post Kernel.

JavaScript supports both global variables and local variables. A global variable is defined outside the scope of a function, for example at the top of the file prior to defining any functions. Global variables are accessible to all functions within the program and will have the same value from function to function. Local variables are only accessible from within the function that they are defined. You can use the same name for local variables in multiple functions and they will each have their own unique value in the separate functions. Unlike the C and C++ languages, local variables defined within an if block or loop are accessible to the entire function and are not local to the block that they are defined in.

3.3.1 Numbers

Besides containing a standard numeric value, a variable assigned to a number creates a *Number* object. For this discussion, we will consider an object a variable with associated functions. These functions are specific to numbers and are listed in the following table.

Function	Description	Returns
toExponential(digits)	Format a number using exponential notation	String representation of number
toFixed(digits)	Format a number with a fixed number of digits	String representation of number
toLocaleString()	Format a number according to locale conventions	String representation of number
toPrecision(digits)	Format a number using either a fixed number of digits or using exponential notation depending on value of number	String representation of number
toString()	Format a number	String representation of number

Number Object Functions

```
var a = 12.12345;
b = a.toExponential(2);    // b = "1.21e+1"
b = a.toFixed(3);          // b = "12.123"
b = a.toString();          // b = "12.12345"
```

Sample Number Output

The JavaScript built-in Math object contains functions and constants that apply to numbers. The following table lists the Math functions and constants that are most likely to be used in a post processor. All Math functions return a value.

Function	Return value
Math.abs(x)	Absolute value of *x*
Math.acos(x)	Arc cosine of *x* in radians
Math.asin(x)	Arc sine of *x* in radians
Math.atan(x)	Arc tangent of *x* in radians
Math.atan2(y, x)	Counterclockwise angle between the positive X-axis and the point *x,y* in radians
Math.ceil(x)	Rounds up *x* to the next integer
Math.cos(x)	Cosine of *x*
Math.floor(x)	Rounds down *x* to the next integer
Math.max(args)	The maximum value of the input arguments
Math.min(args)	The minimum value of the input arguments
Math.PI	The value of PI, approximately 3.14159
Math.pow(x, y)	*x* raised to the power of *y*
Math.round(x)	Rounds *x* to the nearest integer
Math.sin(x)	Sine of *x*
Math.sqrt(x)	Square root of *x*
Math.tan(x)	Tangent of *x*
Math.NaN	The value corresponding to the not-a-number property

Math Object

```
a = Math.sqrt(4);          // a = 2
a = Math.round(4.59);      // a = 5
```

```
a = Math.floor(4.59);          // a = 4
a = Math.PI;                   // a = 3.14159
a = Math.cos(toRad(45));       // a = .7071
a = toDeg(Math.acos(.866));    // a = 60
```

Sample Math Object Output

The Math trigonometric functions all work in radians. As a matter of fact, most functions that pass angles in the post processor work in radians. There are kernel supplied functions that are available for converting between radians and degrees. *toDeg(x)* returns the degree equivalent of the radian value *x* and conversely the *toRad(x)* function returns the radian equivalent of the degree value *x*.

3.3.2 Strings

Variables assigned a text string will create a *String* object, which contain a full complement of functions that can be used to manipulate the string. These functions are specific to strings and are listed in the following table. The table details the basic usage of these functions as you would use them in a post processor. Some of the functions accept a *RegExp* object which is not covered in this manual, please refer to dedicated JavaScript manual for a description of this object.

Function	Description	Returns
charAt(n)	Returns a single character at position *n*	The *nth* character
indexOf(substring, start)	Finds the *substring* within the string. *start* is optional and specifies the starting location within the string to start the search at.	The location of the first occurrence of *substring* within the string.
lastIndexOf(substring, start)	Finds the last occurrence of *substring* within the string. *start* is optional and specifies the starting location within the string to start the search at.	The location of the last occurrence of *substring* within the string.
length	Returns the length of the string. *length* is not a function, but rather a property of a string and does not use () in its syntax.	The length of the string
localeCompare(target)	Compares the string with *target* string.	A negative number if *string* is less than *target*, 0 if the strings are identical, and a positive number if *string* is greater than *target*
replace(pattern, replacement)	Replaces the *pattern* text within the string with the *replacement* text.	The updated string.
slice(start, end)	Creates a substring from the string consisting of the *start* character up to, but not including the *end* character of the string.	A substring containing the text from *string* starting at *start* and ending at *end*-1. A negative value for *start* or *end* specifies a position from the end of the *string*; -1 is the last character, -2 is the second to last character, etc.
split(delimiter, limit)	Splits a string at each occurrence of the *delimiter* string.	An array of strings created by splitting *string* into substrings at the delimiter. A maximum of *limit* substrings will be created.
toLocaleLowerCase()	Converts the string to all lowercase letters in a locale-specific method.	Lowercase string.

Function	Description	Returns
toLocaleUpperCase()	Converts the string to all uppercase letters in a locale-specific method.	Uppercase string.
toLowerCase()	Converts the string to all lowercase letters.	Lowercase string.
toUpperCase()	Converts the string to all uppercase letters.	Uppercase string.

String Object Functions

```
var a = "First, Second, Third";
b = a.charAt(3);                    // b = "s"
b = a.indexOf("Second");            // b = 7
b = a.length;                       // b = 20
b = a.localeCompare("ABC");         // b = 5;
b = a.replace(/,/g, "-");           // b = "First- Second- Third"
b = a.slice(0, -7);                 // b = "First, Second"

b = a.split(",");                   // b[0] = "First", b[1] = "Second", b[2] = "Third";
b = a.toLowerCase() ;               // b = "first, second, third"
b = a.toUpperCase();                // b = "FIRST, SECOND, THIRD"
```

Sample String Output

3.3.3 Booleans

Booleans are the simplest of the variable types. They contain a value of either *true* of *false*, which are JavaScript keywords.

```
var a = true; // 'a' is defined as a boolean
if (a) {
  // processes the code in this if block since 'a' is 'true'
}
```

Sample Boolean Assignment

3.3.4 Arrays

An array is a composite data type that stores values in consecutive order. Each value stored in the array is considered an element of the array and the position within an array is called an index. Each element of an array can be any variable type and each element can have a different variable type than the other elements in the array.

An array, like numbers and strings, are considered an object with functions associated with it. You can define an array using two different methods, as an empty array using a new *Array* object, or by creating an array literal with defined values for the array. You can specify the initial size of the array when defining an *Array* object. The initial size of an array defined with values is the number of values contained in the initialization.

```
var a = new Array();                    // creates a blank array, all values are assigned undefined
```

```
var a = new Array(10);                          // creates a blank array with 10 elements
var a = [true, "a", 3.17];                      // creates an array with the first 3 elements assigned
var a = [{x:1, y:2}, {x:3, y:4}, {x:5, y:6}];  // creates an array of 3 xy objects
```

Array Definitions

You can access an array element by using the [ ] brackets. The name of the array will appear to the left of the brackets and the index to the element within the array inside of the brackets. The index can be a simple number or an equation.

```
var a = [1, 2, "text", false];
b = a[0];          // b = 1
a[5] = "next";   // a = [1, 2, "text", false, "next"]
b = a[2+a[0]];  // b = false;
```

Accessing Elements Within an Array

The *Array* object has the following functions associated with it.

Function	Description	Returns
concat(values)	Appends the values to an array.	Original array with concatenated elements
join(separator)	Combines all elements of an array into a string. *separator* is optional and specifies the string used to separate the elements of the array. The default is a comma.	String containing array elements.
length	Returns the allocated size of the array. *length* is not a function, but rather a property of an array and does not use () in its syntax.	The size of the array.
pop()	Pops the last element from the array and decreases the size of the array by 1.	The value of the last element of the array.
push(values)	Pushes the *values* onto the array and increases the size of the array by the number of *values*.	Updated size of array.
reverse()	Reverses the order of the elements of the array.	Returns nothing, but rather modifies the original array.
shift(values)	Removes the first element from the array and decreases the size of the array by 1.	The value of the first element of the array.
slice(start, end)	Creates a new array consisting of the *start* element up to, but not including the *end* element of the array.	An array containing the elements from *array* starting at *start* and ending at *end*-1. A negative value for *start* or *end* specifies a position from the end of the *array*; -1 is the last element, -2 is the second to last element, etc.
sort(function)	Sorts the elements of the array. The original array will be modified. The sort method uses an alphabetical order of elements converted to strings by default. You can specify a function	The sorted array.

Function	Description	Returns
	that overrides the default sorting algorithm.	
toLocaleString()	Format an array according to locale conventions	String representation of array
toString	Format an array	String representation of array
unshift()	Adds the *values* to the beginning of an array and increases the size of the array by the number of *values*.	Updated size of array.

Array Object Functions

```
var a = [1, 2, 3, 4, 5, 6, 7, 8];
b = a.concat(9, 10, 11);    // b = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
b = a.join(", ");           // b = "1, 2, 3, 4, 5, 6, 7, 8"
b = a.length;               // b = 8
a.push(9, 10, 11)           // a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
b = a.pop();                // a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], b = 10
a.reverse();                // a = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
b = a.unshift(12, 11);      // a = [12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1], b = 12
b = a.shift();              // a = [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1], b = 12
b = a.slice(4, 7);          // b = [7, 6, 5]
a.sort(function(a, b) {     // a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
   return a-b;
  });
b = a.toString()            // b = "1,2,3,4,5,6,7,8,9,10,11"
```

Sample Array Output

3.3.5 Objects

An *Object* is similar to an array in that it stores multiple values within a single variable. The difference is that objects use a name for each sub-entity rather than relying on an index pointer into an array. The *properties* table in a post processor is an object. You can define an object using two different methods, explicitly using the *Object* keyword, or implicitly by creating an object literal with defined names and values for the object. Each named entity within an object can be any type of variable, number, string, array, boolean, and another object. Objects can also be stored in an array.

Objects can be expanded to include additional named elements at any time and are not limited to the named elements when they are created.

```
var a = new Object();                          // creates a blank object, without named elements
var a = {x:1, y:2, z:3};                       // creates an object for storing coordinates
a.feed = 10.0;                                 // adds the 'feed' element to the 'a' Object
var a = [{x:1, y:2}, {x:3, y:4}, {x:5, y:6}]; // creates an array of 3 xy objects
```

Object Definitions

3.3.6 The Vector Object

The *Vector* object is built-in to the post processor and is used to store and work with vectors. The vector components are stored in the *x, y, z* elements of the *Vector* object. Certain post processor variables are stored as vectors and some functions require vectors as input. A *Vector* object is created in the same manner as any other object. *Vector* objects are typically used to store and work on vectors, spatial points, and rotary angles.

```
var a = new Vector();                    // creates a blank Vector object
var a = new Vector(1, 0, 0);             // creates an X-axis vector {x:1, y:0, z:0}
a.x = -1;                                // assigns -1 to the x element of the vector
setWorkPlane(new Vector(0, 0, 0));       // defines a null vector inline
```

Sample Vector Definitions

The following tables describe the attributes and functions contained in the *Vector* object. Since an attribute is simply a value contained in the *Vector* object, it does not have an argument.

Attribute	Description
abs	Contains the absolute coordinates of the vector
length	Contains the length of the vector
length2	
negated	Contains the negated vector
normalized	Contains the normalized/unit vector
x	Contains the X-component
y	Contains the Y component
z	Contains the Z component

Vector Attributes

You can directly modify an attribute of a vector, but if you do then the remaining attributes will not be updated. For example, if you directly store a value in the *x* attribute, *vec.x = .707*, the *length* attribute of the vector will not be updated. You should use the *vec.setX(.707)* method instead.

If the Returns column in the following table has *Implicit*, then there is no return value, rather the *Vector* object associated with the function is modified implicitly. For this reason, if you are going to nest a *Vector* function within an expression, do not use the Implicit function, but rather the equivalent function that returns a vector.

Function	Description	Returns
divide(value)	Divides each component of the object vector by the value	Implicit
getCoordinate(coordinate)	Returns the value of the vector component (0=x, 1=y, 2=z)	Component of vector
getMaximum()	Determines the largest component value in the vector	Maximum component value
getMinimum()	Determines the minimum component value in the vector	Minimum component value
getNegated()	Calculates the negated vector	Vector at 180 degrees to the object vector (vector * -1)

Function	Description	Returns
getNormalized()	Calculates the normalized/unit vector	Normalized or unit vector
getX()	Returns the X-coordinate of the vector	X-coordinate
getXYAngle()	Calculates the angle of the vector in the XY-plane	Angle of vector in XY-plane
getY()	Returns the Y-coordinate of the vector	Y-coordinate
getZ()	Returns the Z-coordinate of the vector	Z-coordinate
getZAngle()	Calculates the Z-angle of the vector relative to the XY-plane	Z-angle of vector relative to the XY-plane
isZero()	Determines if the vector is a null vector (0,0,0)	True if it is a null vector
multiply(value)	Multiplies each component of the vector by the value	Implicit
negate()	Multiplies each component of the vector by -1. Creates a vector at 180 degrees to the object vector	Implicit
setCoordinate(coordinate, value)	Sets the value of the vector component (0=x, 1=y, 2=z)	Implicit
setX()	Sets the X-coordinate of the vector	Implicit
setY()	Sets the Y-coordinate of the vector	Implicit
setZ()	Sets the Z-coordinate of the vector	Implicit
toDeg()	Converts radians to degrees	Angles in degrees
toRad()	Converts degrees to radians	Angles in radians
toString()	Formats the vector as a string, e.g. "(1, 2, 3)"	String representation of vector

Vector Object Functions

Static functions do not require an associated Vector object.

Function	Description	Returns
Vector.cross(left, right)	Calculates the cross product of two vectors	Vector perpendicular to the two vectors
Vector.diff(left, right)	Calculates the difference between two vectors	Left vector minus right vector
Vector.dot(left, right)	Calculates the dot product of the two vectors	Cosine of angle between the two vectors
Vector.getAbsolute()	Converts the vector components to absolute values	Vector with absolute coordinates
Vector.getAngle()	Calculates the angle between two vectors	Angle between the two vectors in radians
Vector.getDistance(left, right)	Calculates the distance between two vectors. Typically used when the vectors store XYZ spatial coordinates rather than vectors.	Distance between two points
Vector.getDistance2(left,right)	Calculates the square of the distance between two vectors.	Squared distance between two points.
Vector.lerp(left, right, u)	Calculates a point at a percentage of the distance between the two coordinates. 'u' specifies the percentage of the distance to create the point at.	Point at a percentage of the line between two points

Function	Description	Returns
Vector.product(vector, value)	Multiplies each component of the vector by the value	Vector * value
Vector.sum(left, right)	Adds the two vectors	Left vector plus right vector

Static Vector Functions

```
b = a.length();                                  // b = length of Vector a
c = Vector.getAngle(a, b)                        // c = angle in radians between vectors a and b
var a = new Vector(1, 2, 1.5);
d = a.getMaximum();                              // d = 2
b = Vector.getDistance(point1, point2).normalized; // b = directional vector from point1 to point2
b = Vector.dot(vector1, vector2);                // b = cosine of angle between vector1 & vector2
b = a.negated;                                   // b = vector at 180 degrees to  Vector a
```

Sample Vector Expressions

3.3.7 The Matrix Object

The *Matrix* object is built-in to the post processor and is used to store and work with matrices. Matrices are normally used when working with multi-axis machines, for 3+2 operations and for adjusting the coordinates for table rotations. Matrices in the post processor contain only the rotations for each axis and do not contain translation values.

Certain post processor variables are stored as matrices, such as the *workPlane* variable, and some functions require matrices as input. A *Matrix* object has functions that can be used when creating the matrix and are not dependent on working with an existing matrix.

Assignment Function	Definition
Matrix()	Identity matrix (1,0,0, 0,1,0, 0,0,1)
Matrix(i1, j1, k1, i2, j2, k2, i3, j3, k3)	Canonical matrix
Matrix(scale)	Scale matrix
Matrix(right, up, forward)	Matrix using 3 vectors
Matrix(vector, angle)	Rotation matrix around the vector

Matrix Assignment Functions

```
var a = new Matrix();                            // creates an identity matrix
var a = new Vector(-1, 0, 0, 0,-1,0, 0,0, 1);    // creates a matrix rotated 180 degrees in the XY-plane
var a = new Matrix(.5);                          // creates a half scale matrix
var a = new Matrix(new Vector(1, 0, 0), 30);     // creates an X-rotation matrix of 30 degrees
```

Sample Matrix Definitions

The following tables describe the attributes and functions contained in the *Matrix* object. Since an attribute is simply a value contained in the Matrix object, it does not have an argument.

Attribute	Description
forward	Contains the forward vector
n1	Contains he length of the row vectors of this matrix
n2	Contains the square root of this matrix vector lengths
Negated	Contains the negated matrix
right	Contains the right vector
transposed	Contains the inverse matrix
up	Contains the up vector

Matrix Attributes

You can directly modify an attribute of a matrix, but if you do then the remaining attributes will not be updated. For example, if you directly store a vector in the *forward* attribute, the other attributes will not be updated to reflect this modification. You should use the *matrix.setForward(vector)* method instead.

If the Returns column in the following table has *Implicit*, then there is no return value, rather the *Matrix* object associated with the function is modified implicitly. For this reason, if you are going to nest a *Matrix* function within an expression, do not use the Implicit function, but rather the equivalent function that returns a matrix.

Function	Description	Returns
add(matrix)	Adds the specified matrix to this matrix	Implicit
getColumn(column)	Retrieves the matrix column as a vector	Vector containing the specified column of this matrix
getElement(row, column)	Retrieves the matrix element as a value	Value of this matrix element
getEuler2(convention)	Calculates the angles for the specified Euler convention	Vector containing Euler angles of this matrix. Refer to the *Work Plane* section of the manual for a description of Euler conventions.
getForward()	Returns the forward vector. This will be 0,0,1 in an identity matrix	Forward vector of this matrix
getN1()	Returns the length of the row vectors of this matrix	Returns right_vector + up_vector + forward_vector of matrix
getN2()	Returns the square root of this matrix vector lengths	Math.sqrt(n1)
getNegated()	Calculates the negated matrix	Matrix * -1.
getRight()	Returns the right vector. This will be 1,0,0 in an identity matrix	Right vector of matrix
getRow(row)	Retrieves the matrix row as a vector	Vector containing the specified row of this matrix
getTiltAndTilt(first, second)	Calculates the X & Y rotations around the fixed frame to match the forward direction. 'first' and 'second' can be 0 or 1 and must be different.	Calculated forward direction of this matrix
getTransposed()	Returns the transposed (inverse) of the matrix	Inversed matrix
getTurnAndTilt(first, second)	Calculates the X, Y, Z rotations around the fixed frame to match the	Calculated forward direction

Function	Description	Returns
	forward direction. 'first' and 'second' can be 0, 1, or 2 and must be different.	
getUp()	Returns the up vector. This will be 0,1,0 in an identity matrix	Right vector of matrix
isIdentity()	Determines if the matrix is an identity matrix (1,0,0, 0,1,0, 0,0,1).	True if it is an identity matrix
isZero()	Determines if the matrix is a null matrix (0,0,0, 0,0,0, 0,0,0)	True if it is a null matrix
multiply(value)	Multiplies each component of the matrix by the value	Result of matrix times specified value
multiply(matrix)	Multiplies the matrix by the specified matrix	Results of matrix times specified matrix
multiply(vector)	Multiplies the specified vector by the matrix	Vector multiplied by the matrix
negate()	Calculates the negated matrix	Implicit
normalize()	Calculates the negated matrix	Implicit
setColumn(column, vector)	Sets the matrix column as a vector	Implicit
setElement(row, column, vector)	Sets the matrix element	Implicit
setForward(vector)	Sets the forward vector	Implicit
setRight(vector)	Sets the right vector	Implicit
setRow(row, vector)	Sets the matrix row as a vector	Implicit
setUp(vector)	Sets the up vector	Implicit
subtract(matrix)	Subtracts the specified matrix from this matrix	Implicit
toString()	Formats the matrix as a string, e.g. "[[1, 0, 0], [0, 1, 0], [0, 0, 1]]"	String representation of matrix
transpose()	Creates the transposed/inverse of this matrix	Implicit

Matrix Functions

Static functions do not require an associated *Matrix* object.

Function	Description	Returns
Matrix.diff(left, right)	Calculates the difference between two matrices	Left matrix minus right matrix
Matrix.getAxisRotation(vector, angle)	Calculates a rotation matrix	Rotation matrix of 'angle' radians around the axis 'vector'
Matrix.getXRotation(angle)	Calculates a rotation matrix around the X-axis	Rotation matrix of 'angle' radians around the X-axis
Matrix.getXYZRotation(abc)	Calculates the rotation matrix for the given angles	Rotation matrix that satisfies the specified XYZ rotations
Matrix.getYRotation(angle)	Calculates a rotation matrix around the Y-axis	Rotation matrix of 'angle' radians around the Y-axis
Matrix.getZRotation(angle)	Calculates a rotation matrix around the Z-axis	Rotation matrix of 'angle' radians around the Z-axis
Matrix.sum(left,right)	Adds the two matrices	Left matrix plus right matrix

Static Matrix Functions

```
var abc = m.getEuler2(EULER_ZXZ_R);          // abc = ZXZ Euler angles for m
var t = m.getTransposed();                   // t = inverse/transposed matrix of m
var fwd = m.getForward();                    // fwd = forward (Z) vector of matrix m
var v = new Vector(0, 0, 1);
var q = m.multiply(v);                       // q = transformation of v though matrix m
var r = Matrix.getZRotation(toDeg(30));      // r = matrix rotated 30 degrees about Z
```

Sample Matrix Expressions

3.4 Expressions

Variables can be assigned a simple value or text string, or can be more complex in nature containing a list of variables or literals and operators that perform operations on the values contained in the expression. The following table lists the common operators supported by JavaScript. and provides samples using the operators. The operator precedence is also listed (column P), where the operators with a higher precedence number are performed prior to the operators of a lower precedence number. Operators with the same precedence number will calculate in the order that they appear in the expression.

Unary operators only require a single operand instead of two. For example, $y = x++$ will increment the variable x after it is assigned to the variable y.

P	Operator	Operands	Description
13	()	Expression	Overrides the assigned precedence of operators
12	++	Integer	Unary increment
	--	Integer	Unary decrement
	~	Integer	Unary bitwise complement
	!	Boolean	Unary logical complement (not)
11	*	Number	Multiplication
	/	Number	Division
	%	Number	Remainder
10	+	Number, String	Addition
	-	Number	Subtraction
9	<<	Integer	Bitwise shift left
	>>	Integer	Bitwise shift right
8	<	Number, String	Less than
	<=	Number, String	Less than or equal to
	>	Number, String	Greater than
	>=	Number, String	Greater than or equal to
7	==	Any	Equal to
	!=	Any	Not equal to
	===	Any	Equal to and same variable type
	!==	Any	Not equal to and same variable type
6	&	Integer	Bitwise AND
5	^	Integer	Bitwise XOR
4	|	Integer	Bitwise OR

P	Operator	Operands	Description
3	&&	Boolean	Logical AND
2	||	Boolean	Logical OR
1	=	Any	Assignment
	+=	Number, String	Assignment with addition
	-=	Number	Assignment with subtraction
	*=	Number	Assignment with multiplication
	/=	Number	Assignment with division

Expression Operators

x	y	Expression	Result	Expression	Result
3	5	z = x + y * 3	18	z = (x + y) * 3	24
		z = ++x	z = 4, x = 4	z = x++	z = 3, x = 4
		x += y	8	x *= y	15
		z = y / x	1.667	z = y % x	2.0
"Start"	"-End"	z = x + y	"Start-End"	x += y	"Start-End"
2	3	z = x & y	2	z = x | y	3
1	"1"	z = x == y	true	x === y	false
true	false	z = x	true	z = !y	true
		z = x || y	true	z = x && y	false

Sample Expressions

3.5 Conditional Statements

Conditional statements are commands or functions that will test the results of an expression and then process statements based on the outcome of the conditional. Conditionals typically check Boolean type expressions, but can also be used to test if a value is *undefined* or a string is blank.

This section describes the conditional statements and functions used when developing a post processor. Some of the conditionals are supported by JavaScript and others are inherent in the post processor kernel.

3.5.1 The if Statement

The *if* statement is the most common method for testing a conditional and executing statements based on the outcome of the test. It can contain a single body of statements to execute when the expression is true, a second body of statements to execute when the expression is false, or it can contain multiple conditionals that are checked in order using the *else if* construct.

As with all commands that affect a body of code, *if* statements can be nested inside of other *if* bodies and loops.

The syntax of *if* statements should follow the Autodesk standard of always including the {} brackets around each body of code, specifying the opening bracket ({) on the conditional line, and the closing

bracket (}) at the start of the line following the body of code for each section as shown in the following examples.

```
if (conditional1) {
  // execute code if conditional1 is true
}

if (conditional1) {
  // execute code if conditional1 is true
} else {
  // execute code if conditional1 is false
}

if (conditional1) {
  // execute code if conditional1 is true
} else if (conditional2) {
  // execute code if conditional1 is false and conditional2 is true
} else {
  // execute code if all conditionals are false
}
```

If Statement Syntax

```
if (hasParameter("operation-comment")) {
  comment = getParameter("operation-comment");
}

if (isProbeOperation()) {
  var workOffset = probeOutputWorkOffset ? probeOutputWorkOffset : currentWorkOffset;
  if (workOffset > 99) {
    error(localize("Work offset is out of range."));
    return;
  } else if (workOffset > 6) {
    probeWorkOffsetCode = probe100Format.format(workOffset - 6 + 100);
  } else {
    probeWorkOffsetCode = workOffset + "."; // G54->G59
  }
}
```

Sample If Statements

3.5.2 The switch Statement

The *switch* statement is similar to an *if* statement in that it causes a branch in the flow of a program's execution based on the outcome of a conditional. *switch* statements are typically used when checking the value of a single variable, whereas *if* conditionals can test complex expressions.

The syntax of *switch* bodies will contain a single switch statement with a variable whose value determines the code to be executed. *case* statements will be included in the *switch* body, with each one containing the value that causes its body of code to be executed. The end of each *case* body of code must have a *break* statement so that the next *case* body of code is not executed. A *default* statement can be defined that contains code that will be executed if the *switch* variable does not match any of the *case* values.

case statements should follow the Autodesk standard of always including specifying the opening bracket ({) on the *switch* line, and the closing bracket (}) at the start of the line at the end of the body of code for each section. The *case* statements will be aligned with the *switch* statement and all code within each *case* body will be indented.

```
switch (variable) {
case value1:
  // execute if variable = value1
  break;
case value2:
  // execute if variable = value2
case value3:
  // execute if variable = value3
default:
  // execute if variable does not equal value1, value2, or value3
  break;
}
```

Switch Block Syntax

```
switch (coolant) {
  case COOLANT_FLOOD:
    m = 8;
    break;
  case COOLANT_THROUGH_TOOL:
    m = 88;
    break;
  case COOLANT_AIR:
    m = 51;
    break;
  default:
    onUnsupportedCoolant(coolant);
  }
}
```

Sample Switch Blocks

3.5.3 The Conditional Operator (?)

The ? conditional operator tests an expression and returns different values based on whether the expression is true or false. It is a compact version of a simple *if* block and is used in an assignment type statement or as part of an expression.

```
var a = conditional ? true_value : false_value;
```

? Conditional Operator

In the above syntax, *a* will be assigned *true_value* if the conditional is true, or *false_value* if it is false.

```
homeGcode = properties.useG30 ? 30 : 28;

// could be expanded into this if block
if (properties.useG30) {
  homeGcode = 30;
} else {
  homeGcode = 28;
}
```

Sample ? Conditional Operator

3.5.4 The typeof Operator

The *typeof* operator is not a conditional operator per the general terminology, but it is always used as a part of a conditional to determine if a function or variable exists. When used in an expression it will return a string that describes the variable type of the operand. This is the only way to test if a function exists prior to calling the function or if a variable exists before referencing it. If you try to reference a non-existent variable or function without testing to see if it exists first, the post processor will terminate with an error.

The *typeof* operator is followed by a single operand name, i.e. "typeof variable". It can return the following string values.

Operand Type	Return Values
number	"number"
string	"string"
boolean	"boolean"
object, array, null	"object"
function	"function"
undefined	"undefined"

typeof Return Values

```
if ((typeof getHeaderVersion == "function") && getHeaderVersion()) {
  writeComment(localize("post version") + ": " + getHeaderVersion());
}
```

Sample typeof Usage

3.5.5 The conditional Function

The *conditional* function will test an expression and if it is true will return the specified value. If the expression is false, then a blank string is returned. The *conditional* function is mainly used for determining if a specific code should be output in a block.

```
conditional(expression, true_value)
```

conditional Syntax

```
writeBlock(
  gRetractModal.format(98), gAbsIncModal.format(90), gCycleModal.format(82),
  getCommonCycle(x, y, z, cycle.retract),
  conditional(P > 0, "P" + milliFormat.format(P)), //optional
  feedOutput.format(F)
);
```

conditional Usage

3.5.6 try / catch

The *try/catch* block is an exception handling mechanism. This allows the post processor to control the outcome of an exception. Depending on the exception that is encountered, the JavaScript code could continue processing or terminate with an error. The *try/catch* block is used to override the normal processing of exceptions in JavaScript.

```
try {
  // code that may generate an exception
} catch (e) { // e is a local variable that contains the exception object or value that was thrown
  // code to perform if an exception is encountered
}
```

try/catch Syntax

```
try {
  programId = getAsInt(programName);
} catch(e) {
  error(localize("Program name must be a number."));
  return;
}
```

try/catch Usage

3.5.7 The validate Function

The *validate* function tests an expression and raises an exception if the expression is false. The post processor will typically output an error if an exception is raised, so in essence, the *validate* function determines if an expression is true or false and outputs an error using the provided message if it is false.

```
validate(expression, error_message)
```

validate Syntax

```
validate(retracted, "Cannot cancel length compensation if the machine is not fully retracted.");
```

Sample validate Code

In the above sample, an error will be generated if *retracted* is set to false.

3.5.8 Comparing Real Values

Real values are stored as binary numbers and are not truncated as you see them in an output file, so there are times when the numbers are not equal even if they show as the same value in the output file. For this reason, it is recommended that you either use a tolerance or truncate them when comparing their values. The *format.getResultingValue* function can be used to truncate a number to a fixed number of decimal places.

```
var a = 3.141592654;
var b = 3.141593174;

// simple comparison
if (a == b) {  // false

// comparison using a tolerance
var toler = .0001;
if (Math.abs(a – b) <= toler) { // true

// comparison using truncated values
var spatialFormat = createFormat({decimals:4});
if ((spatialFormat.getResultingValue(a) - spatialFormat.getResultingValue(b)) == 0) { // true
```

Comparing Real Values

3.6 Looping Statements

Loops perform repetitive actions. There are various styles of looping statements; *for*, *for/in*, *while*, and *do/while*. You should choose the looping statement that lends itself to the style of loop you are coding.

The syntax of looping statements should follow the Autodesk standard of always including the {} brackets around each body of code, specifying the opening bracket ({) on the looping statement, and the closing bracket (}) at the start of the line following the body of code for the loop. Loops can be nested within other bodies of code, like conditionals or other loops.

3.6.1 The for Loop

The *for* loop is the most common of the looping statements. It includes a counter and an expression on when to end the loop, so it will loop through the body of the loop a fixed number of times, unless interrupted by the *break* command. The *counter variable* is initialized before the loop starts and is

tested when the *expression* is evaluated before each iteration of the loop. The *counter variable* is incremented at the end of the loop, just before the *expression* is evaluated again.
Multiple counters can be initialized and incremented in a for loop by separating the counters with a comma (,).

```
for(initialize_counter; test expression ; increment_counter) {
  // body of loop
}
```

for Loop Syntax

```
for (var i = 0; i < getNumberOfSections(); ++i) { // loop for the number of sections in intermediate file
  if (getSection(i).workOffset > 0) {
    error(localize("Using multiple work offsets is not possible if the initial work offset is 0."));
    return;
  }
}

for (i = 0, j = ary.length - 1 ; i < ary.length / 2; ++i, --j) { // reverse the order of an array
  var tl = ary[i];
  ary[i] = ary[j];
  ary[j] = tl;
}
```

Sample for Loops

3.6.2 The for/in Loop

The *for/in* loop allows you to traverse the properties of an *object*. It is not commonly used in post processors (except for the dump.cps post processor), but can be useful for debugging the property names and values in an *object*.

```
for(variable in object) {
  // body of loop
}
```

for/in Loop Syntax

```
for(var element in properties) { // write out the property table
  writeln("properties." + element + " = " + properties[element]);
}
```

Sample for/in Loop

3.6.3 The while Loop

The *while* loop evaluates an expression and will execute the body of the loop when the expression is true and will end the loop when the expression is false. Since the expression is tested at the top of the loop, the body of code in the loop will not be executed when the expression is initially set to false.

```
while (expression) {
  // body of loop
}
```

while Loop Syntax

```
while (c > 2*Math.PI) {
  c -= 2 * Math.PI;
}
```

Sample while Loop

3.6.4 The do/while Loop

The *do/while* loop is pretty much the same as the while loop, but the expression is tested at the end of the loop rather than at the start of the loop. This means that the loop will be executed at least once, even if the expression is initially set to false.

```
do {
  // body of loop
} while (expression)
```

do/while Loop Syntax

```
var i = 0;
var found = false;
do {
  if (mtype[i++] == "Start") {
    found = true;
  }
} while (!found && i < mtype.length);
```

Sample do/while Loop

3.6.5 The break Statement

The *break* statement is used to interrupt a loop or switch statement prematurely. When the *break* statement is encountered during a loop or switch body, then the innermost loop/switch will be ended and control will move to the first statement outside of the loop/switch.

break is pretty much mandatory with switch statements. For loops, *break* can be used to get out of the loop when an error is encountered, or when a defined pattern is found within an array.

```
for (i = 0; i < mtype.length; ++i) {
  if (mtype[i] == "Start") {
    break; // exits the loop
  }
}
```

Sample Usage of break Command

3.6.6 The continue Statement

The *continue* statement is used to bypass the remainder of the loop body and restarts the loop at the next iteration.

```
for (i = 0; i < mtype.length; ++i) {
  if (mtype[i] < 0) {
    continue; // skips this iteration of the loop and continues with the next iteration
  }
  …
}
```

Sample Usage of break Command

3.7 Functions

Functions in JavaScript behave in the same manner as functions in other high-level programming languages. In a post processor all code, except for the global settings at the top of the file, is contained in functions, either entry functions (*onOpen*, *onSection*, etc.) or helper functions (*writeBlock*, *setWorkPlane*, etc.). The code in a function will not be processed until that function is called from within another routine (for the sake of clarity the calling function will be referred to as a 'routine' in this section). Here are the main reasons for placing code in a separate function rather than programming it in the upper level routine that calls the function.

1. The same code is executed in different areas of the code, either from the same function or in multiple functions. Placing the common code in its own function eliminates duplicate code from the file, making it easier to understand and maintain.
2. To logically separate logic and make it easier to understand. Separating code into its own function can keep the calling routine from becoming too large and harder to follow, even if the function is only called one time.

3.7.1 The function Statement

A function consists of the function statement, a list of arguments, the body of the function (JavaScript code), and optional return statement(s).

```
function name([arg1 [,arg2 […, argn]]]) {
  …
  code
  …
}
```

function Statement Syntax

The argument list is optional and contains identifiers that are passed into the function by the calling routine. The arguments passed to the function are considered read-only as far as the calling routine is concerned, meaning that any changes to these variables will be kept local to the called function and not propagated to the calling routine. You use the *return* statement to return value(s) to the calling routine.

```
function writeComment(text) {
  writeln(formatComment(text));  // text is accepted as an argument and passed to formatComment
}
```

Sample function Definition

Arguments accepted by a function can either be named identifiers as shown in the previous example, or you can use the *arguments* array to reference the function arguments. The *arguments* array is built-in to JavaScript and is treated as any other *Array* object, meaning that it has the length property and access to the *Array* attributes and functions.

```
  transferType = parseChoice(properties.transferType,"PHASE","SPEED","STOP");
…
function parseChoice() {
  for (var i = 1; i < arguments.length; ++i) {
    if (String(arguments[0]).toUpperCase() == String(arguments[i]).toUpperCase()) {
      return i - 1;
    }
  }
  return -1;
}
```

Sample Usage of arguments Array

3.7.2 Calling a function

A function call is treated the same as any other expression. It can be standalone, assign a value, and be placed anywhere within an expression. The value returned by the called function is treated as any other variable. You simply type the name of the function with its arguments.

```
setWorkPlane(abc); // function does not return a value
seqno = formatSequenceNumber(); // function returns a value
circumference = getRadius(circle) * 2.0 * Math.PI; // function used in a regular expression
```

Sample function Calls

3.7.3 The return Statement

As you can see in the previous sections, a function can be treated the same as any other expression and all expressions have values. The *return* statement is used to provide a value back to the calling routine. You will recall that a function does not have to return a value, in this case you do not have to place a return statement in the function, the function will automatically return when the end of the function body

is reached. You can place a *return* statement anywhere within the function, the function will be ended whenever a *return* statement is reached.

```
return [expression]
```

return Statement Syntax

The return value can be any valid variable type; a number, string, object, or array. If you want to return multiple values from a function, then you must return either an object or an array. You can also propagate the JavaScript *this* object which will be automatically returned to the calling routine when the end of the function is reached or when processing a return statement without an expression. If the *this* object is used, then the function will be used to create a new object and you will need to define the function call as if you were creating any other type of object as shown in the following example.

```
function writeComment(text) {
  writeln(formatComment(text));
} // implicit return

function parseChoice() {
  for (var i = 1; i < arguments.length; ++i) {
    if (String(arguments[0]).toUpperCase() == String(arguments[i]).toUpperCase()) {
      return i - 1; // return the matching choice
    }
  }
  return -1; // return choice not found
}

function FeedContext(id, description, feed) {
  this.id = id;
  this.description = description;
  this.feed = feed;
} // return this object {id, description, feed}

var feedContext = new FeedContext(id, "Cutting", feedCutting); // create new FeedContext object
```

Sample return Usage

4 Entry Functions

The post processor Entry functions are the interface between the kernel and the post processor. An Entry function will be called for each record in the intermediate file. Which Entry function is called is determined by the intermediate file record type. All Entry functions have the 'on' prefix, so it is recommended that you do not use this prefix with any functions that you add to the post processor.

Here is a list of the supported Entry functions and when they are called. The following sections in this Chapter provide more detailed documentation for the most common of the Entry functions.

Entry Function	Invoked When …
onCircular(clockwise, cx, cy, cz, x, y, z, feed)	Circular move
onClose()	End of post processing
onCommand(value)	Manual NC command not handled in its own function
onComment(string)	*Comment* Manual NC command
onCycle()	Start of a cycle
onCycleEnd()	End of a cycle
onCyclePoint(x, y, z)	Each cycle point
onDwell(value)	*Dwell* Manual NC command
onLinear(x, y, z, feed)	3-axis cutting move
onLinear5D(x, y, z, a, b, c, feed)	5-axis cutting move
onMachine()	Machine configuration changes
onMovement(value)	Movement type changes
onOpen()	Post processor initialization
onOrientateSpindle(value)	Spindle orientation is requested
onParameter(string, value)	Each parameter setting
onPassThrough(string)	*Pass through* Manual NC command
onPower(boolean)	Power mode for water/plasma/laser changes
onRadiusCompensation()	Radius compensation mode changes
onRapid(x, y, z)	3-axis Rapid move
onRapid5D(x, y, z, a, b, c)	5-axis Rapid move
onRewindMachine(a, b, c)	Rotary axes limits are exceeded
onSection()	Start of an operation
onSectionEnd()	End of an operation
onSectionEndSpecialCycle()	End of a special cycle operation
onSectionSpecialCycle()	Start of a special cycle operation (Stock Transfer)
onSpindleSpeed(value)	Spindle speed changes
onTerminate()	Post processing has completed, output files are closed
onToolCompensation(value)	Tool compensation mode changes

Entry Functions

4.1 Global Section

The global section is not an Entry function, but rather is called when the post processor is first initialized. It defines settings used by the post processor kernel, the property table displayed with the post processor dialog inside of HSM, definitions for formatting output codes, and global variables used by the post processor.

While the global section is typically located at the top of the post processor, any variables defined outside of a function are in the global section and accessible by all functions, even the functions defined before the variable. You may notice global variables being defined in the middle of the post processor code just before a function. This allows for a group of functions to be easily cut-and-pasted from one post to another post, including the required global variables.

4.1.1 Kernel Settings

Some of the variables defined in the global section are actually defined in and used by the post engine. These variables are usually at the very top of the file and are easily discerned, since they are not preceded by *var*. The following table provides a description of the kernel settings that you will find in most post processors.

Setting	Description
allowedCircularPlanes	Defines the allowed circular planes. This setting is described in the *onCircular* section.
allowHelicalMoves	Specifies whether helical moves are allowed. This setting is described in the *onCircular* section.
allowSpiralMoves	Specifies whether spiral moves are allowed. This setting is described in the *onCircular* section.
capabilities	Defines the capabilities of the post processor. The capabilities can be CAPABILITY_MILLING, CAPABILITY_TURNING, CAPABILITY_JET, CAPABILITY_SETUP_SHEET, and CAPABILITY_INTERMEDIATE. Multiple capabilities can be enabled by using the logical OR operator. *capabilities = CAPABILITY_MILLING | CAPABILITY_TURNING;*
certificationLevel	Certification level of the post configuration used to determine if the post processor is certified to run against the post engine. This value rarely changes.
description	Short description of post processor. This will be displayed along with the post processor name in the *Post Process* dialog in HSM when selecting a post processor to run.
extension	The output NC file extension.
highFeedMapping	Specifies the high feed mapping mode for rapid moves. Valid modes are HIGH_FEED_NO_MAPPING, HIGH_FEED_MAP_MULTI, HIGH_FEED_MAP_XY_Z, and HIGH_FEED_MAP_ANY. This setting can be changed dynamically in the Property table when running the post processor.
highFeedrate	Specifies the feedrate to use when mapping rapid moves to linear moves.
legal	Legal notice of company that authored the post processor
mapToWCS	Specifies whether the work plane is mapped to the model origin and work plane. When disabled the post is responsible for handling mapping from the model origin to the setup origin. This variable must be defined using the following syntax and can only be defined in the global section. Any

Setting	Description

	deviation from this format, including adding extra spaces, will cause this command to be ignored.
	
	mapToWCS = true;
	mapToWCS = false;

mapWorkOrigin	Specifies whether the coordinates are mapped to the work plane origin. When disabled the post is responsible for handling the work plane origin. This variable must be defined using the following syntax and can only be defined in the global section. Any deviation from this format, including adding extra spaces, will cause this command to be ignored.
	
	mapWorkOrigin = true;
	mapWorkOrigin = false;

maximumCircularRadius	Specifies the maximum radius of circular moves that can be output as circular interpolation and can be changed dynamically in the Property table when running the post processor. This setting is described in the *onCircular* section.
maximumCircularSweep	Specifies the maximum circular sweep of circular moves that can be output as circular interpolation. This setting is described in the *onCircular* section.
minimumChordLength	Specifies the minimum delta movement allowed for circular interpolation and can be changed dynamically in the Property table when running the post processor. This setting is described in the *onCircular* section.
minimumCircularRadius	Specifies the minimum radius of circular moves that can be output as circular interpolation and can be changed dynamically in the Property table when running the post processor. This setting is described in the *onCircular* section.
minimumCircularSweep	Specifies the minimum circular sweep of circular moves that can be output as circular interpolation. This setting is described in the *onCircular* section.
minimumRevision	The minimum revision of the post kernel that is supported by the post processor. This value will remain the same unless the post processor takes advantage of functionality added to a later version of the post engine that is not available in earlier versions.
programNameIsInteger	Specifies whether the program name must be an integer (*true*) or can be a text string (*false*).
tolerance	Specifies the tolerance used to linearize circular moves that are expanded into a series of linear moves. This setting is described in the *onCircular* section.
unit	Contains the output units of the post processor. This is usually the same as the input units, either MM or IN, but can be changed in the *onOpen* function of the post processor by setting it to the desired units.
vendor	Name of the machine tool manufacturer.
vendorUrl	URL of the machine tool manufacturer's web site.

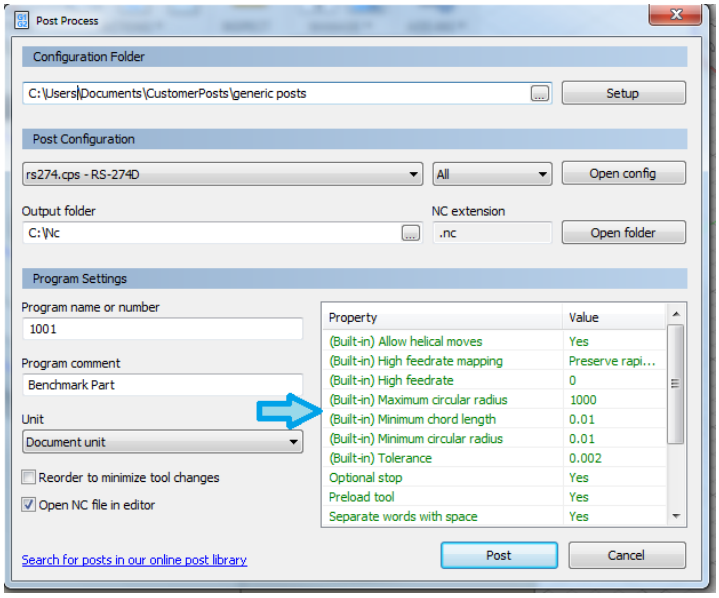
```
description = "RS-274D";
vendor = "Autodesk";
vendorUrl = "http://www.autodesk.com";
legal = "Copyright (C) 2012-2017 by Autodesk, Inc.";
certificationLevel = 2;
minimumRevision = 24000;

longDescription = "Generic post for the RS-274D format. Most CNCs will use a format very similar
to RS-274D. When making a post for a new CNC control this post will often serve as the basis.";

extension = "nc";
setCodePage("ascii");

capabilities = CAPABILITY_MILLING;
tolerance = spatial(0.002, MM);

minimumChordLength = spatial(0.01, MM);
minimumCircularRadius = spatial(0.01, MM);
maximumCircularRadius = spatial(1000, MM);
minimumCircularSweep = toRad(0.01);
maximumCircularSweep = toRad(180);
allowHelicalMoves = true;
allowedCircularPlanes = undefined; // allow any circular motion
```

Sample Post Kernel Settings Code

4.1.2 Property Table

Stock post processors are designed to run the machine without any modifications, but may not create the output exactly as you would like to see it. The Property Table contains settings that can be changed at runtime so that the stock post can remain generic in nature, but still be easily customized by various users. The settings in the Property Table will typically be used to control small variations in the output created by the post processor, with major changes handled by settings in the Fixed Settings section.

When you Post Process from HSM you will be presented with a dialog that allows you to select the post processor to execute, the output file path, and other settings. The Property Table will also be displayed in the dialog allowing you to override settings within the post processor each time it is run.



Property Table in Post Process Dialog

The Property Table is defined in the post processor so you have full control over the information displayed in it, with the exception of the *Built-in* properties, which are displayed with every post processor and define the post kernel variables described previously. The *properties* object defined in the post processor defines the property names as they are used in the post processor along with the default values assigned to each property.

```
// user-defined properties
properties = {
  writeMachine: true, // write machine
  writeTools: true, // writes the tools
  preloadTool: true, // preloads next tool on tool change if any
  showSequenceNumbers: true, // show sequence numbers
  sequenceNumberStart: 10, // first sequence number
  sequenceNumberIncrement: 5, // increment for sequence numbers
  optionalStop: true, // optional stop
  separateWordsWithSpace: true // specifies that the words should be separated with a white space
  rotaryTableAxis: "none" // none, X, Y, Z, -X, -Y, -Z
};
```

Property Table Definition

The default values for the variables can be a number, boolean, or a text string.

The *propertyDefinitions* object gives you control on how you want the properties displayed to the user in the Property Table. There should be a matching entry in the *propertyDefinitions* object for every entry in the *properties* object. If there is not a matching entry, then the variable name from the *properties* object will be displayed and this property will not have tool tip text associated with it.

```
// user-defined property definitions
```

```
propertyDefinitions = {
  writeMachine: {title:"Write machine",
    description:"Output the machine settings in the header of the code.", group:0, type:"boolean"},
    writeTools: {title:"Write tool list", description:"Output a tool list in the header of the code.",
    group:0, type:"boolean"},
  preloadTool: {title:"Preload tool",
    description:"Preloads the next tool at a tool change (if any).",
    type:"boolean"},
  showSequenceNumbers: {title:"Use sequence numbers",
    description:"Use sequence numbers for each block of outputted code.", group:1, type:"boolean"},
    sequenceNumberStart: {title:"Start sequence number", description:"Sequence number start value",
    group:1, type:"integer"},
  sequenceNumberIncrement: {title:"Sequence number increment",
    description:"The amount by which the sequence number is incremented by in each block.",
    group:1, type:"integer"},
  optionalStop: {title:"Optional stop",
    description:"Outputs optional stop code during when necessary in the code.",
    type:"boolean"},
  separateWordsWithSpace: {title:"Separate words with space",
    description:"Adds spaces between words if 'yes' is selected.",
    type:"boolean"},
  rotaryTableAxis: {
    title: "Rotary table axis",
    description: "Selects the rotary table axis orientation.",
    type: "enum",
    values:[
      {title:"No rotary", id:"none"},
      {title:"Along +X", id:"x"},
      {title:"Along +Y", id:"y"},
      {title:"Along +Z", id:"z"},
      {title:"Along -X", id:"-x"},
      {title:"Along -Y", id:"-y"},
      {title:"Along -Z", id:"-z"}
    ]
  }
};
```

Property Table User Interface Definition

The following table describes the supported variable properties in the *propertyDefinitions* object. It is important that the format of the *propertyDefinitions* object follows the above example, where the name of the variable is first, followed by a colon (:), and the properties enclosed in braces ({}). The *values* property is an array and its properties must be enclosed in brackets ([]).

Property	Description
title	Description of the variable displayed in the User Interface within the *Property* column.

Property	Description
description	A description of the variable displayed as the tool tip when the mouse is positioned over this variable.
group	The group number that this variable belongs to. All variables with the same group number will be displayed together in the User Interface. (This property is not supported as of this writing)
type	Defines the input type. The input types are described in the following table.
values	Contains a list (array) of choices for the *enum* or *integer* input type. It is not valid with any other input type.

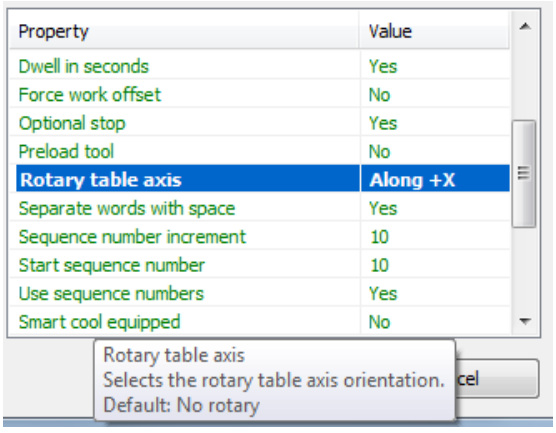
Property Table User Interface Definition

Input Type	Description
"integer"	Integer value
"number"	Real value
"spatial"	Real value
"angle"	Angular value in degrees
"boolean"	true or false
"string"	Text string
"enum"	The *enum* input type defines this variable as having fixed choices associated with it. These choices are defined individually in the *values* property array. An *enum* input type should be defined using string values.

Property Table Input Types

Values Property	Description
title	The text of the choice item displayed in the User Interface for this variable.
id	The value that will be returned in the variable when the post processor is called. All references to this property, e.g. *properties.rotaryTableAxis*, in the post processor should expect only one of these *id* values as its value. The *id* must be a text string when associated with an *enum* input type or an integer value when associated with an *integer*.

Enum Choices Properties



Property Table with Titles and Tool Tips

4.1.3 Format Definitions

The format definitions area of the global section is used to define the formatting of codes output to the NC file. It consists of the format definitions (*createFormat*) as well as definitions that determine when the codes will be output or suppressed (*createModal*, *createVariable*, *createReferenceVariable*, *createIncrementalVariable*).

The *createFormat* command defines how codes are formatted before being output to the NC file. It can be used to create a complete format for an output code, including the letter prefix, or to create a primary format that is referenced with the output definitions. It has the following syntax.

```
createFormat({specifier:value, specifier:value, ...});
```

createFormat Syntax

The specifiers must be enclosed in braces ({}) and contain the specifier name followed by a colon (:) and then by a value. Multiple specifiers are separated by commas.

Specifier	Value
prefix	Defines the prefix of the output value as a text string. The prefix should only be defined if this is a standalone format and is not used for multiple output definitions.
suffix	Defines the suffix of the output value as a text string. The suffix should only be defined if this is a standalone format and is not used for multiple output definitions.
decimals	Defines the number digits to the right of the decimal point to output. The default is 6.
forceDecimal	When set to *true* the decimal point will always be included with the formatted number. *false* will remove the decimal point for integer values.
forceSign	When set to *true* will force the output of a plus (+) sign on positive numbers. The default is *false*.
width	Specifies the minimum width of the output string. If the formatted value's width is less than the *width* value, then the start of the number will either be filled with spaces or zeros depending on the value of *zeropad*. If the format is used to output a code to the NC file be sure to set *zeropad* to *true*, otherwise the prefix and value could be separated by spaces. The width of the output string includes the decimal point when it is included in the number, but not the sign of the number. The default is 0.
zeropad	When set to *true* will fill the beginning of the output string with zeros to match the specified width. If *width* is not specified or the output string is longer than *width*, then no zeros will be added. The default is *false*.
trim	When set to *true* the trailing zeros will be trimmed from the right of the decimal point. The default is *true*.
trimLeadZero	When set to *true* will trim the lead zero from a floating-point number if the number is fractional, e.g. .123 instead of 0.123. The default is *false*.

Specifier	Value
scale	Defines a scale factor to multiply the value by prior to formatting it for output. *scale* can be a number or a number designator, such as *DEG*. The default is 1.
offset	Defines a number to add to the value prior to formatting it for output. The default is 0.
separator	Defines the character to use as the decimal point. The default is '.'.
inherit	Inherits all properties from an existing *format*.

createFormat Properties

Once a *format* is created, it can be used to create a formatted text string of a value that matches the properties in the defined *format*. The following table describes the functions defined in the *format* object.

Function	Description
areDifferent(a, b)	Returns *true* if the input values are different after being formatted.
format(value)	Returns the formatted text string representation of the number.
getError(value)	Returns the inverse of the remaining portion of the value that is not formatted for the number. For example, if the formatted value of 4.5005 is "4.500", then the value returned from *getError* will be -0.0005.
getMinimumValue()	Returns the minimum value that can be formatted using this *format*, for example, 1 for *decimals:0*, .1 for *decimals:1*, etc.
getResultingValue (value)	Returns the real value that the formatted output text string represents.
isSignificant(value)	Returns true if the value will be non-zero when formatted.

***format* Functions**

```
var xFormat = createFormat({decimals:3, trim:false, forceSign:true});
xFormat.format(4.5); // returns "+4.500"
xFormat.areDifferent(9.123, 9.1234); // returns false, both numbers are 9.123
xFormat.getMinimumValue(); // returns 0.001
xFormat.isSignificant(.0005); // returns true (rounded to .001)
xFormat.isSignificant(.00049); // returns false

var yFormat = createFormat({decimals:3, forceSign:true});
yFormat.format(4.5); // returns "+4.5"
yFormat.getResultingValue(3.1234); // returns 3.123

var toolFormat = createFormat({prefix:"T", decimals:0, zeropad:true, width:2});
toolFormat.format(7); // returns "T07"

var aFormat = createFormat({decimals:3, forceSign:true, forceDecimal:true, scale:DEG});
aFormat.format(Math.PI); // returns "+180."

var zFormat = createFormat({decimals:4, scale:10000, forceDecimal:false});
zFormat.format(1.23); // returns 12300 (leading zero suppression)
```

Example *format* Commands

4.1.4 Output Variable Definitions

The format object is used to format values, but has no connection to the output of the variable, except for formatting a text string that could be output. It does not know what the last output variable is, which is important when you do not want to output a code if the value has not changed from its previous output value.

The *createVariable*, *createModal*, *createReferenceVariable*, and *createIncrementalVariable* functions create output objects that are used to control the output of a code. The *createVariable* and *createModal* objects are used to output codes/registers only when they change from the previous output value, the *createReferenceVariable* is used to output values when they are different from a specified reference value, and the *createIncrementalVariable* is used for the output of incremental values, i.e. the output value will be an incremental value based on the previous value and the input value.

The *createVariable* and *createModal* objects can be used interchangeably since they both output only the values that have changed. In a post processor you will see that the *createModal* object is used for the output of G-code or M-code modal groups, where multiple codes can be output in a single block and will only be output when the code changes value from the previous code in this group. The *createVariable* object is used for all other code/register output such as the axes registers, spindle speed, feedrates, etc. The only difference in these objects the functions that belong to them, for example you can disable the output of a Variable, but not of a Modal.

You can use the *createFormat* object for codes/registers that should be output whenever they are encountered in the post, just be sure to add the prefix to the definition.

```
createVariable({specifier:value, specifier:value, ...}, format);
createModal({specifier:value, specifier:value, ...}, format);
createReferenceVariable({specifier:value, specifier:value, ...}, format);
createIncrementalVariable({specifier:value, specifier:value, ...}, format);
```

Output Variables Syntax

The specifiers must be enclosed in braces ({}) and contain the specifier name followed by a colon (:) and then by a value. Multiple specifiers are separated by commas. A *format* object is provided as the second parameter. Some of the specifiers are common to all three objects and some to a particular object, as listed in the following table.

Specifier	Object	Value
prefix	(all)	Text string that overrides the prefix defined in *format*.
force	(all)	When set to *true* forces the formatting of the value even if it does not change from the previous value. The default is *false*.
onchange	createVariable createModal	Defines the method to be invoked when the formatting of the value results in output.

Specifier	Object	Value
suffix	createModal	Text string that overrides the suffix defined in *format*.
first	createIncrementalVariable	Defines the initial value of an incremental variable. You will also have to call the *variable.format(first)* function after creating the IncrementalVariable to properly store the initial value.

Output Variable Properties

The *onchange* property typically defines a function that is called whenever the formatting of the variable results in an output text string, such as when the value changes or is forced out. The following example will force out the gMotionModal code whenever the plane code is changed.

```
var gPlaneModal = createModal({onchange:function () {gMotionModal.reset();}}, gFormat);
```

onChange Usage

Once an output variable is created, it can be used to create a formatted text string for output. The following table describes the functions assigned to the output variable objects. The functions are properties of the defined *variable* object.

Function	Object	Description
disable()	Variable ReferenceVariable IncrementalVariable	Disables this variable from being output. Will cause the return value from the *format* function to always be a blank string ("").
enable()	Variable Reference Variable IncrementalVariable	Enables this variable for output. This is the default condition when the variable is created.
format(value [,ref])	(all)	Returns the formatted text string representation of the number. Can return a blank string if the value is the same as the stored value in the Variable and Modal objects, the same as the reference value in the ReferenceVariable object, or generates a value of 0 in the IncrementalVariable object. The call to *format* for a ReferenceVariable object must contain the second *ref* parameter, which determines if the value should be formatted for output.
getCurrent()	Variable Modal IncrementalVariable	Returns the value currently stored in this variable.
isEnabled()	Variable ReferenceVariable IncrementalVariable	Returns *true* if this variable is enabled for output

Function	Object	Description
reset()	Variable Modal IncrementalVariable	Forces the output of the formatted text string on the next call to format, overriding the rules for not outputting a value.
setPrefix(prefix-text)	(all)	Overrides the prefix of the variable.
setSuffix(suffix-text)	Modal	Overrides the suffix of the variable.

***Variable* Functions**

```
var xyzFormat = createFormat({decimals:3, forceDecimal:true});
var xOutput = createVariable({prefix:"X"}, xyzFormat);
xOutput.format(4.5); // returns "X4.5"
xOutput.format(4.5); // returns "" (4.5 is currently stored in the xOutput variable)
xOutput.reset();        // force xOuput on next formatting
xOutput.format(4.5); // returns "X4.5"
xOutput.disable();     // disable xOutput formatting
xOutput.format(1.2); // returns "" since it is disabled

var gFormat = createFormat({prefix:"G", decimals:0, width:2, zeropad:true});
var gMotionModal = createModal({force:true}, gFormat);
gMotionModal.format(0); // returns G00
gMotionModal.format(0); // returns G00 (force is set to 'true')
gMotionModal.format.setPrefix("G1=");
gMotionModal.setSuffix("*");
gMotionModal.format(1); // returns "G1=01*"

var iOutput = createReferenceVariable({prefix:"I", forceDecimal}, xyzFormat);
iOutput.format(.001, 0); // returns "I0.001"
iOutput.format(.0001, 0); // returns ""

var zOutput = createIncrementalVariable({prefix:"Z", first:.5}, xyzFormat);
zOutput.format(.5); // after creating the IncrementalVariable you must call the format function
                    // with the same value as 'first' to properly set the initial value
zOutput.format(1.2); // returns "Z0.7"
zOutput.format(1.5); // returns "Z0.3"
zOutput.format(1.5); // returns ""
zOutput.format(0); // returns "Z-1.5"
```

Example *Variable* Commands

4.1.5 Fixed Settings

The fixed settings area of the global section defines settings in the post processor that enable features that may change from machine to machine, but are not common enough to place in the Property Table. These settings are usually not modified by the post processor, but can be modified to enable features on your machine that are disabled in a stock post processor or vice versa.

```
// fixed settings
var firstFeedParameter = 500;
var useMultiAxisFeatures = false;
var forceMultiAxisIndexing = false; // force multi-axis indexing for 3D programs
var maximumLineLength = 80; // the maximum number of characters allowed in a line
var minimumCyclePoints = 5; // min number of points in cycle operation to consider for subprogram

var WARNING_WORK_OFFSET = 0;

var ANGLE_PROBE_NOT_SUPPORTED = 0;
var ANGLE_PROBE_USE_ROTATION = 1;
var ANGLE_PROBE_USE_CAXIS = 2;
```

Sample Fixed Settings Code

4.1.6 Collected State

The collected state area of the global section contains global variables that will be changed during the execution of the post processor and are either referenced in multiple functions or need to maintain their values between calls to the same function.

```
// collected state
var sequenceNumber;
var currentWorkOffset;
```

Sample Collected State Code

4.2 onOpen

```
function onOpen() {
```

The *onOpen* function is called at start of each CAM operation and can be used to define settings used in the post processor and output the startup blocks.

1. Define settings based on properties
2. Define the multi-axis machine configuration
3. Output program name and header
4. Perform checks for duplicate tool numbers and work offsets
5. Output initial startup codes

4.2.1 Define Settings Based on Post Properties

The fixed settings section at the top of the post processor contain settings that are fixed and will not be changed during the processing of the intermediate file. Settings and variables that are dependant on the properties defined in the Property Table are defined in the *onOpen* function, since this is the function called when the post processor first starts.

Some of the variables that may be defined here are the maximum circular sweep, starting sequence number, formats, properties that can be changed using a Manual NC command, etc.

```
if (properties.useRadius) {
  maximumCircularSweep = toRad(90); // avoid potential center calculation errors for CNC
}

// define sequence number output
if (properties.sequenceNumberOperation) {
  properties.showSequenceNumbers = false;
}
sequenceNumber = properties.sequenceNumberStart;

// separate codes with a space in output block
if (!properties.separateWordsWithSpace) {
  setWordSeparator("");
}

// Manual NC command can change the transfer type
transferType = parseToggle(properties.transferType, "PHASE", "SPEED");
```

Defining Dynamic Variables in the onOpen Function

The majority of machines on the market today accept input in both inches and millimeters. It is possible that your machine must be programmed in only one unit. If this is the case, then you can define the *unit* variable in the onOpen function to force the output of all relevant information in inches or millimeters.

```
unit = MM;  // set output units to millimeters, use IN for inches
```

Support for Only One Input Unit

4.2.2 Define the Multi-Axis Configuration

The multi-axis machine configuration is defined in the *onOpen* function. Following is an example of this code. For a complete description of defining a multi-axis configuration please see the *Create the Rotary Axes Formats* section.

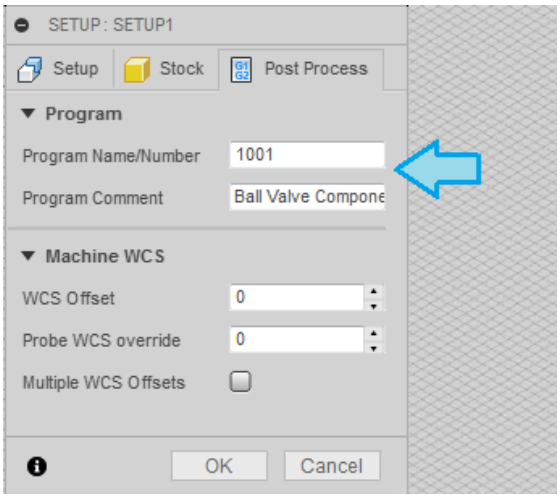
```
if (true) {
  var aAxis = createAxis({coordinate:0, table:true, axis:[1, 0, 0], range:[-35, 110], preference:1});
  var cAxis = createAxis({coordinate:2, table:true, axis:[0, 0, 1], cyclic:true, preference:0});
  machineConfiguration = new MachineConfiguration(aAxis, cAxis);
  setMachineConfiguration(machineConfiguration);
  optimizeMachineAngles2(1); // map tip mode
}

if (!machineConfiguration.isMachineCoordinate(0)) {
  aOutput.disable();
}
```

```
if (!machineConfiguration.isMachineCoordinate(1)) {
  bOutput.disable();
}
if (!machineConfiguration.isMachineCoordinate(2)) {
  cOutput.disable();
}
```

Defining the Machine Configuration

4.2.3 Output Program Name and Header

The program name and program comment are defined in the Post Process tab of the CAM setup in HSM. The *programNameIsInteger* variable defined at the top of the program determines if the program name needs to be a number or can be a text string.



Defining the Program Name and Comment

```
writeln("%"); // output start of NC file
if (programName) {
   var programId;
   try {
     programId = getAsInt(programName);
   } catch(e) {
     error(localize("Program name must be a number."));
     return;
   }
   if (!((programId >= 1) && (programId <= 99999))) {
     error(localize("Program number is out of range."));
     return;
   }
   writeln(
     "O" + oFormat.format(programId) +
     conditional(programComment, " " + formatComment(programComment.substr(0,
     maximumLineLength - 2 - ("O" + oFormat.format(programId)).length - 1)))
```

```
  );
  lastSubprogram = programId;
} else {
  error(localize("Program name has not been specified."));
  return;
}
```

Output the Program Name as an Integer and Program Comment

Some machines don't use a program number and accept the program name as a comment.

```
writeln("%"); // output start of NC file
if (programName) {
  writeComment(programName);
}
if (programComment) {
  writeComment(programComment);
}
```

Output the Program Name as a Comment

The program header can consist of the output filename, version numbers, the run date and time, the description of the machine, and the list of tools used in the program.

```
// Output current run information
if (hasParameter("generated-by") && getParameter("generated-by")) {
  writeComment("           " + localize("CAM") + ": " + getParameter("generated-by"));
}
if (hasParameter("document-path") && getParameter("document-path")) {
  writeComment("       " + localize("Document") + ": " + getParameter("document-path"));
}
var eos = longDescription.indexOf(".");
writeComment(localize(" Post Processor: ") + ((eos == -1) ?
  longDescription : longDescription.substr(0, eos + 1)));
if ((typeof getHeaderVersion == "function") && getHeaderVersion()) {
  writeComment("   " + localize("Post version") + ": " + getHeaderVersion());
}
if ((typeof getHeaderDate == "function") && getHeaderDate()) {
  writeComment("  " + localize("Post modified") + ": " + getHeaderDate());
}
var d = new Date();  // output current date and time
writeComment("          " + localize("Date") + ": " + d.toLocaleDateString() + " " +
  d.toLocaleTimeString());
```

Output the Description of the Current Run

```
// dump machine configuration
var vendor = machineConfiguration.getVendor();
var model = machineConfiguration.getModel();
```

```
var description = machineConfiguration.getDescription();

if (properties.writeMachine && (vendor || model || description)) {
  writeComment(localize("Machine"));
  if (vendor) {
    writeComment("  " + localize("vendor") + ": " + vendor);
  }
  if (model) {
    writeComment("  " + localize("model") + ": " + model);
  }
  if (description) {
    writeComment("  " + localize("description") + ": "  + description);
  }
}
```

Output Machine Information

In the above code sample, the machine information is retrieved from the machineConfiguration, but a machine configuration file is not always available to the post processor, so it is possible to hard code the machine description.

```
machineConfiguration.setVendor("Doosan");
machineConfiguration.setModel("Lynx");
machineConfiguration.setDescription(description);
```

Defining the Machine Information

```
// dump tool information
  if (properties.writeTools) {
    var zRanges = {};
    if (is3D()) {
      var numberOfSections = getNumberOfSections();
      for (var i = 0; i < numberOfSections; ++i) {
        var section = getSection(i);
        var zRange = section.getGlobalZRange();
        var tool = section.getTool();
        if (zRanges[tool.number]) {
          zRanges[tool.number].expandToRange(zRange);
        } else {
          zRanges[tool.number] = zRange;
        }
      }
    }

    var tools = getToolTable();
    if (tools.getNumberOfTools() > 0) {
      for (var i = 0; i < tools.getNumberOfTools(); ++i) {
        var tool = tools.getTool(i);
```

```
      var comment = "T" + toolFormat.format(tool.number) + " " +
        "D=" + xyzFormat.format(tool.diameter) + " " +
        localize("CR") + "=" + xyzFormat.format(tool.cornerRadius);
      if ((tool.taperAngle > 0) && (tool.taperAngle < Math.PI)) {
        comment += " " + localize("TAPER") + "=" + taperFormat.format(tool.taperAngle) +
          localize("deg");
      }
      if (zRanges[tool.number]) {
        comment += " - " + localize("ZMIN") + "=" +
          xyzFormat.format(zRanges[tool.number].getMinimum());
      }
      comment += " - " + getToolTypeName(tool.type);
      writeComment(comment);
    }
  }
}
```

Output List of Tools Used

4.2.4 Performing General Checks

Basic checks for using duplicate tool numbers, undefined work offsets, and other requirements can be done in the onOpen function since all operations can be accessed at any time during post processing.

```
if (false) {  // set to true to check for duplicate tool numbers w/different cutter geometry
  // check for duplicate tool number
  for (var i = 0; i < getNumberOfSections(); ++i) {
    var sectioni = getSection(i);
    var tooli = sectioni.getTool();
    for (var j = i + 1; j < getNumberOfSections(); ++j) {
      var sectionj = getSection(j);
      var toolj = sectionj.getTool();
      if (tooli.number == toolj.number) {
        if (xyzFormat.areDifferent(tooli.diameter, toolj.diameter) ||
            xyzFormat.areDifferent(tooli.cornerRadius, toolj.cornerRadius) ||
            abcFormat.areDifferent(tooli.taperAngle, toolj.taperAngle) ||
            (tooli.numberOfFlutes != toolj.numberOfFlutes)) {
          error(
            subst(
              localize("Using the same tool number for different cutter geometry for operation '%1' and
'%2'."),
              sectioni.hasParameter("operation-comment") ?
                sectioni.getParameter("operation-comment") : ("#" + (i + 1)),
              sectionj.hasParameter("operation-comment") ?
                sectionj.getParameter("operation-comment") : ("#" + (j + 1))
            )
          );
```

```
          return;
        }
      }
    }
  }
}
```

Check for Duplicate Tool Numbers using Different Cutter Geometry

```
// don't allow WCS 0 unless it is the only WCS used in the program
if ((getNumberOfSections() > 0) && (getSection(0).workOffset == 0)) {
  for (var i = 0; i < getNumberOfSections(); ++i) {
    if (getSection(i).workOffset > 0) {
      error(localize("Using multiple work offsets is not possible if the initial work offset is 0."));
      return;
    }
  }
}
```

Check for Work Offset 0 when Multiple Work Offsets are Used in Program

4.2.5 Output Initial Startup Codes

Codes that set the machine to its default condition are usually output at the beginning of the NC file. These codes could include the units setting, absolute mode, the feedrate mode, etc.

```
// output default codes
writeBlock(gAbsIncModal.format(90), gFeedModeModal.format(94), gPlaneModal.format(17),
  gFormat.format(49), gFormat.format(40), gFormat.format(80));

// output units code
switch (unit) {
case IN:
  writeBlock(gUnitModal.format(20));
  break;
case MM:
  writeBlock(gUnitModal.format(21));
  break;
}
```

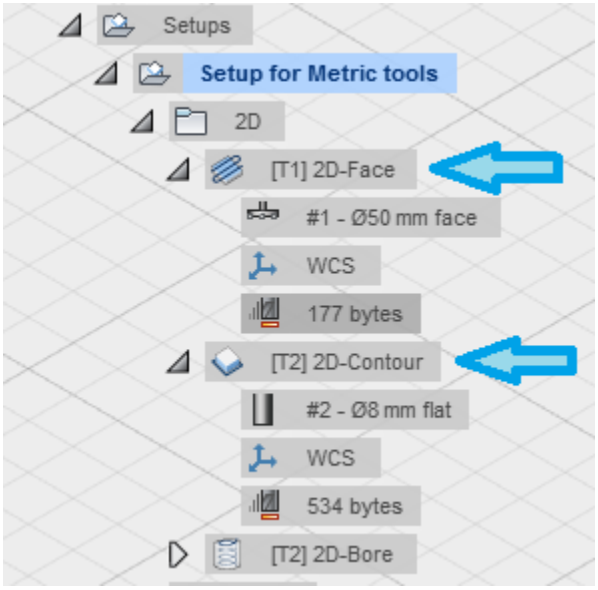
Output Initial Startup Codes

4.3 onSection

```
function onSection() {
```

The *onSection* function is called at start of each CAM operation and controls the output of the following blocks.

1. End of previous section
2. Operation comments and notes
3. Tool change
4. Work plane
5. Initial position



onSection is Called for Each Operation

The first part of *onSection* determines if there is a change in the tool being used and if the Work Coordinate System offset or Work Plane is different from the previous section. These settings determine the output required between operations.

```
  var insertToolCall = isFirstSection() ||
    currentSection.getForceToolChange && currentSection.getForceToolChange() ||
    (tool.number != getPreviousSection().getTool().number);

  var retracted = false; // specifies that the tool has been retracted to the safe plane
  var newWorkOffset = isFirstSection() ||
    (getPreviousSection().workOffset != currentSection.workOffset); // work offset changes
  var newWorkPlane = isFirstSection() ||
    !isSameDirection(getPreviousSection().getGlobalFinalToolAxis(),
currentSection.getGlobalInitialToolAxis());
```

Tool Change, Work Coordinate Sysetm Offset, and Work Plane Settings

4.3.1 Ending the Previous Operation

You would expect that the NC blocks output at the end of an operation to be output in the *onSectionEnd* function, but in most posts, this is handled in *onSection* and for the final operation, in the *onClose* function. This code will typically stop the spindle, turn off the coolant, and retract the tool.

```
  if (insertToolCall || newWorkOffset || newWorkPlane) {

    // stop spindle before retract during tool change
    if (insertToolCall && !isFirstSection()) {
      onCommand(COMMAND_STOP_SPINDLE);
    }

    // retract to safe plane
    retracted = true;
    writeRetract(Z);
  }
…
…
    onCommand(COMMAND_COOLANT_OFF);

    if (!isFirstSection() && properties.optionalStop) {
      onCommand(COMMAND_OPTIONAL_STOP);
    }
```

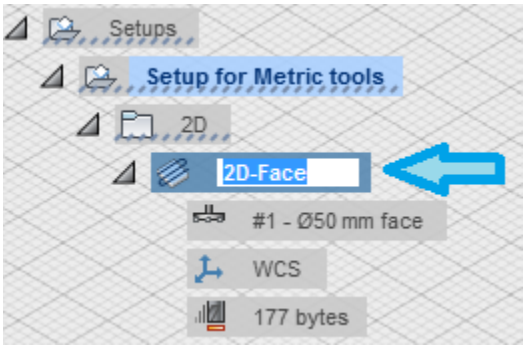
Ending the Previous Operation

The code to retract the tool can vary from post to post, depending on the controller model and the machine configuration. It can output an absolute move to the machine home position, for example using G53, or move to a clearance plane relevant to the current work offset, for example G00 Z5.0.

The *onSectionEnd* section has an example of ending the operation when not done in the *onSection* function.

4.3.2 Operation Comments and Notes

The operation comment is output in the *onSection* function and optionally notes that the user attached to the operation.

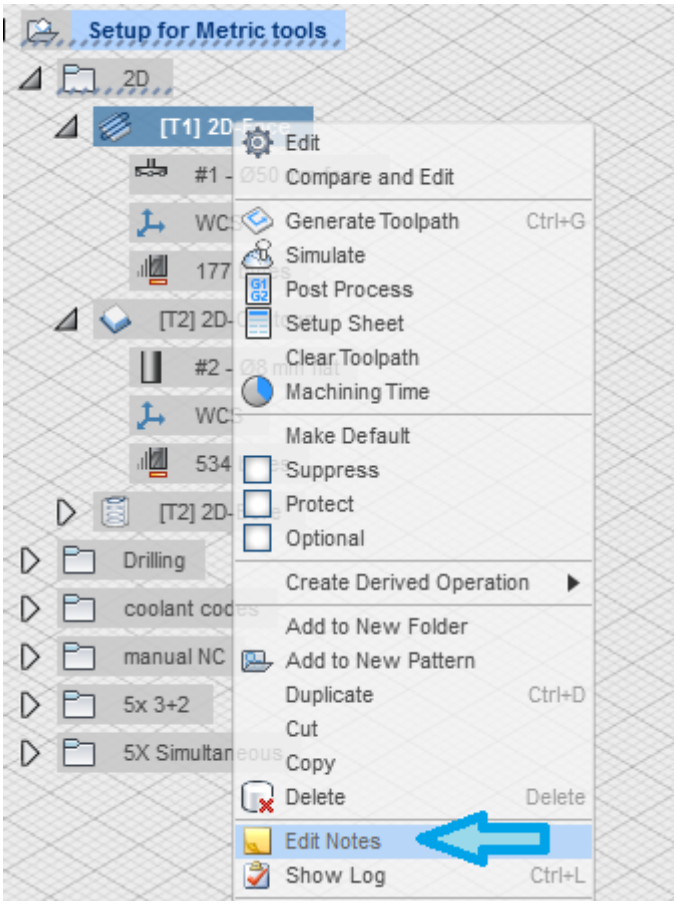


Create Operation Comment

```
  if (hasParameter("operation-comment")) {
    var comment = getParameter("operation-comment");
    if (comment) {
```

```
    writeComment(comment);
  }
}
```

Output Operation Comment



Right Click to Show Menu to Create Operation Notes

The output of the operation notes is normally handled by the post processor property *showNotes*.

```
// user-defined properties
properties = {
…
  showNotes: false, // specifies that operation notes should be output
…
}
```

Define the showNotes Property

```
if (properties.showNotes && hasParameter("notes")) {
    var notes = getParameter("notes");
    if (notes) {
      var lines = String(notes).split("\n");
      var r1 = new RegExp("^[\\s]+", "g");
```

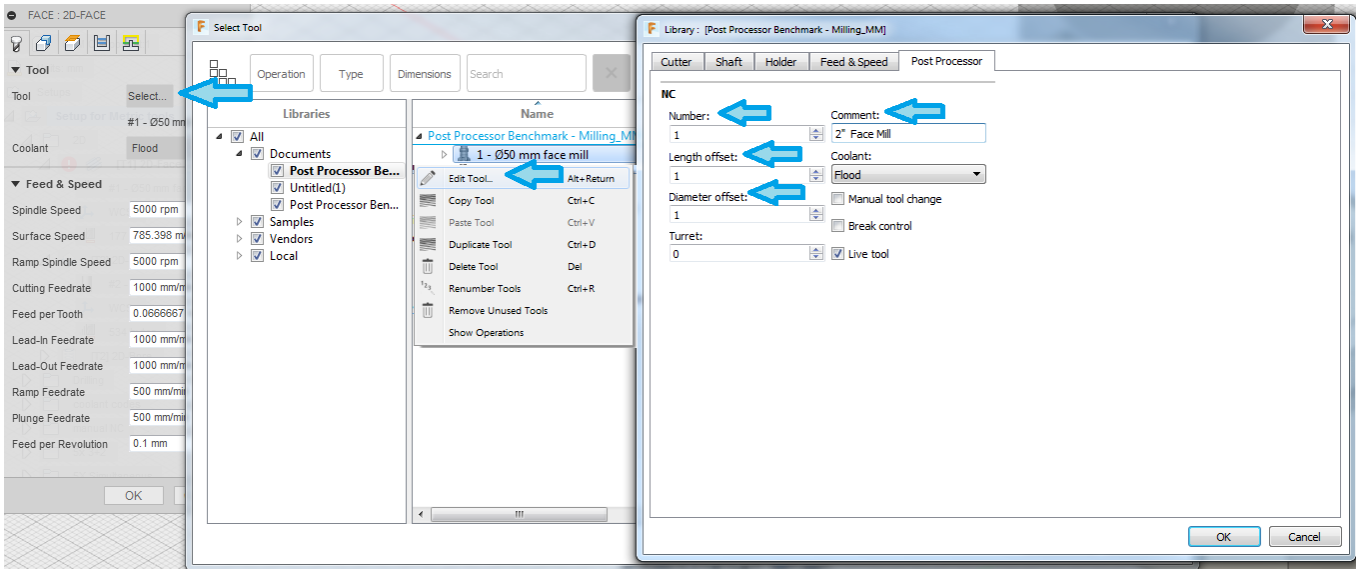
```
    var r2 = new RegExp("[\\s]+$", "g");
    for (line in lines) {
      var comment = lines[line].replace(r1, "").replace(r2, "");
      if (comment) {
        writeComment(comment);
      }
    }
  }
```

Output Operation Notes

4.3.3 Tool Change

Tool change blocks are output whenever a new tool is loaded in the spindle or the tool change is forced, either by a Manual NC *Force tool change* command or internally, for example when a safe start is forced at each operation. The tool change blocks usually contain the following information.

1. Tool number and tool change code
2. Tool comment
3. Comment containing lower Z-limit for tool (optional)
4. Selection of next tool
5. Spindle speed and direction
6. Coolant codes



Tool Parameters Used in Tool Change

The Length Offset value is usually output with the Initial Position as described further in this chapter. The Diameter Offset value is output with a motion block in *onLinear*. All other tool parameters are output in the tool change code.

```
  if (insertToolCall) {
...
```

```
  if (tool.number > numberOfToolSlots) {
    warning(localize("Tool number exceeds maximum value."));
  }

  writeBlock("T" + toolFormat.format(tool.number), mFormat.format(6));
  if (tool.comment) {
    writeComment(tool.comment);
  }
…
```

Output Tool Change and Tool Comment

You will have to change the setting of *showToolZMin* to *true* if you want the lower Z-limit comment output at a tool change.

```
  var showToolZMin = false;  // set to true to enable output of lower Z-limit
  if (showToolZMin) {
    if (is3D()) {
      var numberOfSections = getNumberOfSections();
      var zRange = currentSection.getGlobalZRange();
      var number = tool.number;
      for (var i = currentSection.getId() + 1; i < numberOfSections; ++i) {
        var section = getSection(i);
        if (section.getTool().number != number) {
          break;
        }
        zRange.expandToRange(section.getGlobalZRange());
      }
      writeComment(localize("ZMIN") + "=" + zRange.getMinimum());
    }
  }
```

Output Lower Limit of Z for This Operation

The selection of the next tool is optional and is controlled by the post processor property *preloadTool*.

```
// user-defined properties
properties = {
…
preloadTool: true, // preloads next tool on tool change if any
…
}
```

Define the preloadTool Property

The first tool will be loaded on the last operation of the program.

```
if (properties.preloadTool) {
    var nextTool = getNextTool(tool.number);
```

```
    if (nextTool) {
      writeBlock("T" + toolFormat.format(nextTool.number));
    } else {
      // preload first tool
      var section = getSection(0);
      var firstToolNumber = section.getTool().number;
      if (tool.number != firstToolNumber) {
        writeBlock("T" + toolFormat.format(firstToolNumber));
      }
    }
  }
```

Preload the Next Tool

The spindle codes will be output with a tool change and if the spindle speed changes.

```
if (insertToolCall ||
    isFirstSection() ||
    (rpmFormat.areDifferent(tool.spindleRPM, sOutput.getCurrent())) ||
    (tool.clockwise != getPreviousSection().getTool().clockwise)) {
  if (tool.spindleRPM < 1) {
    error(localize("Spindle speed out of range."));
    return;
  }
  if (tool.spindleRPM > 99999) {
    warning(localize("Spindle speed exceeds maximum value."));
  }
  writeBlock(
    sOutput.format(tool.spindleRPM), mFormat.format(tool.clockwise ? 3 : 4)
  );
}
```

Output Spindle Codes

You will find different methods of outputting the coolant codes in the various posts. The latest method uses a table to define the coolant on and off codes. The table is defined just after the properties table at the top of the post processor. You can define a single code for each coolant mode or multiple codes using an array. When adding or changing the coolant codes supported by your machine, this is the only area of the code that needs to be changed.

```
var singleLineCoolant = false; // specifies to output multiple coolant codes in one line rather than in separate lines
// samples:
// {id: COOLANT_THROUGH_TOOL, on: 88, off: 89}
// {id: COOLANT_THROUGH_TOOL, on: [8, 88], off: [9, 89]}
var coolants = [
  {id: COOLANT_FLOOD, on: 8},
  {id: COOLANT_MIST},
```

```
  {id: COOLANT_THROUGH_TOOL, on: 88, off: 89},
  {id: COOLANT_AIR},
  {id: COOLANT_AIR_THROUGH_TOOL},
  {id: COOLANT_SUCTION},
  {id: COOLANT_FLOOD_MIST},
  {id: COOLANT_FLOOD_THROUGH_TOOL, on: [8, 88], off: [9, 89]},
  {id: COOLANT_OFF, off: 9}
];
```

Coolant Definition Table

The coolant code is output using the following code in *onSection*.

```
  // set coolant after we have positioned at Z
  setCoolant(tool.coolant);
```

Output of Coolant Codes

The *setCoolant* function will output each coolant code in separate blocks. It does this by calling the *getCoolantCodes* function to obtain the coolant code(s) and using *writeBlock* to output each individual coolant code. Both of these functions are generic in nature and should not have to be modified.

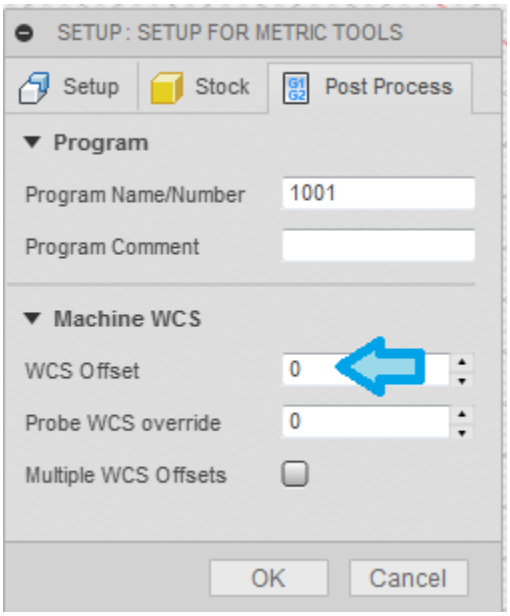
It may be that you want to output the coolant codes(s) in a block with other codes, such as the initial position or the spindle speed. In this case you can call *getCoolantCodes* directly in the *onSection* function and add the output of the coolant codes to the appropriate block. The following example will output the coolant codes with the initial position of the operation.

```
  var coolantCodes = getCoolantCodes(tool.coolant);
  var initialPosition = getFramePosition(currentSection.getInitialPosition());
  writeBlock(
    gAbsIncModal.format(90),
    gMotionModal.format(0),
    xOutput.format(initialPosition.x),
    yOutput.format(initialPosition.y),
    coolantCodes,
  );
```

getCoolantCodes Function Supports Multiple Codes for Single Coolant Mode

4.3.4 Work Coordinate System Offsets

The active Work Coordinate System (WCS) offset is defined in the CAM setup dialog. It defaults to 0 and if you are only using a single WCS, this should be fine as the post processor will convert it to 1. If you are using multiple WCS, then you will need to explicitly define the WCS or the post processor will fail when a default of 0 is used for one setup and a positive number is used for another setup. You can override the WCS defined in the setup in either a folder or pattern.



Define the Work Coordinate System Offset Number

WCS codes are output when a new tool is used for the operation or when the WCS offset number used is changed. WCS offsets are typically controlled using the G54 to G59 codes and possibly an extended syntax for handling work offsets past 6.

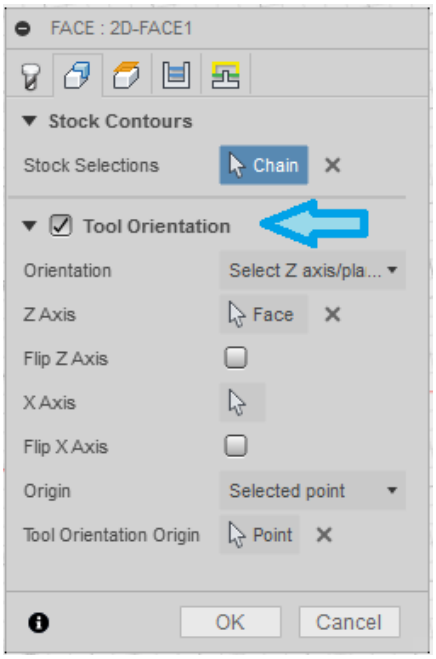
```
// wcs
  if (insertToolCall) { // force work offset when changing tool
    currentWorkOffset = undefined;
  }
  var workOffset = currentSection.workOffset;
  if (workOffset == 0) { // change work offset of 0 to 1
    warningOnce(localize("Work offset has not been specified. Using G54 as WCS."),
WARNING_WORK_OFFSET);
    workOffset = 1;
  }
  if (workOffset > 0) {
    if (workOffset > 6) { // handle work offsets greater than 6
      var code = workOffset - 6;
      if (code > 3) {
        error(localize("Work offset out of range."));
        return;
      }
      if (workOffset != currentWorkOffset) {
        forceWorkPlane();
        writeBlock(gFormat.format(59) + "." + code); // G59.n
        currentWorkOffset = workOffset;
      }
    } else { // handle work offsets 1-6
```

```
    if (workOffset != currentWorkOffset) {
      forceWorkPlane();
      writeBlock(gFormat.format(53 + workOffset)); // G54->G59
      currentWorkOffset = workOffset;
    }
  }
}
```

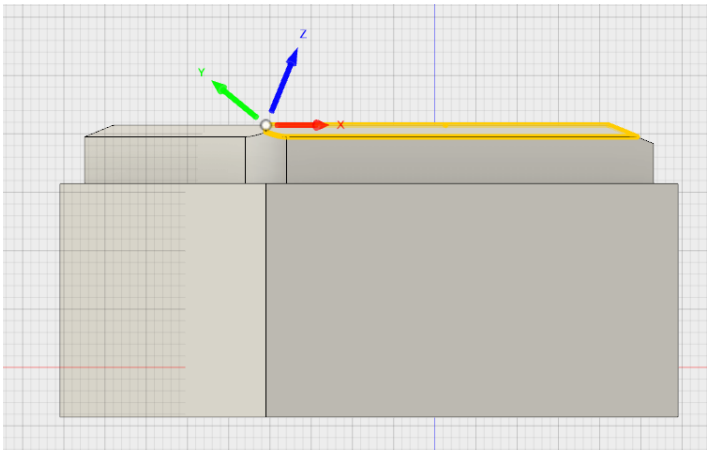
Output the Work Coordinate System Offset Number

4.3.5 Work Plane – 3+2 Operations

3+2 operations are supported by defining a tool orientation for the operation. This tool orientation is referenced as the Work Plane in the post processor. The tool orientation is defined in the Geometry tab of the operation.



Defining the Work Plane



Work Plane for 3+2 Operation

The output for a Work Plane will either be the rotary axes positions or the definition of the Work Plane itself as Euler angles. For machine controls that support both formats the *useMultiAxisFeatures* variable is defined at the top of the post processor to determine the Work Plane method to use.

```
// fixed settings
var useMultiAxisFeatures = false; // false = use rotary axis positions, true = use Euler angles
```

The function *getWorkPlaneMachineABC* is used to calculate the rotary axes positions that satisfy the Work Plane. This function is standard from post to post, but there are a couple of areas that are controlled by user defined settings.

The first step is whether you require the rotary axes positions to be output closest to the angles used for the previous Work Plane. This setting is *false* by default, but you can set it to *true* to enable it. You can also define a starting position for the rotary axes so that if the first operation is a 3+2 operation, then it will choose the closest angles to your starting angles.

```
var closestABC = true; // choose closest machine angles
var currentMachineABC = new Vector(0, 0, 0);  // set initial angles

function getWorkPlaneMachineABC(workPlane, _setWorkPlane, _setRotation) {
```

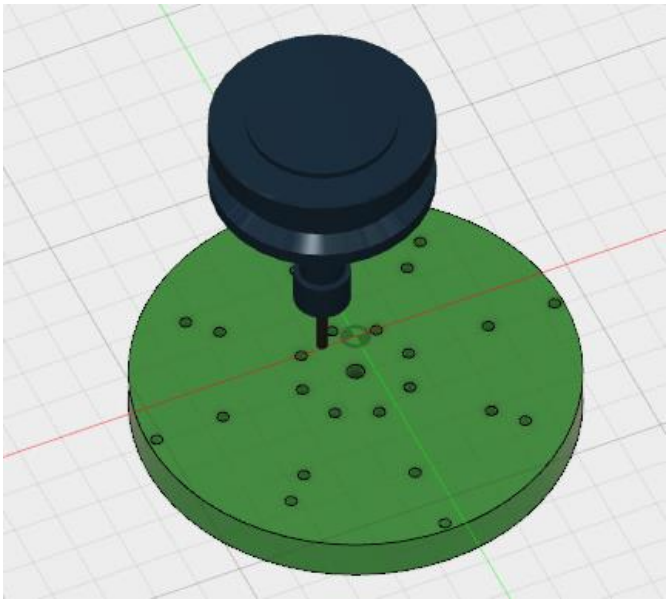
Select the Closest Machine Angles when Defining the Work Plane Orientation

You will also have to define whether Tool Control Point (TCP) programming is supported by the machine or if the tool endpoint coordinates need to be adjusted for the rotary axes positions. You do this by setting the tcp variable to *true* (TCP is supported) or *false* (adjust points for rotary axes).

```
  var tcp = false;
  if (tcp) {
    setRotation(W); // TCP mode
  } else {
    var O = machineConfiguration.getOrientation(abc);
    var R = machineConfiguration.getRemainingOrientation(abc, W);
```

```
  setRotation(R);
}
```

Define the TCP Setting for 3+2 Machining



Tool Perpendicular to Rotary Table

In situations where the tool can be become perpendicular to the rotary table it is not possible to control the rotary table position using the Work Plane. You will notice in this case that the table does not move to satisfy the Work Plane rotation, but rather the output points are rotated to satisfy the Work Plane. You can override this logic and have the rotary table move instead by adding the following code to the TCP setting for 3+2 machining.

```
var tcp = false;
  cancelTransformation();
  if (tcp) {
    setRotation(W); // TCP mode
  } else {
    var O = machineConfiguration.getOrientation(abc);
    var R = machineConfiguration.getRemainingOrientation(abc, W);

    // rotate table if possible to satisfy Work Plane rotation
    var rotate = true;
    var axis = machineConfiguration.getAxisV();
    if (axis.isEnabled() && axis.isTable()) {
      var ix = axis.getCoordinate();
      var rotAxis = axis.getAxis();
      if (isSameDirection(machineConfiguration.getDirection(abc), rotAxis) ||
          isSameDirection(machineConfiguration.getDirection(abc), Vector.product(rotAxis, -1))) {
        var direction = isSameDirection(machineConfiguration.getDirection(abc), rotAxis) ? 1 : -1;
        abc.setCoordinate(ix, Math.atan2(R.right.y, R.right.x) * direction);
```

```
      rotate = false;
    }
  }
  if (rotate) {
    setRotation(R);
  }
}
```

Rotate Table when Tool is Perpendicular to Table

The logic that controls the Work Plane calculation is typically in the *onSection* function, but in some post processors, especially those that support subprograms, you will find this logic in the *defineWorkPlane* function.

```
var abc = new Vector(0, 0, 0);
// use 5-axis indexing for multi-axis mode
if (!is3D() || machineConfiguration.isMultiAxisConfiguration()) {
  //
  if (currentSection.isMultiAxis()) {
    forceWorkPlane();
    cancelTransformation();
  } else {
    // use Euler angles for Work Plane
    if (useMultiAxisFeatures) {
      var eulerXYZ = currentSection.workPlane.getEuler2(EULER_ZXZ_R);
      abc = new Vector(eulerXYZ.x, eulerXYZ.y, eulerXYZ.z);
      cancelTransformation();
    // use rotary axes angles for Work Plane
    } else {
      abc = getWorkPlaneMachineABC(currentSection.workPlane, true, true);
    }
    // output the work plane
    setWorkPlane(abc);
  }
} else { // pure 3D
  var remaining = currentSection.workPlane;
  if (!isSameDirection(remaining.forward, new Vector(0, 0, 1))) {
    error(localize("Tool orientation is not supported."));
    return abc;
  }
  setRotation(remaining);
}
```

Work Plane Calculations

You should be aware that the X-axis direction of the Work Plane does affect the Euler angle calculation. The typical method of defining the Work Plane is to keep the X-axis orientation pointing in the positive direction as you look down the Z-axis, but on some table/table style machines this will cause the

machining to be on the back side of the table, so in this case you will want the X-axis pointing in the negative direction.

The *getEuler2* function is used to calculate the Euler angles for the Work Plane. The argument passed to the *getEuler2* function specifies the order of the primary axis rotations that the machine control requires and can be one of the values in the following table.

Parameter	Parameter	Parameter	Parameter
EULER_XYZ_R	EULER_XYX_R	EULER_XZX_R	EULER_XZY_R
EULER_YXY_R	EULER_YXZ_R	EULER_YZX_R	EULER_YZY_R
EULER_ZXY_R	EULER_ZXZ_R	EULER_ZYX_R	EULER_ZYZ_R
EULER_XYZ_S	EULER_XYX_S	EULER_XZX_S	EULER_XZY_S
EULER_YXY_S	EULER_YXZ_S	EULER_YZX_S	EULER_YZY_S
EULER_ZXY_S	EULER_ZXZ_S	EULER_ZYX_S	EULER_ZYZ_S

Euler Angle Order

Check the Programming Manual for your machine to determine if Euler angles are supported and the order of rotations. The direction of the Euler angles is also important. A general rule to follow is that the directions should match the definition of the rotary axis using the *createAxis* command. If the vector defining the rotation axis is positive for the rotary axis, then the Euler angles will be positive as shown in the above code.

If the rotation axis is in the negative direction, usually when the machine has a rotary head, then the Euler angles should be output in the opposite direction as shown in the following code.

```
    abc = new Vector(-eulerXYZ.x, -eulerXYZ.y, -eulerXYZ.z);
```

Output Euler Angles in Opposite Direction

The *setWorkPlane* function does the actual output of the Work Plane and can vary from post processor to post processor, depending on the requirements of the machine control. It will output the calculated Euler angles or rotary axes positions, and in some cases, both. In the following code, G68.2 is used to define the Work Plane using Euler angles.

```
function setWorkPlane(abc) {
  // the Work Plane does not change, do not output it
  if (!((currentWorkPlaneABC == undefined) ||
        abcFormat.areDifferent(abc.x, currentWorkPlaneABC.x) ||
        abcFormat.areDifferent(abc.y, currentWorkPlaneABC.y) ||
        abcFormat.areDifferent(abc.z, currentWorkPlaneABC.z))) {
    return; // no change
  }

  // unlock rotary axes
  onCommand(COMMAND_UNLOCK_MULTI_AXIS);
```

```
  // output using Euler angles
  if (useMultiAxisFeatures) {
    if (abc.isNonZero()) {
      writeBlock(gFormat.format(68.2), "X" + xyzFormat.format(0), "Y" +
          xyzFormat.format(0), "Z" + xyzFormat.format(0), "A" + abcFormat.format(abc.x),
          "B" + abcFormat.format(abc.y), "C" + abcFormat.format(abc.z));
    // Work Plane is not active
    } else {
      writeBlock(gFormat.format(69)); // cancel frame
    }
  // output rotary axes positions
  } else {
    gMotionModal.reset();
    writeBlock(
      gMotionModal.format(0),
      conditional(machineConfiguration.isMachineCoordinate(0), "A" + abcFormat.format(abc.x)),
      conditional(machineConfiguration.isMachineCoordinate(1), "B" + abcFormat.format(abc.y)),
      conditional(machineConfiguration.isMachineCoordinate(2), "C" + abcFormat.format(abc.z))
    );
  }

  // lock rotary axes
  onCommand(COMMAND_LOCK_MULTI_AXIS);
  currentWorkPlaneABC = abc;
}
```

Output Work Plane in setWorkPlane Function

Some machine controls require that the rotary axes positions be output prior to the Euler angle block. If this is the case, then the code to output the Work Plane can be modified to output both variations of the Work Plane.

```
    if (true && machineConfiguration.isMultiAxisConfiguration()) { // prepositioning for ABC axes
    var angles =
      abc.isNonZero() ? getWorkPlaneMachineABC(currentSection.workPlane, false, false) : abc;
      gMotionModal.reset();
      writeBlock(
        gMotionModal.format(0),
        conditional(machineConfiguration.isMachineCoordinate(0), "A" + abcFormat.format(angles.x)),
        conditional(machineConfiguration.isMachineCoordinate(1), "B" + abcFormat.format(angles.y)),
        conditional(machineConfiguration.isMachineCoordinate(2), "C" + abcFormat.format(angles.z))
      );
    }

    // output Euler angles
    if (abc.isNonZero()) {
```

```
    writeBlock(gFormat.format(68.2), "X" + xyzFormat.format(0), "Y" +
      xyzFormat.format(0), "Z" + xyzFormat.format(0), "A" + abcFormat.format(abc.x),
      "B" + abcFormat.format(abc.y), "C" + abcFormat.format(abc.z));
  // Work Plane is not active
  } else {
    writeBlock(gFormat.format(69)); // cancel frame
  }
```

Output Rotary Axes Positions and Work Plane Euler Angles

4.3.6 Initial Position

The initial position of the operation is available to the *onSection* function and is output here. Tool length compensation on the control is enabled with the initial position when the tool is changed or if it has been disabled between operations.

```
// force all axes to be output at start of operation
forceAny();

// get the initial tool position and retract in Z if necessary
var initialPosition = getFramePosition(currentSection.getInitialPosition());
if (!retracted) {
  if (getCurrentPosition().z < initialPosition.z) {
    writeBlock(gMotionModal.format(0), zOutput.format(initialPosition.z));
  }
}

// output tool length offset on tool change or if tool has been retracted
if (insertToolCall || retracted) {
  var lengthOffset = tool.lengthOffset;
  if (lengthOffset > numberOfToolSlots) {
    error(localize("Length offset out of range."));
    return;
  }

  gMotionModal.reset();
  writeBlock(gPlaneModal.format(17));

  // output XY and then Z with 3-axis or table configuration
  if (!machineConfiguration.isHeadConfiguration()) {
    writeBlock(
      gAbsIncModal.format(90),
      gMotionModal.format(0), xOutput.format(initialPosition.x), yOutput.format(initialPosition.y)
    );
    writeBlock(gMotionModal.format(0), gFormat.format(43), zOutput.format(initialPosition.z),
      hFormat.format(lengthOffset));
  // output XYZ with head configuration
```

```
  } else {
    writeBlock(
      gAbsIncModal.format(90),
      gMotionModal.format(0),
      gFormat.format(43), xOutput.format(initialPosition.x),
      yOutput.format(initialPosition.y),
      zOutput.format(initialPosition.z), hFormat.format(lengthOffset)
    );
  }
// do not activate tool length compensation if already activated
} else {
  writeBlock(
    gAbsIncModal.format(90),
    gMotionModal.format(0),
    xOutput.format(initialPosition.x),
    yOutput.format(initialPosition.y)
  );
}
```

Output Current Position and Tool Length Compensation

4.4 onSectionEnd

```
function onSectionEnd() {
```

The *onSectionEnd* function can be used to define the end of an operation, but in most post processors this is handled in the *onSection* function. The reason for this is that different output will be generated depending on if there is a tool change, WCS change, or Work Plane change and this logic is handled in the *onSection* function (see the *insertToolCall* variable), though it could be handled in the *onSectionEnd* function if desired by referencing the *getNextSection* and *isLastSection* functions.

```
  var insertToolCall = isLastSection() ||
    getNextSection().getForceToolChange && getNextSection().getForceToolChange() ||
    (getNextSection().getTool().number != tool.number);

  var retracted = false; // specifies that the tool has been retracted to the safe plane
  var newWorkOffset = isLastSection() ||
    (currentSection.workOffset != getNextSection().workOffset); // work offset changes
  var newWorkPlane = isLastSection() ||
    !isSameDirection(currentSection.getGlobalFinalToolAxis(),
     getNextSection().getGlobalInitialToolAxis());

  if (insertToolCall || newWorkOffset || newWorkPlane) {
    // stop spindle before retract during tool change
    if (insertToolCall) {
      onCommand(COMMAND_STOP_SPINDLE);
    }
```

```
  // retract to safe plane
  retracted = true;
  writeBlock(gFormat.format(28), gAbsIncModal.format(91), "Z" + xyzFormat.format(0)); // retract
  writeBlock(gAbsIncModal.format(90));
  zOutput.reset();
  if (insertToolCall) {
    onCommand(COMMAND_COOLANT_OFF);

    if (properties.optionalStop) {
      onCommand(COMMAND_OPTIONAL_STOP);
    }
  }
}
```

Ending the Operation in onSectionEnd

You will need to remove the similar code from the *onSection* function and probably the *onClose* function, which will duplicate the session ending code if left intact.

One reason for ending the operation in the *onSectionEnd* function is if a Manual NC command is used between operations. The Manual NC command will be processed prior to the *onSection* function and if the previous operation is terminated in *onSection*, then the Manual NC command will be acted upon prior to ending the previous operation.

The *onSectionEnd* function is pretty basic in most posts and will reset codes that may have been changed in the operation and possibly some variables that are operation specific.

```
function onSectionEnd() {
  writeBlock(gPlaneModal.format(17));
  forceAny();
}
```

Basic onSectionEnd Function

4.5 onClose

```
function onClose() {
```

The *onClose* function is called at the end of the last operation, after *onSectionEnd*. It is used to define the end of an operation, if not handled in *onSectionEnd*, and to output the end-of-program codes.

```
function onClose() {
  // end previous operation
  writeln("");
  optionalSection = false;

  onCommand(COMMAND_COOLANT_OFF);
```

```
  writeRetract(Z); // retract
  disableLengthCompensation(true);
  setSmoothing(false);
  zOutput.reset();
  setWorkPlane(new Vector(0, 0, 0)); // reset working plane
  writeRetract(X, Y); // return to home

  // output end-of-program codes
  onImpliedCommand(COMMAND_END);
  onImpliedCommand(COMMAND_STOP_SPINDLE);
  writeBlock(mFormat.format(30)); // stop program, spindle stop, coolant off
  writeln("%");
}
```

Basic onClose Function

4.6 onTerminate

```
function onTerminate() {
```

The *onTerminate* function is called at the end of post processing, after *onClose*. It is called after all output to the NC file is finished and the NC file is closed. It may be used to rename the output file(s) after processing has finished, to automatically create a setup sheet, or to run another program against the output NC file.

```
function onTerminate() {
  var outputPath = getOutputPath();
  var programFilename = FileSystem.getFilename(outputPath);
  var programSize = FileSystem.getFileSize(outputPath);
  var postPath = findFile("setup-sheet-excel-2007.cps");
  var intermediatePath = getIntermediatePath();
  var a = "--property unit " + ((unit == IN) ? "0" : "1"); // use 0 for inch and 1 for mm
  if (programName) {
    a += " --property programName \"" + programName + "\"";
  }
  if (programComment) {
    a += " --property programComment \"" + programComment + "\"";
  }
  a += " --property programFilename \"" + programFilename + "\"";
  a += " --property programSize \"" + programSize + "\"";
  a += " --noeditor --log temp.log \"" + postPath + "\" \"" + intermediatePath + "\" \"" +
    FileSystem.replaceExtension(outputPath, "xlsx") + "\"";
  execute(getPostProcessorPath(), a, false, "");
  executeNoWait("excel", "\"" + FileSystem.replaceExtension(outputPath, "xlsx") + "\"", false, "");
}
```

Create and Display Setup Sheet from onTerminate

4.7 onCommand

```
function onCommand(command) {
```

Arguments	Description
command	Command to process.

The *onCommand* function can be called by a Manual NC command, directly from HSM, or from the post processor.

Command	Description
COMMAND_ACTIVATE_SPEED_FEED_SYNCHRONIZATION	Activate threading mode
COMMAND_ALARM	Alarm
COMMAND_ALERT	Alert
COMMAND_BREAK_CONTROL	Tool break control
COMMAND_CALIBRATE	Run calibration cycle
COMMAN_CHANGE_PALLET	Change pallet
COMMAND_CLEAN	Run cleaning cycle
COMMAND_CLOSE_DOOR	Close primary door
COMMAND_COOLANT_OFF	Coolant off (M09)
COMMAND_COOLANT_ON	Coolant on (M08)
COMMAND_DEACTIVATE_SPEED_FEED_SYNCHRONIZATION	Deactivate threading mode
COMMAND_END	Program end (M02)
COMMAND_EXACT_STOP	Exact stop
COMMAND_LOAD_TOOL	Tool change (M06)
COMMAND_LOCK_MULTI_AXIS	Locks the rotary axes
COMMAND_MAIN_CHUCK_CLOSE	Close main chuck
COMMAND_MAIN_CHUCK_OPEN	Open main chuck
COMMAND_OPEN_DOOR	Open primary door
COMMAND_OPTIONAL_STOP	Optional program stop (M01)
COMMAND_ORIENTATE_SPINDLE	Orientate spindle (M19)
COMMAND_POWER_OFF	Power off
COMMAND_POWER_ON	Power on
COMMAND_SECONDARY_CHUCK_CLOSE	Close secondary chuck
COMMAND_SECONDARY_CHUCK_OPEN	Open secondary chuck
COMMAND_SECONDARY_SPINDLE_SYNCHRONIZATION_ACTIVATE	Activate spindle synchronization
COMMAND_SECONDARY_SPINDLE_SYNCHRONIZATION_DEACTIVATE	Deactivate spindle synchronization
COMMAND_SPINDLE_CLOCKWISE	Clockwise spindle direction (M03)
COMMAND_SPINDLE_COUNTERCLOCKWISE	Counter-clockwise spindle direction (M04)
COMMAND_START_CHIP_TRANSPORT	Start chip conveyor
COMMAND_START_SPINDLE	Start spindle in previous direction
COMMAND_STOP	Program stop (M00)
COMMAND_STOP_CHIP_TRANSPORT	Stop chip conveyor

Command	Description
COMMAND_STOP_SPINDLE	Stop spindle (M05)
COMMAND_TOOL_MEASURE	Measure tool
COMMAND_UNLOCK_MULTI_AXIS	Unlocks the rotary axes
COMMAND_VERIFY	Verify path/tool/machine integrity

Valid Commands

The Manual NC commands that call *onCommand* are described in the *Manual NC Commands* chapter. Internal calls to *onCommand* are usually generated when expanding a cycle. The post processor itself will call *onCommand* directly to perform simple functions, such as outputting a program stop, cancelling coolant, opening the main door, turning on the chip conveyor, etc.

```
// stop spindle and cancel coolant before retract during tool change
if (insertToolCall && !isFirstSection()) {
  onCommand(COMMAND_COOLANT_OFF);
  onCommand(COMMAND_STOP_SPINDLE);
}
```

Calling onCommand Directly from Post Processor

The *onImpliedCommand* function changes the state of certain settings in the post engine without calling *onCommand* and outputting the associated codes with the command. The state of certain parameters is important when the post processor engine expands cycles.

```
onImpliedCommand(COMMAND_END);
onImpliedCommand(COMMAND_STOP_SPINDLE);
onImpliedCommand(COMMAND_COOLANT_OFF);
writeBlock(mFormat.format(30)); // stop program, spindle stop, coolant off
```

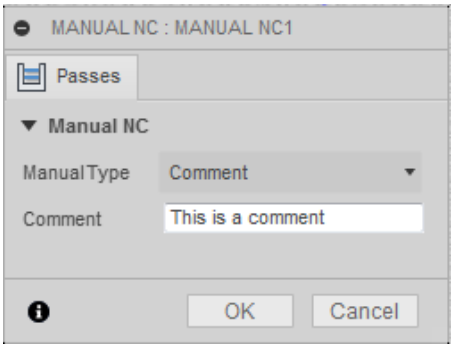
Using onImpliedCommand

4.8 onComment

```
function onComment(message) {
```

Arguments	Description
message	Text of comment to output.

The *onComment* function is called when the Manual NC command *Comment* is issued. It will format and output the text of the comment to the NC file.



The Comment Manual NC Command

There are two other functions that are used to format and output comments, *formatComment* and *writeComment*. These comment functions are standard in nature and do not typically have to be modified, though the *permittedCommentChars* variable, defined at the top of the post, is used to define the characters that are allowed in a comment and may have to be changed to match the control. The *formatComment* function will remove any characters in the comment that are not specified in this variable. Lowercase letters will be converted to uppercase by the *formatComment* function. If you want to support lowercase letters, then they would have to be added to the *permittedCommentChars* variable and the *formatComment* function would need to have the conversion to uppercase removed.

```
var permittedCommentChars = " ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789.,=_-";
```

Defining the Permitted Characters for Comments

```
/** Format a comment */
function formatComment(text) {
  return "(" + filterText(String(text).toUpperCase(), permittedCommentChars).replace(/[\(\)]/g, "") +
")";
}

/**   Output a comment */
function writeComment(text) {
  writeln(formatComment(text));
}

/** Process the Manual NC Comment command */
function onComment(message) {
  var comments = String(message).split(";"); // allow multiple lines of comments per command
  for (comment in comments) {
    writeComment(comments[comment]);
  }
}
```

The Comment Functions

4.9 onDwell

```
function onDwell(seconds) {
```

Arguments	Description
seconds	Dwell time in seconds.

The *onDwell* function can be called by a Manual NC command, directly from HSM, or from the post processor. The Manual NC command that calls *onDwell* is described in the *Manual NC Commands* chapter. Internal calls to *onDwell* are usually generated when expanding a cycle. The post processor itself will call *onDwell* directly to output a dwell block.

```
function onDwell(seconds) {
  if (seconds > 99999.999) {
    warning(localize("Dwelling time is out of range."));
  }
  milliseconds = clamp(1, seconds * 1000, 99999999);
  writeBlock(gFeedModeModal.format(94), gFormat.format(4), "P" +
    milliFormat.format(milliseconds));
}
```

Output the Dwell Time in Milliseconds

```
  onCommand(COMMAND_COOLANT_ON);
  onDwell(1.0);  // dwell 1 second after turning coolant on
```

Calling onDwell Directly from Post Processor

4.10 onParameter

```
function onParameter(name, value) {
```

Arguments	Description
name	Parameter name.
value	Value stored in the parameter.

Almost all parameters used for creating a machining operation in HSM are passed to the post processor. Common parameters are available using built in post processor variables (currentSection, tool, cycle, etc.) as well as being made available as parameters. Other parameters are passed to the *onParameter* function.

```
74: onParameter('operation:context', 'operation')
75: onParameter('operation:strategy', 'drill')
76: onParameter('operation:operation_description', 'Drill')
77: onParameter('operation:tool_type', 'tap right hand')
78: onParameter('operation:undercut', 0)
79: onParameter('operation:tool_isTurning', 0)
80: onParameter('operation:tool_isMill', 0)
81: onParameter('operation:tool_isDrill', 1)
82: onParameter('operation:tool_taperedType', 'tapered_bull_nose')
83: onParameter('operation:tool_unit', 'inches')
```

```
84: onParameter('operation:tool_number', 4)
85: onParameter('operation:tool_diameterOffset', 4)
86: onParameter('operation:tool_lengthOffset', 4)
```

Sample Parameters Passed to the onParameter Function from Dump Post Processor

The name of the parameter along with its value is passed to the *onParameter* function. Some Manual NC commands will call the *onParameter* function, these are described in the *Manual NC Commands* chapter. You can see how to run and analyze the output from the *dump.cps* post processor in the *Debugging* chapter.

```
function onParameter(name, value) {
  if (name == "probe-output-work-offset") {
    probeOutputWorkOffset = (value > 0) ? value : 1;
  }
}
```

Sample onParameter Function

4.10.1 getParameter Function

```
value = getParameter(name)
```

Arguments	Description
name	Parameter name.

You can retrieve operation parameters at any place in the post processor by calling the *getParameter* function. Operation parameters are defined as parameters that are redefined for each machining operation. There is a chance that a parameter does not exist in all flavors of HSM, so it is recommended that the presence of the parameter is first verified by calling the *hasParameter* function.

```
  if (hasParameter("operation-comment")) {
    var comment = getParameter("operation-comment");
    if (comment) {
      writeComment(comment);
    }
  }
```

Verify a Parameter Exists Using the hasParameter Function

When scanning through the operations in the intermediate file it is possible to access the parameters for that operation by using the section variant of the *hasParameter* and *getParameter* functions.

```
  // write out all operation comments
  writeln("List of Operations:");
  for (var i = 0; i < getNumberOfSections(); ++i) {
    var section = getSection(i);
    if (section.hasParameter("operation-comment") {
      var comment = section.getParameter("operation-comment");
```

```
    if (comment) {
      writeln("  " + comment);
    }
  }
}
writeln("");
```

Using Section Variant of getParameter

4.10.2 getGlobalParameter Function

```
value = getGlobalParameter(name)
```

Arguments	Description
name	Parameter name.

Some parameters are defined at the start of the intermediate file prior to the first operation. These parameters are considered global and are accessed using the *hasGlobalParameter* and *getGlobalParameter* functions. The same rules that apply to the operation parameters apply to global parameters.

```
-1: onOpen()
0: onParameter('product-id', 'fusion360')
1: onParameter('generated-by', 'Fusion 360 CAM 2.0.3803')
2: onParameter('generated-at', 'Saturday, March 24, 2018 4:34:36 PM')
3: onParameter('hostname', 'host')
4: onParameter('username', 'user')
5: onParameter('document-path', 'Water-Laser-Plasma v2')
6: onParameter('leads-supported', 1)
7: onParameter('job-description', 'Laser')
9: onParameter('stock', '((0, 0, -5), (300, 200, 0))')
11: onParameter('stock-lower-x', 0)
13: onParameter('stock-lower-y', 0)
15: onParameter('stock-lower-z', -5)
17: onParameter('stock-upper-x', 300)
19: onParameter('stock-upper-y', 200)
21: onParameter('stock-upper-z', 0)
23: onParameter('part-lower-x', 0)
25: onParameter('part-lower-y', 0)
27: onParameter('part-lower-z', -5)
29: onParameter('part-upper-x', 300)
31: onParameter('part-upper-y', 200)
33: onParameter('part-upper-z', 0)
35: onParameter('notes', '')
```

Sample Global Variables

When processing multiple setups at the same time some of the global parameters will change from one setup to the next. The *getGlobalParameter* function though will always reference the parameters of the first setup, so if you want to access the parameters of the active setup then you will need to use the *onParameter* function rather than the *getGlobalParameter* function.

```
function onParameter(name, value) {
  if (name == "job-description") {
    setupName = value;
  }
}
```

Using onParameter to Store the Active Setup Name

4.11 onPassThrough

```
Function onPassThrough (value)
```

Arguments	Description
value	Text to be output to the NC file.

The *onPassThrough* function is called by the *Pass through* Manual NC command and is used to pass a text string directly to the NC file without any processing by the post processor. This function is described in the Manual NC Commands chapter.

4.12 onSpindleSpeed

```
function onSpindleSpeed(speed) {
```

Arguments	Description
spindleSpeed	The new spindle speed in RPM.

The *onSpindleSpeed* function is used to output changes in the spindle speed during an operation, typically from the post processor engine when expanding a cycle.

```
function onSpindleSpeed(spindleSpeed) {
  writeBlock(sOutput.format(spindleSpeed));
}
```

Sample onSpindleSpeed Function

4.13 onOrientateSpindle

```
function onOrientateSpindle(angle) {
```

Arguments	Description
angle	Spindle orientation angle in radians.

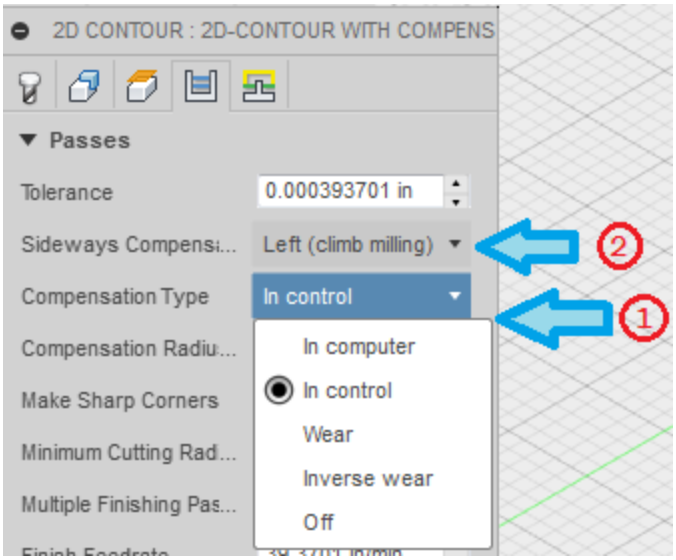
The *onOrientateSpindle* function is not typically called. When a cycle that orientates the spindle is expanded the *onCommand(COMMAND_ORIENTATE_SPINDLE)* function is called.

4.14 onRadiusCompensation

```
function onRadiusCompensation() {
```

The *onRadiusCompensation* function is called when the radius (cutter) compensation mode changes. It will typically set the pending compensation mode, which will be handled in the motion functions (*onRapid*, *onLinear*, *onCircular*, etc.). Radius compensation, when enabled in an operation, will be enabled on the move approaching the part and disabled after moving off the part.

The state of radius compensation is stored in the global *radiusCompensation* variable and is not passed to the onRadiusCompensation function. Radius compensation is defined when creating the machining operation in HSM (1). The Sideways Compensation (2) setting determines the side of the part that the tool will be on when cutting. It is based on the forward direction of the tool during the cutting operation.



Enabling/Disabling Radius Compensation

Compensation Type	Description
In computer	The tool is offset from the part based on the tool diameter. The center line of the offset tool is sent to the post processor and the radius compensation mode is OFF (G40).
In control	The tool is not offset from the part. The centerline of the tool as if it is on the part is sent to the post processor and the radius compensation mode is determined by the Sideways Compensation setting (G41/G42). The control will perform the entire offsetting of the tool.
Wear	The tool is offset from the part based on the tool diameter. The center line of the offset tool is sent to the post processor and the radius compensation mode is determined by the Sideways Compensation setting (G41/G42). The control will compensate for tool wear.

Compensation Type	Description
Inverse wear	Same as Wear, but the opposite compensation direction will be used (G42/G41).
Off	The tool is not offset from the part. The centerline of the tool as if it is on the part is sent to the post processor and the radius compensation mode will be disabled (G40).

Radius Compensation Modes

```
var pendingRadiusCompensation = -1;

function onRadiusCompensation() {
  pendingRadiusCompensation = radiusCompensation;
}
```

Sample onRadiusCompensation Function

4.15 onMovement

```
function onMovement(movement) {
```

Arguments	Description
movement	Movement type for the following motion(s).

onMovement is called whenever the movement type changes. It is used to tell the post when there is a positioning, entry, exit, or cutting type move. There is also a *movement* global variable that contains the movement setting. This variable can be referenced directly in other functions, such as *onLinear*, to access the movement type without defining the *onMovement* function.

The supported movement types are listed in the following table.

Movement Type	Description
MOVEMENT_CUTTING	Standard cutting motion.
MOVEMENT_EXTENDED	Extended movement type. Not common.
MOVEMENT_FINISH_CUTTING	Finish cutting motion.
MOVEMENT_HIGH_FEED	Movement at high feedrate. Not typically used. Rapid moves output using a linear move at the high feedrate will use the MOVEMENT_RAPID type.
MOVEMENT_LEAD_IN	Lead-in motion.
MOVEMENT_LEAD_OUT	Lead-out motion.
MOVEMENT_LINK_DIRECT	Direction (non-cutting) linking move.
MOVEMENT_LINK_TRANSITION	Transition (cutting) linking move.
MOVEMENT_PLUNGE	Plunging move.
MOVEMENT_PREDRILL	Predrilling motion.
MOVEMENT_RAMP	Ramping entry motion.
MOVEMENT_RAMP_HELIX	Helical ramping motion.
MOVEMENT_RAMP_PROFILE	Profile ramping motion.

Movement Type	Description
MOVEMEN_RAMP_ZIG_ZAG	Zig-Zag ramping motion.
MOVEMENT_RAPID	Rapid movement.
MOVEMENT_REDUCED	Reduced cutting motion.

Movement Types

Movement types are used in defining parametric feedrates in some milling posts and for removing all non-cutting moves for waterjet/plasma/laser machines that require only the cutting profile.

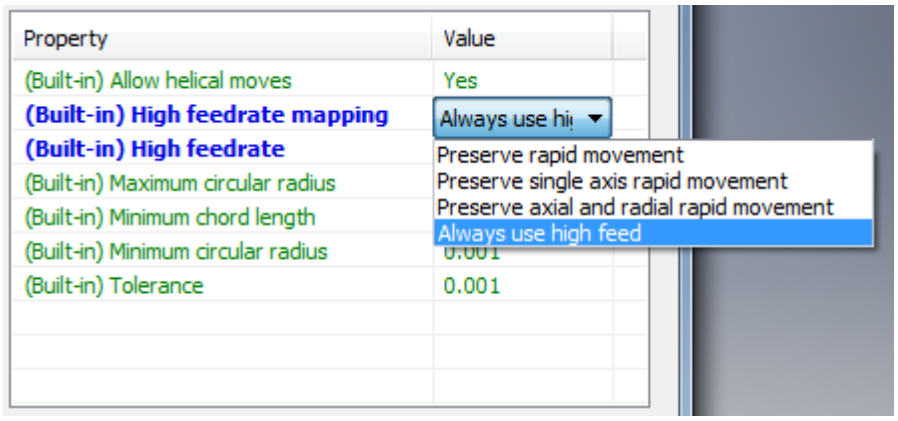
4.16 onRapid

```
function onRapid(_x, _y, _z) {
```

Arguments	Description
_x, _y, _z	The tool position.

The *onRapid* function handles rapid positioning moves (G00) while in 3-axis mode. The tool position is passed as the *_x, _y, _z* arguments. The format of the onRapid function is pretty basic, it will handle a change in radius compensation, may determine if the rapid moves should be output at a high feedrate (due to the machine making dogleg moves while in rapid mode), and output the rapid move to the NC file.

If the *High feedrate mapping* property is set to *Always use high feed*, then the *onLinear* function will be called with the high feedrate passed in as the feedrate and the *onRapid* function will not be called.



Using High Feedrates for Positioning Moves

```
function onRapid(_x, _y, _z) {
  // format tool position for output
  var x = xOutput.format(_x);
  var y = yOutput.format(_y);
  var z = zOutput.format(_z);

  // ignore if tool does not move
```

```
  if (x || y || z) {
    if (pendingRadiusCompensation >= 0) { // handle radius compensation
      error(localize("Radius compensation mode cannot be changed at rapid traversal."));
      return;
    }

    // output move at high feedrate if movement in more than one axis
    if (!properties.useG0 && (((x ? 1 : 0) + (y ? 1 : 0) + (z ? 1 : 0)) > 1)) {
      writeBlock(gFeedModeModal.format(94), gMotionModal.format(1), x, y, z,
        getFeed(highFeedrate));

    // output move in rapid mode
    } else {
      writeBlock(gMotionModal.format(0), x, y, z);
      forceFeed();
    }
  }
}
```

Sample onRapid Function

4.17 onExpandedRapid

```
onExpandedRapid(x, y, z);
```

Arguments	Description
x, y, z	The tool position.

It is possible that the post processor will need to generate rapid positioning moves during the processing of the intermediate file. An example would be creating your own expanded drilling cycle. Instead of calling *onRapid* with the post generated moves, it is recommended that *onExpandedRapid* be called instead. This will ensure that the post engine is notified of the move and the current position is set. *onExpandedRapid* will then call *onRapid* with the provided arguments.

The *onExpandedRapid* function is not considered an entry function, since it will never be called directly by the post processor engine.

4.18 onLinear

```
function onLinear(_x, _y, _z, feed) {
```

Arguments	Description
_x, _y, _z	The tool position.
feed	The feedrate.

The *onLinear* function handles linear moves (G01) at a feedrate while in 3-axis mode. The tool position is passed as the *_x, _y, _z* arguments. The format of the *onLinear* function is pretty basic, it will handle a change in radius compensation and outputs the linear move to the NC file.

```
function onLinear(_x, _y, _z, feed) {
  // force move when radius compensation changes
  if (pendingRadiusCompensation >= 0) {
    xOutput.reset();
    yOutput.reset();
  }

  // format tool position for output
  var x = xOutput.format(_x);
  var y = yOutput.format(_y);
  var z = zOutput.format(_z);
  var f = getFeed(feed);

  // ignore if tool does not move
  if (x || y || z) {
    // handle radius compensation changes
    if (pendingRadiusCompensation >= 0) {
      pendingRadiusCompensation = -1;
      var d = tool.diameterOffset;
      if (d > 200) {
        warning(localize("The diameter offset exceeds the maximum value."));
      }
      writeBlock(gPlaneModal.format(17));
      switch (radiusCompensation) {
      case RADIUS_COMPENSATION_LEFT:
        dOutput.reset();
        writeBlock(gFeedModeModal.format(94), gMotionModal.format(1), gFormat.format(41), x, y, z,
          dOutput.format(d), f);
        break;
      case RADIUS_COMPENSATION_RIGHT:
        dOutput.reset();
        writeBlock(gFeedModeModal.format(94), gMotionModal.format(1), gFormat.format(42), x, y, z,
          dOutput.format(d), f);
        break;
      default:
        writeBlock(gFeedModeModal.format(94), gMotionModal.format(1), gFormat.format(40), x, y, z,
          f);
      }
    // output non-compensation change move at feedrate
    } else {
      writeBlock(gFeedModeModal.format(94), gMotionModal.format(1), x, y, z, f);
    }
```

```
  // no movement, but feedrate changes
  } else if (f) {
    if (getNextRecord().isMotion()) { // try not to output feed without motion
      forceFeed(); // force feed on next line
    } else {
      writeBlock(gFeedModeModal.format(94), gMotionModal.format(1), f);
    }
  }
}
```

Sample onLinear Function

4.19 onExpandedLinear

```
onExpandedLinear(x, y, z, feed);
```

Arguments	Description
_x, _y, _z	The tool position.
feed	The feedrate.

It is possible that the post processor will need to generate cutting moves during the processing of the intermediate file. An example would be creating your own expanded drilling cycle. Instead of calling *onLinear* with the post generated moves, it is recommended that onExpandedLinear be called instead. This will ensure that the post engine is notified of the move and the current position is set. *onExpandedLinear* will then call *onLinear* with the provided arguments.

The *onExpandedLinear* function is not considered an entry function, since it will never be called directly by the post processor engine.

4.20 onRapid5D

```
function onRapid5D(_x, _y, _z, _a, _b, _c) {
```

Arguments	Description
_x, _y, _z	The tool position.
_a, _b, _c	The rotary angles if a machine configuration has been defined, otherwise the tool axis vector is passed.

The *onRapid5D* function handles rapid positioning moves (G00) in multi-axis operations. The tool position is passed as the *_x, _y, _z* arguments and the rotary angles as the *_a, _b, _c* arguments. If a machine configuration has not been defined, then *_a, _b, _c* contains the tool axis vector. The *onRapid5D* function will be called for all rapid moves in a multi-axis operation, even if the move is only a 3-axis linear move without rotary movement.

Like the *onRapid* function, the *onRapid5D* function handles a change in radius compensation, may determine if the rapid moves should be output at a high feedrate (due to the machine making dogleg moves while in rapid mode), and outputs the rapid move to the NC file.

```
function onRapid5D(_x, _y, _z, _a, _b, _c) {
  // enable this code if machine does not accept IJK tool axis vector input
  if (false) {
    if (!currentSection.isOptimizedForMachine()) {
      error(localize("This post configuration has not been customized for 5-axis toolpath."));
      return;
    }
  }

  // handle radius compensation changes
  if (pendingRadiusCompensation >= 0) {
    error(localize("Radius compensation mode cannot be changed at rapid traversal."));
    return;
  }

  // Machine Configuration has been defined, output rotary angles with move
  if (currentSection.isOptimizedForMachine()) {
    var x = xOutput.format(_x);
    var y = yOutput.format(_y);
    var z = zOutput.format(_z);
    var a = aOutput.format(_a);
    var b = bOutput.format(_b);
    var c = cOutput.format(_c);
    writeBlock(gMotionModal.format(0), x, y, z, a, b, c);
  // Machine Configuration has not been defined, output tool axis with move
  } else {
    forceXYZ();
    var x = xOutput.format(_x);
    var y = yOutput.format(_y);
    var z = zOutput.format(_z);
    var i = ijkFormat.format(_a);
    var j = ijkFormat.format(_b);
    var k = ijkFormat.format(_c);
    writeBlock(gMotionModal.format(0), x, y, z, "I" + i, "J" + j, "K" + k);
  }
  forceFeed();
}
```

Sample onRapid5D Function

Please refer to the *Multi-Axis Post Processors* chapter for a detailed explanation on supporting a multi-axis machine.

4.21 onLinear5D

```
function onLinear5D(_x, _y, _z, _a, _b, _c, feed) {
```

Arguments	Description
_x, _y, _z	The tool position.
_a, _b, _c	The rotary angles if a machine configuration has been defined, otherwise the tool axis vector is passed.
feed	The feedrate.

The *onLinear5D* function handles cutting moves (G01) in multi-axis operations. The tool position is passed as the *_x, _y, _z* arguments and the rotary angles as the *_a, _b, _c* arguments. If a machine configuration has not been defined, then *_a, _b, _c* contains the tool axis vector. The *onLinear5D* function will be called for all cutting moves in a multi-axis operation, even if the move is only a 3-axis linear move without rotary movement.

Like the *onLinear* function, the *onLinear5D* function handles a change in radius compensation, and outputs the cutting move to the NC file.

```
function onLinear5D(_x, _y, _z, _a, _b, _c, feed) {
  // enable this code if machine does not accept IJK tool axis vector input
  if (false) {
    if (!currentSection.isOptimizedForMachine()) {
      error(localize("This post configuration has not been customized for 5-axis toolpath."));
      return;
    }
  }

  // handle radius compensation changes
  if (pendingRadiusCompensation >= 0) {
    error(localize("Radius compensation cannot be activated/deactivated for 5-axis move."));
    return;
  }

  // Machine Configuration has been defined, output rotary angles with move
  if (currentSection.isOptimizedForMachine()) {
    var x = xOutput.format(_x);
    var y = yOutput.format(_y);
    var z = zOutput.format(_z);
    var a = aOutput.format(_a);
    var b = bOutput.format(_b);
    var c = cOutput.format(_c);

    // calculate multi-axis feedrate
    var f = {frn:0, fmode:0};
    if (a || b || c) {
      f = getMultiaxisFeed(_x, _y, _z, _a, _b, _c, feed);
    } else {
      f.frn = getFeed(feed);
      f.fmode = 94;
```

```
    }

    // ignore if tool does not move
    if (x || y || z || a || b || c) {
      writeBlock(gMotionModal.format(1), x, y, z, a, b, c, f);
    } else if (f) {
      if (getNextRecord().isMotion()) { // try not to output feed without motion
        forceFeed(); // force feed on next line
      } else {
        writeBlock(gMotionModal.format(1), f);
      }
    }

  // Machine Configuration has not been defined, output tool axis with move
  } else {
    forceXYZ();
    var x = xOutput.format(_x);
    var y = yOutput.format(_y);
    var z = zOutput.format(_z);
    var i = ijkFormat.format(_a);
    var j = ijkFormat.format(_b);
    var k = ijkFormat.format(_c);
    var f = getFeed(feed);

    // ignore if tool does not move
    if (x || y || z || i || j || k) {
      writeBlock(gMotionModal.format(1), x, y, z, "I" + i, "J" + j, "K" + k, f);
    } else if (f) {
      if (getNextRecord().isMotion()) { // try not to output feed without motion
        forceFeed(); // force feed on next line
      } else {
        writeBlock(gMotionModal.format(1), f);
      }
    }
  }
}
```

Sample onLinear5D Function

Please refer to the *Multi-Axis Post Processors* chapter for a detailed explanation on supporting a multi-axis machine.

4.22 onCircular

```
function onCircular(clockwise, cx, cy, cz, x, y, z, feed) {
```

Argument	Description
clockwise	Set to *true* if the circular direction is in the clockwise direction, *false* if counter-clockwise.
cx, cy, cz	Center coordinates of circle.
x, y, z	Final point on circle
feed	The feedrate.

The *onCircular* function is called whenever there is circular, helical, or spiral motion. The circular move can be in any of the 3 standard planes, XY-plane, YZ-plane, or ZX-plane, it is up to the *onCircular* function to determine which types of circular are valid for the machine and to correctly format the output.

The structure of the *onCircular* function in most posts uses the following layout.

1. Test for radius compensation. Most controls do not allow radius compensation to be started on a circular move.
2. Full circle output.
3. Center point (IJK) output.
4. Radius output.

Each of the different styles of output will individually handle the output of circular interpolation in each of the planes and possibly 3-D circular interpolation if it is supported.

```
  if (pendingRadiusCompensation >= 0) {  // Disallow radius compensation
    error(localize("Radius compensation cannot be activated/deactivated for a circular move."));
    return;
  }
…
  if (isFullCircle()) { // Full 360 degree circles
    if (properties.useRadius || isHelical()) { // radius mode does not support full arcs
      linearize(tolerance);
      return;
    }
…
  } else if (!properties.useRadius) {  // Incremental center point output
    switch (getCircularPlane()) {
    case PLANE_XY:
…
  } else { // Use radius mode
    var r = getCircularRadius();
    if (toDeg(getCircularSweep()) > (180 + 1e-9)) {
      r = -r; // allow up to <360 deg arcs
    }
…
```

Standard onCircular Structure

```
switch (getCircularPlane()) {
case PLANE_XY:
  writeBlock(gPlaneModal.format(17), gMotionModal.format(clockwise ? 2 : 3),
    xOutput.format(x), yOutput.format(y), zOutput.format(z),
    iOutput.format(cx - start.x, 0), jOutput.format(cy - start.y, 0), getFeed(feed));
  break;
case PLANE_ZX:
  writeBlock(gPlaneModal.format(18), gMotionModal.format(clockwise ? 2 : 3),
    xOutput.format(x), yOutput.format(y), zOutput.format(z),
    iOutput.format(cx - start.x, 0), kOutput.format(cz - start.z, 0), getFeed(feed));
  break;
case PLANE_YZ:
  writeBlock(gPlaneModal.format(19), gMotionModal.format(clockwise ? 2 : 3),
    xOutput.format(x), yOutput.format(y), zOutput.format(z),
    jOutput.format(cy - start.y, 0), kOutput.format(cz - start.z, 0), getFeed(feed));
  break;
default: // circular record is not in major plane
  linearize(tolerance);
}
```

Circular Output Based on Plane

4.22.1 Circular Interpolation Settings

There are settings that affect how circular interpolation is handled in the post engine, basically telling the post engine when to call *onCircular* or when to linearize the points by calling *onLinear* multiple times instead. The following table describes the circular interpolation settings.

Setting	Description
allowedCircularPlanes	Defines the standard planes that circular interpolation is allowed in, PLANE_XY, PLANE_YZ, PLANE_ZX. It can be set to *undefined* to allow circular interpolation in all three planes, 0 to disable circular interpolation, or a bit mask of PLANE_XY, PLANE_YZ, and/or PLANE_YZ to allow only certain planes.
allowHelicalMoves	Helical interpolation is allowed when this variable is set to *true*. Helical moves are linearized if set to *false*.
allowSpiralMoves	Spiral interpolation is defined as circular moves that have a different starting radius than ending radius and can be enabled by setting this variable to *true*. Spiral moves are linearized if set to *false*.
maximumCircularRadius	Specifies the maximum radius of circular moves that can be output as circular interpolation and can be changed dynamically in the Property table when running the post processor. Any circular records whose radius exceeds this value will be linearized. This variable must be set in millimeters (MM). maximumCircularRadius = spatial(1000, MM); // 39.37 inch

Setting	Description
maximumCircularSweep	Specifies the maximum angular sweep of circular moves that can be output as circular interpolation and is specified in radians. Any circular records whose delta angle exceeds this value will be linearized.
minimumChordLength	Specifies the minimum delta movement allowed for circular interpolation and can be changed dynamically in the Property table when running the post processor. Any circular records whose delta linear movement is less than this value will be linearized. This variable must be set in millimeters (MM).
minimumCircularRadius	Specifies the minimum radius of circular moves that can be output as circular interpolation and can be changed dynamically in the Property table when running the post processor. Any circular records whose radius is less than this value will be linearized. This variable must be set in millimeters (MM).
minimumCircularSweep	Specifies the minimum angular sweep of circular moves that can be output as circular interpolation and is specified in radians. Any circular records whose delta angle is less than this value will be linearized.
tolerance	Specifies the tolerance used to linearize circular moves that are expanded into a series of linear moves. Circular interpolation records can be linearized due to the conditions of the circular interpolation settings not being met or by the *linearize* function being called. This variable must be set in millimeters (MM).

Circular Interpolation Settings

```
allowedCircularPlanes = undefined; // allow all circular planes
allowedCircularPlanes = 0; // disable all circular planes
allowedCircularPlanes = (1 << PLANE_XY) | (1 << PLANE_ZX); // XY, ZX planes

tolerance = spatial(0.002, MM); // linearization tolerance of .00008 IN
minimumChordLength = spatial(0.01, MM); // minimum linear movement of .0004 IN
minimumCircularRadius = spatial(0.01, MM); // minimum circular radius of .0004 IN
maximumCircularRadius = spatial(1000, MM); // maximum circular radius of 39.37 IN
minimumCircularSweep = toRad(0.01); // minimum angular movement of .01 degrees
maximumCircularSweep = toRad(180); // circular interpolation up to 180 degrees
allowHelicalMoves = true; // enable helical interpolation
allowSpiralMoves = false; // disallow spiral interpolation
```

Example Circular Interpolation Settings

4.22.2 Circular Interpolation Common Functions

There are built-in functions that are utilized by the *onCircular* function. These functions return values used in the *onCircular* function, determine if the circular record should be linearized, and control the flow of the *onCircular* function logic.

Function	Description
getCircularCenter()	Returns the center point of the circle as a Vector.
getCircularChordLength()	Returns the delta linear movement of the circular interpolation record.
getCircularNormal()	Returns the normal of the circular plane as a Vector. The normal is flipped if the circular movement is in the clockwise direction. This follows the righthand plane convention.
getCircularPlane()	Returns the plane of the circular interpolation record, PLANE_XY, PLANE_ZX, or PLANE_YZ. If the return value is -1, then the circular plane is not a major plane, but is in 3-D space.
getCircularRadius()	Returns the end radius of the circular motion.
getCircularStartRadius()	Returns the start radius of the circular motion. This will be different than the end radius for spiral moves.
getCircularSweep()	Returns the angular sweep of the circular interpolation record in radians.
getCurrentPosition()	Returns the starting point of the circular move as a Vector.
getHelicalDistance()	Returns the distance the third axis will move during helical interpolation. Returns 0 for a 2-D circular interpolation record.
getHelicalOffset()	Returns the distance along the third axis as a Vector. This function is used when helical interpolation is supported outside one of the three standard circular planes.
getHelicalPitch()	Returns the distance that the third axis travels for a full 360-degree sweep, i.e. the pitch value of the thread.
getPositionU(u)	Returns the point on the circle at *u* percent along the arc as a Vector.
isFullCircle()	Returns *true* if the angular sweep of the circular motion is 360 degrees.
isHelical()	Returns *true* if the circular interpolation record contains helical movement. The variable *allowHelicalMoves* must be set to *true* for helical records to be passed to the *onCircular* function.
isSpiral()	Returns true if the circular interpolation record contains spiral movement (the start and end radii are different). The variable *allowSpiralMoves* must be set to *true* for spiral records to be passed to the *onCircular* function.
linearize(tolerance)	Linearizes the circular motion by outputting a series of linear moves.

onCircular Common Functions

4.22.3 Helical Interpolation

Helical interpolation is defined as circular interpolation with movement along the third linear axis. The third linear axis is defined as the axis that is not part of the circular plane, for example, the Z-axis is the third linear axis for circular interpolation in the XY-plane. The variable *allowHelicalMoves* must be set to true for the post processor to support helical interpolation.

Helical interpolation is typically output using the same format as circular interpolation with the addition of the third axis and optionally a pitch value (incremental distance per 360 degrees) for the third axis.

Most stock post processors are already setup to output the third axis with circular interpolation (it won't be output for a 2-D circular move).

```
case PLANE_XY:
  writeBlock(gPlaneModal.format(17), gMotionModal.format(clockwise ? 2 : 3),
    xOutput.format(x), yOutput.format(y), zOutput.format(z),
    iOutput.format(cx-start.x, 0), jOutput.format(cy-start.y, 0), kOutput.format(getHelicalPitch()),
    feedOutput.format(feed));
  break;
```

Helical Interpolation with Pitch Output

4.22.4 Spiral Interpolation

Spiral interpolation is defined as circular interpolation that has a different radius at start of the circular move than the radius at the end of the move. The variable *allowSpiralMoves* must be set to true for the post processor to support helical interpolation.

Spiral interpolation when supported on a control is typically specified with a G-code different than the standard G02/G03 circular interpolation G-codes. Most stock post processors do not support spiral interpolation.

```
if (isSpiral()) {
  var startRadius = getCircularStartRadius();
  var endRadius = getCircularRadius();
  var dr = Math.abs(endRadius - startRadius);
  if (dr > maximumCircularRadiiDifference) { // maximum limit
    if (isHelical()) { // not supported
      linearize(tolerance);
      return;
    }

    switch (getCircularPlane()) {
    case PLANE_XY:
      writeBlock(gPlaneModal.format(17), gMotionModal.format(clockwise ? 2.1 : 3.1),
        xOutput.format(x), yOutput.format(y), zOutput.format(z),
        iOutput.format(cx - start.x, 0), jOutput.format(cy - start.y, 0), getFeed(feed));
      break;
    case PLANE_ZX:
      writeBlock(gPlaneModal.format(18), gMotionModal.format(clockwise ? 2.1 : 3.1),
        xOutput.format(x), yOutput.format(y), zOutput.format(z),
        iOutput.format(cx - start.x, 0), kOutput.format(cz - start.z, 0), getFeed(feed));
      break;
    case PLANE_YZ:
      writeBlock(gPlaneModal.format(19), gMotionModal.format(clockwise ? 2.1 : 3.1),
        xOutput.format(x), yOutput.format(y), zOutput.format(z),
        jOutput.format(cy - start.y, 0), kOutput.format(cz - start.z, 0), getFeed(feed));
```

```
      break;
    default:
      linearize(tolerance);
    }
    return;
  }
}
```

Spiral Interpolation Output

4.22.5 3-D Circular Interpolation

3-D circular interpolation is defined as circular interpolation that is not on a standard circular plane (XY, ZX, YZ).

3-D circular interpolation when supported on a control is typically specified with a G-code different than the standard G02/G03 circular interpolation G-codes and must contain either the mid-point of the circular move and/or the normal vector of the circle. Most stock post processors do not support 3-D circular interpolation.

```
default:
  if (properties.allow3DArcs) {  // a post property is used to enable support of 3-D circular
    // make sure maximumCircularSweep is well below 360deg
    var ip = getPositionU(0.5); // calculate mid-point of circle
    writeBlock(gMotionModal.format(clockwise ? 2.4 : 3.4),  // 3-D circular direction G-codes
      xOutput.format(ip.x), yOutput.format(ip.y), zOutput.format(ip.z), // output mid-point of circle
      getFeed(feed));
    writeBlock(xOutput.format(x), yOutput.format(y), zOutput.format(z)); // output end-point
  } else {
    linearize(tolerance);
  }
}
```

3-D Circular Interpolation Output

4.23 onCycle

```
function onCycle() {
```

The *onCycle* function is called once at the beginning of an operation that contains a canned cycle and can contain code to prepare the machine for the cycle. Mill post processors will typically set the machining plane here.

```
function onCycle() {
  writeBlock(gPlaneModal.format(17));
}
```

Sample onCycle Function

Mill/Turn post processors will usually handle the stock transfer sequence in the onCycle function. Logic for the Mill/Turn post processors will be discussed in a dedicated chapter.

4.24 onCyclePoint

```
function onCyclePoint(x, y, z) {
```

Argument	Description
x, y, z	Hole bottom location.

Canned cycle output is handled in the *onCyclePoint* function, which includes positioning to the clearance plane, formatting of the cycle block, calculating the cycle parameters, discerning if the canned cycle is supported on the machine or should be expanded, and probing cycles which will not be discussed in this chapter.

The location of the hole bottom for the cycle is passed in as the *x, y, z* arguments to the *onCyclePoint* function. All other parameters are available in the *cycle* object or through cycle specific function calls. The flow of outputting canned cycles usually follows the following logic.

1. First hole location in cycle
    a. Position to clearance plane
    b. Canned cycle is supported on machine
        i. Calculate common cycle parameters
        ii. Format and output canned cycle
    c. Canned cycle is not supported on machine
        i. Expand cycle into linear moves
2. 2nd through nth holes
    a. Cycle is not expanded
        i. Output hole location
    b. Cycle is expanded
        i. Expand cycle at new location

The actual output of the cycle blocks is handled in a *switch* block, with a separate *case* for each of the supported cycles.

```
switch (cycleType) {
  case "drilling":
    writeBlock(
      gRetractModal.format(98), gAbsIncModal.format(90), gCycleModal.format(81),
      getCommonCycle(x, y, z, cycle.retract),
      feedOutput.format(F)
    );
    break;
```

Sample Cycle Formatting Code

If a cycle is not supported and needs to be expanded by the post engine, then you can either remove the entire case block for this cycle and it will be handled in the default block, or you can specifically expand the cycle. The second method is handy when the canned cycle does not support all of the parameters available in HSM, for example if a dwell is not supported for a deep drilling cycle on the machine, but you want to be able to use a dwell.

```
case "deep-drilling":
  if (P > 0) {  // the machine does not support a dwell code, so expand the cycle
    expandCyclePoint(x, y, z);
  } else {
    writeBlock(
      gRetractModal.format(98), gAbsIncModal.format(90), gCycleModal.format(83),
      getCommonCycle(x, y, z, cycle.retract),
      "Q" + xyzFormat.format(cycle.incrementalDepth),
      feedOutput.format(F)
    );
  }
  break;
```

Expanding a Cycle When a Feature is not Support on the Machine

The 2nd through the nth locations in a cycle operation are typically output using simple XY moves without any of the cycle definition codes. Expanded cycles still need to be expanded at these locations.

```
} else {  // end of isFirstCyclePoint() condition
  if (cycleExpanded) {
    expandCyclePoint(x, y, z);
  } else {
    var _x = xOutput.format(x);
    var _y = yOutput.format(y);
    if (!_x && !_y) {
      xOutput.reset(); // at least one axis is required
      _x = xOutput.format(x);
    }
    writeBlock(_x, _y);
  }
}
```

Output the 2nd through nth Cycle Locations

4.24.1 Drilling Cycle Types

The following table contains the drilling (hole making cycles). The cycle type is stored in the *cycleType* variable as a text string. The standard G-code used for the cycle is included in the description, where expanded defines the cycle as usually not being supported on the machine and expanded instead.

cycleType	Description
drilling	Feed in to depth and rapid out (G81)

cycleType	Description
counter-boring	Feed in to depth, dwell, and rapid out (G82)
chip-breaking	Multiple pecks with periodic partial retract to clear chips (G73)
deep-drilling	Peck drilling with full retraction at end of each peck (G83)
break-through-drilling	Allows for reduced speed and feed before breaking through hole (expanded)
gun-drilling	Guided deep drilling allows for a change in spindle speed for positioning (expanded)
tapping	Feed in to depth, reverse spindle, optional dwell, and feed out. Automatically determines left or right tapping depending on the tool selected. (G74/G84)
left-tapping	Left-handed tapping (G74)
right-tapping	Right-handed tapping (G84)
tapping-with-chip-breaking	Tapping with multiple pecks. Automatically determines left or right tapping depending on the tool selected. (expanded)
reaming	Feed in to depth and feed out (G85)
boring	Feed in to depth, dwell, and feed out (G86)
stop-boring	Feed to depth, stop the spindle, and feed out (G87)
fine-boring	Feed to depth, orientate the spindle, shift from wall, and rapid out (G76)
back-boring	Orientate the spindle, rapid to depth, start spindle, shift the tool to wall, feed up to bore height, orientate spindle, shift from wall, and rapid out (G77)
circular-pocket-milling	Mills out a hole (expanded)
thread-milling	Helical thread cutting (expanded)

Types of Drilling Cycles

Any of these cycles can be expanded if the machine control does not support the specific cycle. There are some caveats, where the post (and machine) must support certain capabilities for the expanded cycle to run correctly on the machine. The following table lists the commands that must be defined in the *onCommand* function to support the expansion of these cycles. It is expected that the machine will support these features if they are enabled in the post processor.

cycleType	Supported onCommand Command
tapping left-tapping right-tapping tapping-with-chip-breaking	COMMAND_SPINDLE_CLOCKWISE COMMAND_SPINDLE_COUNTERCLOCKWISE COMMAND_ACTIVATE_SPEED_FEED_SYNCHRONIZATION COMMAND_DEACTIVATE_SPEED_FEED_SYNCHRONIZATION
stop-boring	COMMAND_STOP_SPINDLE COMMAND_START_SPINDLE
fine-boring back-boring	COMMAND_STOP_SPINDLE COMMAND_START_SPINDLE COMMAND_ORIENTATE_SPINDLE

Required Command Support for Expanded Cycles

Certain cycles also use the following parameters when they are expanded.

machineParameters.	Description
drillingSafeDistance	Specifies the safety distance above the stock when repositioning into the hole for the *chip-breaking* and *deep-drilling* cycles.
spindleSpeedDwell	Dwell in seconds after the spindle speed changes during a cycle.

Parameters for Expanded Cycles

You define the expanded cycle parameters using the following syntaxes.

```
machineParameters.drillingSafeDistance = toPreciseUnit(0.1, MM);
machineParameters.spindleSpeedDwell = 1.5;
```

Defining Expanded Cycles Parameters

4.24.2 Cycle parameters

The parameters defined in the cycle operation are passed to the cycle functions using the *cycle* object. The following variables are available and are referenced as 'cycle.*parameter*'.

Parameter	Description
accumulatedDepth	The depth of the combined cuts before the tool will be fully retracted during a *chip-breaking* cycle.
backBoreDistance	The cutting distance of a *back-boring* cycle.
bottom	The bottom of the hole.
breakThroughDistance	The distance above the hole bottom to switch to the break-through feedrate and spindle speed during a *break-through-drilling* cycle.
breakThroughFeedRate	The feedrate used when breaking through the hole during a *break-through-drilling* cycle.
breakThroughSpindleSpeed	The spindle speed used when breaking through the hole during a *break-through-drilling* cycle.
chipBreakDistance	The distance to retract the tool to break the chip during a *chip-breaking* cycle.
clearance	Clearance plane where to tool will retract the tool to after drilling a hole and position to the next hole.
compensation	Radius compensation in effect for *circular-pocket-milling* and *thread-milling* cycles. This value can be *control*, *wear*, and *inverseWear*.
compensationShiftOrientation	Same as *shiftOrientation*.
depth	The depth of the hole.
diameter	The diameter of the hole for *circular-pocket-milling* and *thread-milling* cycles.
direction	Either *climb* or *conventional* milling for *circular-pocket-milling* and *thread-milling* cycles.
dwell	The dwell time in seconds.
dwellDepth	The distance above the cut depth at which to dwell, used for *gun-drilling* cycles.
feedrate	The primary cutting feedrate.

Parameter	Description
incrementalDepth	The incremental pecking depth of the first cut.
incrementalDepthReduction	The incremental pecking depth reduction per cut for pecking cycles.
minimumIncrementalDepth	The minimum pecking depth of cut for pecking cycles.
numberOfSteps	The number of horizontal passes for the *thread-milling* cycle.
plungeFeedrate	The cutting feedrate. The same as *feedrate*.
plungesPerRetract	The number of cuts before the tool will be fully retracted during a *chip-breaking* cycle.
postioningFeedrate	The feedrate used to position the tool during a *gun-drilling* cycle.
positioningSpindleSpeed	The spindle speed used when positioning the tool during a *gun-drilling* cycle.
repeatPass	Set to true if the final pass should be repeated for *circular-pocket-milling* and *thread-milling* cycles.
retract	The plane where the tool will position to prior to starting the cycle (feeding into the hole).
retractFeedrate	The tool retraction feedrate, used when feeding out of the hole.
shift	The distance to shift the tool away from the wall during a *fine-boring* and *back-boring* cycle.
shiftDirection	The direction in radians to shift the tool away from the wall during a *fine-boring* and *back-boring* cycle. The shift direction will be PI radians (180 degrees) from the wall plus this amount.
shiftOrientation	The spindle orientation of the tool in radians when shifting the tool away from the wall during a *fine-boring* or *back-boring* cycle.
stepover	The horizontal stepover distance for *circular-pocket-milling* and *thread-milling* cycles.
stock	The top of the hole.
stopSpindle	When set to *1*, the spindle will be stopped during positioning/retracting in a *gun-drilling* cycle.
threading	Either *right* or *left*-handed threading for *thread-milling* cycles.

Cycle Parameters

4.24.3 The Cycle Planes/Heights

The drilling cycles use different heights during the execution of the cycle. These heights are specified in the Heights tab for the Drilling operation. One thing you should keep in mind is that the names given to these heights do not match the cycle parameter names in the post processor. The following table gives the relationship between the HSM height names and the equivalent *cycle* parameter names.

Operation Heights Tab	Cycle Parameter	Description
Clearance Height	(none)	The plane to position to the first point of the cycle and to retract the tool to after the final point of the cycle.
Retract Height	cycle.clearance	The tool rapids to this plane from the clearance height

Operation Heights Tab	Cycle Parameter	Description
		and will position between the holes at this height.
Feed Height	cycle.retract	The tool will feed from this plane into the hole.
Top Height	cycle.stock	The top of the hole.
Bottom Height	cycle.bottom	The bottom of the hole. This height is the plane where the tool will drill to and will be different from the actual bottom of the hole if the *Drill tip through bottom* box is checked.

Correlation Between Cycle Operation Heights and Cycle Parameters

HSM assumes that the tool will always be retracted to the Retract Height (*cycle.clearance*) between holes, you will notice this in the simulation of the cycle in HSM. This is typically handled in the machine control with a G98 (Retract to clearance plane) code. Of course this code can be different from machine control to machine control and there are controls that will always retract to the Feed Height (*cycle.retract*) at the end of a drilling operation. In this case it is up to the post processor to retract the tool to the Retract Height.

You can cancel the cycle at the end of the *onCyclePoint* function and output a tool retract block to take the tool back up to the Retract Height. When this method is used it is also mandatory that the full cycle be output for every hole in the operation and not just the first cycle point. Some machines support a retract plane to be specified with the cancel cycle code, i.e. G80 Rxxx.

```
function onCyclePoint(x, y, z) {
  // if (isFirstCyclePoint()) {
  if (true) { // output a full cycle block for every hole in the operation
    repositionToCycleClearance(cycle, x, y, z);
...
...
    default:
      expandCyclePoint(x, y, z);
    }
    // retract tool (add at the end of the cycleType switch code)
    gMotionModal.format.reset();
    writeBlock(gCycleModal.format(80), gMotionModal.format(0), zOutput.format(cycle.clearance));
  } else {
    if (cycleExpanded) {
```

Retracting the Tool to the Retract Plane when Unsupported by Machine Control

4.24.4 Common Cycle Functions

There are functions that are commonly used in the *onCyclePoint* function. The following table lists these functions.

Function	Description
isFirstCyclePoint()	Returns *true* if this is the first point in the cycle operation. It is usually called to determine whether to output a full cycle block or just the cycle location.
isLastCyclePoint()	Returns *true* if this is the last point in the cycle operation. This function is typically used for a lathe threading operation since HSM sends a single pass to the *onCyclePoint* function and the full depth of the thread is required to output a single threading block. *onCycleEnd* is used to terminate a drilling cycle, so this function is not typically used in drilling cycles.
isProbingCycle()	Returns *true* if this is a probing cycle.
repositionToCycleClearance()	Moves the tool to the Retract Height plane (*cycle.clearance*). This function is typically called prior to outputting a full cycle block.
getCommonCycle(x, y, z, r)	Formats the common cycle parameters (X, Y, Z, R) for output.

Common Cycle Functions

These functions are built into the post engine, except the *getCommonCycle* function, which is contained in the post processor. It takes the cycle location (x, y, z) and the retract plane/distance (r) as arguments. Some machines require that the retract value be programmed as a distance from the current location rather than as an absolute position. There are two ways to accomplish this. You can pass in the distance as the retract value.

```
function getCommonCycle(x, y, z, r) {
  forceXYZ();
  return [xOutput.format(x), yOutput.format(y),
    zOutput.format(z),
    "R" + xyzFormat.format(r)];
}
…
    case "drilling":
      writeBlock(
        gRetractModal.format(98), gAbsIncModal.format(90), gCycleModal.format(81),
        getCommonCycle(x, y, z, cycle.retract – cycle.clearance),
        feedOutput.format(F)
      );
      break;
```

Pass Retract Distance to Standard getCommonCycle Function

Or you can pass the clearance plane in to the *getCommonCycle* function and have it calculate the distance. This method is typically used in post processors that support subprograms that require a retract plane while in absolute mode and a distance when in incremental mode.

```
function getCommonCycle(x, y, z, r, c) {
  forceXYZ(); // force xyz on first drill hole of any cycle
  if (incrementalMode) {
    zOutput.format(c);
    return [xOutput.format(x), yOutput.format(y),
      "Z" + xyzFormat.format(z - r),
      "R" + xyzFormat.format(r - c)];
  } else {
    return [xOutput.format(x), yOutput.format(y),
      zOutput.format(z),
      "R" + xyzFormat.format(r)];
  }
}
…
    case "drilling":
      writeBlock(
        gRetractModal.format(98), gCycleModal.format(81),
        getCommonCycle(x, y, z, cycle.retract, cycle.clearance),
        feedOutput.format(F)
      );
      break;
```

Pass Retract and Clearance Heights to getCommonCycle Function

4.24.5 Pitch Output with Tapping Cycles

Tapping cycles can be sometimes be output with a standard FPM feedrate, sometimes with a thread pitch, and sometimes using the FPR feedrate mode. There are different variables and formats involved depending on the format used. When using pitch or FPR feedrates you will need to create a format for this style of feedrate. The format is defined at the top of the post processor with the rest of the format definitions. Refer to the *Format Defintions* and *Output Variable Definitions* sections.

```
var feedFormat = createFormat({decimals:(unit == MM ? 0 : 1), forceDecimal:true});
var pitchFormat = createFormat({decimals:(unit == MM ? 3 : 4), forceDecimal:true});
…
var feedOutput = createVariable({prefix:"F"}, feedFormat);
var pitchOutput = createVariable({prefix:"F", force:true}, pitchFormat);
```

Create the Pitch Output Format

In the tapping sections of the *onCyclePoint* function you will need to assign the correct pitch value to the output. The tapping pitch is stored in the *tool.threadPitch* variable.

```
    case "tapping":
      writeBlock(
        gRetractModal.format(98), gCycleModal.format((84),
        getCommonCycle(x, y, z, cycle.retract),
        (conditional(P > 0, "P" + milliFormat.format(P)),
```

```
      pitchOutput.format(tool.threadPitch)
    );
    forceFeed(); // force the feedrate to be output after a tapping cycle with pitch output
    break;
```

Output the Thread Pitch on a Tapping Cycle

If the tapping cycle requires that the machine be placed in FPR mode, then you can also calculate the pitch value by dividing the feedrate by the spindle speed. You will also need to output the FPR code (G95) with the tapping cycle and reset it at the end of the tapping operation, usually in the *onCycleEnd* function.

```
case "tapping":
    var F = cycle.feedrate / spindleSpeed;
    writeBlock(
      gRetractModal.format(98), gFeedModeModal.format(95), gCycleModal.format((84),
      getCommonCycle(x, y, z, cycle.retract),
      (conditional(P > 0, "P" + milliFormat.format(P)),
      pitchOutput.format(F)
    );
    forceFeed(); // force the feedrate to be output after a tapping cycle with pitch output
    break;
```

Output the Feedrate as FPR on a Tapping Cycle

4.25 onCycleEnd

```
function onCycleEnd() {
```

The *onCycleEnd* function is called after all points in the cycle operation have been processed. The cycle is cancelled in this function and the feedrate mode (FPM) is reset if it is a tapping operation that uses FPR feedrates.

```
function onCycleEnd() {
  if (!cycleExpanded) {
    writeBlock(gCycleModal.format(80));
    // writeBlock(gFeedModeModal.format(94)), gCycleModal.format(80)); //  reset FPM mode
    zOutput.reset();
  }
}
```

onCycleEnd Function

4.26 onRewindMachine

```
function onRewindMachine(_a, _b, _c) {
```

Argument	Description
_a, _b, _c	Rotary axes rewind positions.

The *onRewindMachine* function is used to reposition the rotary axes when a machine limit is reached. It is described in detail in the *Rewinding of the Rotary Axis when Limits are Reached* section of this manual.

4.27 Common Functions

There are functions that are common in most of the generic posts. Some of these functions are used in conjuction with other post processor functions and are described in the appropriate section of this manual, for example the *formatComment* function is described with the *onComment* function. This section describes the common functions that are generic in nature and used throughout the post processor.

4.27.1 writeln

```
writeln(text);
```

Arguments	Description
text	Text to output to the NC file

The *writeln* function is built into the post engine and is not defined in the post processor. It is used to output text to the NC file without formatting it. Text can be a quoted text string or a text expression. *writeln* is typically used for outputting text strings that don't require formatting, or debug messages.

```
writeln("%"); // outputs '%'
writeln("Vector = " + new Vector(x, y, z));  // outputs the x, y, z variables in vector format
writeln("");  // outputs a blank line
writeln(formatComment("Load tool " + tool.number + " in spindle");
  // outputs 'Load tool n in spindle' as a comment
```

Sample writeln Calls

4.27.2 writeBlock

```
function writeBlock(arguments) {
```

Arguments	Description
arguments	Comma separated list of codes/text to output.

The *writeBlock* function writes a block of codes to the output NC file. It will add a sequence number to the block, if sequence numbers are enabled and add an optional skip character if this is an optional operation. A list of formatted codes and/or text strings are passed to the *writeBlock* function. The code list is separated by commas, so that each code is passed as an individual argument, which allows for the codes to be separated by the word separator defined by the *setWordSeparator* function.

```
/**
```

```
  Writes the specified block.
*/
function writeBlock() {
  var text = formatWords(arguments);
  if (!text) {
    return;
  }
  if (properties.showSequenceNumbers) {  // add sequence numbers to output blocks
    if (optionalSection) {
      if (text) {
        writeWords("/", "N" + sequenceNumber, text);
      }
    } else {
      writeWords2("N" + sequenceNumber, text);
    }
    sequenceNumber += properties.sequenceNumberIncrement;
  } else { // no sequence numbers
    if (optionalSection) {
      writeWords2("/", text);
    } else {
      writeWords(text);
    }
  }
}
```

Sample writeBlock Function

```
writeBlock(gAbsIncModal.format(90), xFormat.format(x), yFormat.format(y));
writeBlock("G28", "X" + xFormat.format(0), "Y" + yFormat.format(0));  // outputs 'G28 X0 Y0'
writeBlock("G28" + "X" + xFormat.format(0) + "Y" + yFormat.format(0));  // outputs 'G28 X0Y0'
```

Sample writeBlock Calls

The *writeBlock* function does not usually have to be modified.

4.27.3 toPreciseUnit

```
toPreciseUnit(value, units);
```

Arguments	Description
value	The input value.
units	The units that the value is given in, either MM or IN.

The *toPreciseUnit* function allows you to specify a value in a given units and that value will be returned in the active units of the input intermediate CNC file. When developing a post processor, it is highly recommended that any unit based hard coded numbers use the *toPreciseUnit* function when defining the number.

```
yAxisMinimum = toPreciseUnit(gotYAxis ? -50.8 : 0, MM); // minimum range for the Y-axis
yAxisMaximum = toPreciseUnit(gotYAxis ? 50.8 : 0, MM); // maximum range for the Y-axis
xAxisMinimum = toPreciseUnit(0, MM); // maximum range for the X-axis (radius mode)
```

Defining Values using toPreciseUnit

4.27.4 force---

The *force* functions are used to force the output of the specified axes and/or feedrate the next time they are supposed to be output, even if it has the same value as the previous value.

Function	Description
forceXYZ	Forces the output of the linear axes (X, Y, Z) on the next motion block.
forceABC	Forces the output of the rotary axes (A, B, C) on the next motion block.
forceFeed	Forces the output of the feedrate on the next motion block.
forceAny	Forces all axes and the feedrate on the next motion block.

Force Functions

```
/** Force output of X, Y, and Z on next output. */
function forceXYZ() {
  xOutput.reset();
  yOutput.reset();
  zOutput.reset();
}

/** Force output of A, B, and C on next output. */
function forceABC() {
  aOutput.reset();
  bOutput.reset();
  cOutput.reset();
}

/** Force output of F on next output. */
function forceFeed() {
  currentFeedId = undefined;
  feedOutput.reset();
}

/** Force output of X, Y, Z, A, B, C, and F on next output. */
function forceAny() {
  forceXYZ();
  forceABC();
  forceFeed();
}
```

Sample Force Functions

4.27.5 writeRetract

```
function writeRetract(arguments) {
```

Arguments	Description
arguments	X, Y, and/or Z. Separated by commas when multiple axes are specified.

The *writeRetract* function is used to retract the Z-axis to its clearance plane and move the X and Y axes to their home positions.

The *writeRetract* function can be called with one or more axes to move to their home position. The axes are specified using their standard names of *X, Y, Z,* and are separated by commas if multiple axes are specified in the call to *writeRetract*.

```
writeRetract(Z); // move the Z-axis to its home position
writeRetract(X, Y); // move the X and Y axes to their home positions
```

Sample writeRetract Calls

The writeRetract function is not generic in nature and may have to be changed to match your machine's requirements. For example, some machines use a G28 to move an axis to its home position, some will use a G53 with the home position, and some use a standard G00 block.

```
/** Output block to do safe retract and/or move to home position. */
function writeRetract() {
  // initialize routine
  var _xyzMoved = new Array(false, false, false);
  var _useG28 = properties.useG28; // can be either true or false

  // check syntax of call
  if (arguments.length == 0) {
    error(localize("No axis specified for writeRetract()."));
    return;
  }
  for (var i = 0; i < arguments.length; ++i) {
    if ((arguments[i] < 0) || (arguments[i] > 2)) {
      error(localize("Bad axis specified for writeRetract()."));
      return;
    }
    if (_xyzMoved[arguments[i]]) {
      error(localize("Cannot retract the same axis twice in one line"));
      return;
    }
    _xyzMoved[arguments[i]] = true;
  }

  // special conditions
  if (_useG28 && _xyzMoved[2] && (_xyzMoved[0] || _xyzMoved[1])) { // XY don't use G28
```

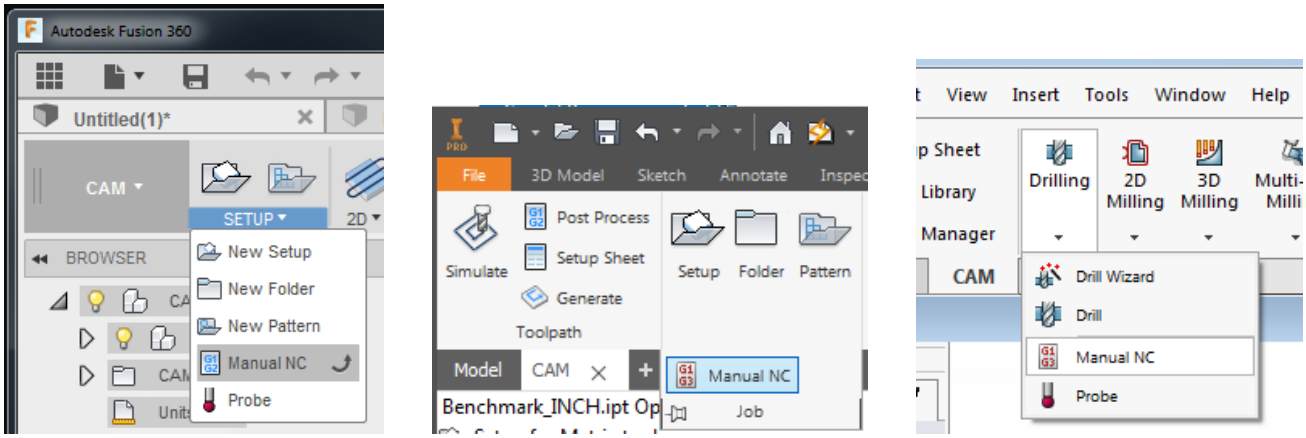
```
    error(localize("You cannot move home in XY & Z in the same block."));
    return;
  }
  if (_xyzMoved[0] || _xyzMoved[1]) {
    _useG28 = false;
  }

  // define home positions
  var _xHome;
  var _yHome;
  var _zHome;
  if (_useG28) {
    _xHome = 0;
    _yHome = 0;
    _zHome = 0;
  } else {
    if (properties.homePositionCenter &&
      hasParameter("part-upper-x") && hasParameter("part-lower-x")) {
      _xHome = (getParameter("part-upper-x") + getParameter("part-lower-x")) / 2;
    } else {
      _xHome = machineConfiguration.hasHomePositionX() ?
machineConfiguration.getHomePositionX() : 0;
    }
    _yHome = machineConfiguration.hasHomePositionY() ?
machineConfiguration.getHomePositionY() : 0;
    _zHome = machineConfiguration.getRetractPlane();
  }

  // format home positions
  var words = []; // store all retracted axes in an array
  for (var i = 0; i < arguments.length; ++i) {
    // define the axes to move
    switch (arguments[i]) {
    case X:
      // special conditions
      if (properties.homePositionCenter) { // output X in standard block by itself if centering
        writeBlock(gMotionModal.format(0), xOutput.format(_xHome));
        _xyzMoved[0] = false;
        break;
      }
      words.push("X" + xyzFormat.format(_xHome));
      break;
    case Y:
      words.push("Y" + xyzFormat.format(_yHome));
      break;
    case Z:
```

```
      words.push("Z" + xyzFormat.format(_zHome));
      retracted = true;
      break;
    }
  }

  // output move to home
  if (words.length > 0) {
    if (_useG28) {  // use G28 to move to home position
      gAbsIncModal.reset();
      writeBlock(gFormat.format(28), gAbsIncModal.format(91), words);
      writeBlock(gAbsIncModal.format(90));
    } else { // use G53 to move to home position
      gMotionModal.reset();
      writeBlock(gAbsIncModal.format(90), gFormat.format(53), gMotionModal.format(0), words);
    }

    // force any axes that move to home on next block
    if (_xyzMoved[0]) {
      xOutput.reset();
    }
    if (_xyzMoved[1]) {
      yOutput.reset();
    }
    if (_xyzMoved[2]) {
      zOutput.reset();
    }
  }
}
```

Sample writeRetract Function

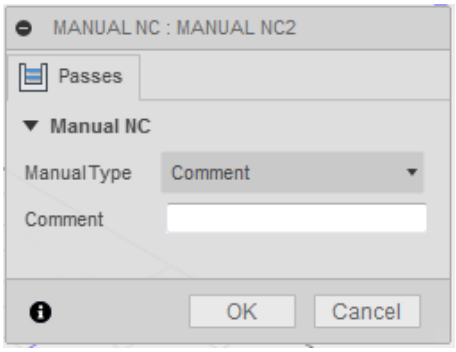
5 Manual NC Commands

Manual NC commands are used to control the behavior of individual operations when there is not a setting in the operation form for controlling a specific feature of a control. You can use Manual NC commands to display a message, insert codes into the output NC file, perform an optional stop, define a setting, etc. The Manual NC menu is accessed from different areas of the ribbon menu depending on the product you are running.



Selecting a Manual NC Command in the HSM Products (Fusion 360, Inventor, HSMWorks)

Once you select the Manual NC menu you will see a form displayed that is used to select the type of Manual NC command that you want to pass to the post processor and optionally the parameter that will be passed with the command.



Defining a Manual NC Command

If you use a Manual NC command in your part, then it is necessary that the post processor is equipped to handle this command. Some of the commands are supported by the stock post processors, such as *Stop*, *Optional stop*, and *Dwell*, while support would have to be added to the post processor to support other Manual NC commands. If you use a Manual NC command that is not supported by the post, then it will either generate an error or be ignored. The general rule is it will generate an error if the *onCommand* function is called and will be ignored when another function is called.

5.1 onManualNC and expandManualNC

```
function onManualNC(command, value) {
expandManualNC(command, value)
```

Arguments	Description
command	The Manual NC command that invoked the function.
value	The value entered with the command.

The *onManualNC* function is defined in the post processor and is used to process Manual NC commands. It accepts the command and the value that is assigned to the command. If the *onManualNC* function is not defined in the post processor, then a separate function will be called as listed in the table below.

The *expandManualNC* command can also be used to process the Manual NC command using the separate functions listed in the table. It is typically used as the default condition in the *onManualNC* function to process commands where you do not care if they are entered as a Manual NC command or from an internal call in the post processor.

The following table describes the Manual NC commands along with the function that will be called when the command is processed when the *onManualNC* function does not exist or *expandManualNC* is called.

Manual NC Command	Description	Command	Value	Function Called
Comment	Operator message	COMMAND_COMMENT	message	onComment
Stop	Machine stop	COMMAND_STOP		onCommand
Optional Stop	Optional stop	COMMAND_OPTIONAL_STOP		onCommand
Dwell	Dwell	COMMAND_DWELL	Dwell time in seconds	onDwell
Tool break control	Check for tool breakage	COMMAND_BREAK_CONTROL		onCommand
Measure tool	Automatically measure tool length	COMMAND_TOOL_MEASURE		onCommand
Start chip transport	Start chip conveyor	COMMAND_START_CHIP_TRANSPORT		onCommand
Stop chip transport	Stop chip conveyor	COMMAND_STOP_CHIP_TRANSPORT		onCommand
Open door	Open main door	COMMAND_OPEN_DOOR		onCommand
Close door	Close main door	COMMAND_CLOSE_DOOR		onCommand
Calibrate	Calibration of machine	COMMAND_CALIBRATE		onCommand
Verify	Verify integrity of machine	COMMAND_VERIFY		onCommand
Clean	Request a cleaning cycle	COMMAND_CLEAN		onCommand
Action	User defined action	COMMAND_ACTION	text	onParameter
Print message	Print a message from the machine	COMMAND_PRINT_MESSAGE	message	onParameter
Display message	Display operator message	COMMAND_DISPLAY_MESSAGE	message	onParameter

Manual NC Command	Description	Command	Value	Function Called
Alarm	Create an alarm on the machine	COMMAND_ALARM		onCommand
Alert	Request an alert event on the machine	COMMAND_ALERT		onCommand
Pass through	Output literal text to NC file	COMMAND_PASS_THROUGH	text	onPassThrough
Force tool change	Force a tool change	section.getForceToolChange()		(none)
Call program	Call a subprogram	COMMAND_CALL_PROGRAM	text	onParameter

Manual NC Commands

5.1.1 Sample onManualNC Function

The *onManualNC* function is a recent addition to the post processor and will not be found in most generic post processors. You do not have to define it to process Manual NC commands, and if it is defined, do not need to process all Manual NC commands in this function. It could be used to process only the commands where you need to know if they were generated from a CAM Manual NC command instead of a direct call from within the post processor.

For example, the following *onManualNC* function definition could be used to process comments entered using the CAM Manual NC command differently than comments generated from the post processor. It simply appends the text 'MSG,' prior to the comment for a Manual NC *Display comment* command. All other Manual NC commands are processed normally.

```
function onManualNC(command, value) {
  switch (command) {
  case COMMAND_DISPLAY_MESSAGE:
    writeComment("MSG, " + value);
    break;
  default:
    expandManualNC(command, value);  // normal processing of Manual NC command
  }
}
```

Handling of Display Message Command in onManualNC

5.1.2 Delay Processing of Manual NC Commands

Manual NC commands are processed at the placement in the operation tree where they are entered, which means that they will be processed prior to the call to *onSection*. Since *onSection* typically terminates the previous operation prior to starting the new operation, this might not be the desirable location to process the Manual NC command.

The following code examples show how Manual NC commands can be buffered and output at any point during the operation. You can simply copy the *onManualNC* and *executeManualNC* functions into your post processor and add the appropriate call(s) to *executeManualNC* where you want to process the Manual NC commands.

```
/**
  Buffer Manual NC commands for processing later
*/
var manualNC = [];
function onManualNC(command, value) {
  manualNC.push({command:command, value:value});
}

/**
  Processes the Manual NC commands
  Pass the desired command to process or leave argument list blank to process all buffered commands
*/
function executeManualNC(command) {
  for (var i = 0; i < manualNC.length; ++i) {
    if (!command || (command == manualNC[i].command)) {
      switch(manualNC[i].command) {
      case COMMAND_DISPLAY_MESSAGE:
        writeComment("MSG, " + manualNC[i].value);
        break;
      default:
        expandManualNC(manualNC[i].command, manualNC[i].value);
      }
    }
  }
  for (var i = 0; i < manualNC.length; ++i) {
    if (!command || (command == manualNC[i].command)) {
      manualNC.splice(i, 1);
    }
  }
}
```

Manual NC Commands Support Functions

The calls to process the Manual NC commands can be placed anywhere in the post processor. In the following code example, the COMMAND_DISPLAY_MESSAGE command is processed just before the tool change block is output and the rest of the Manual NC commands after the tool change block.

```
  executeManualNC(COMMAND_DISPLAY_MESSAGE);  // display Manual NC message
  writeBlock("T" + toolFormat.format(tool.number), mFormat.format(6));
  if (tool.comment) {
    writeComment(tool.comment);
```

```
  }
  executeManualNC(); // process remaining Manual NC commands
```

Processing of Manual NC Commands in the Desired Location

The following sections give a description of the functions that are called by the Manual NC commands outside of the onManualNC function and samples on how they are handled in the functions. The *onComment* and *onDwell* functions are described in the *Entry Functions* chapter, since they are simple functions and behave in the same manner no matter how they are called.

5.2 onCommand

```
function onCommand(command) {
```

Arguments	Description
command	Command to process.

All Manual NC commands that do not require an associated parameter are passed to the *onCommand* function and as you see from the *Manual NC Commands* table, this entails the majority of the commands. The onCommand function also handles other commands that are not generated by a Manual NC command and these are described in the *onCommand* section in the *Entry Functions* chapter.

```
// define commands that output a single M-code
var mapCommand = {
  COMMAND_STOP:0,
  COMMAND_OPTIONAL_STOP:1,
  COMMAND_START_CHIP_TRANSPORT:31,
  COMMAND_STOP_CHIP_TRANSPORT:33
  …
};

function onCommand(command) {
  switch (command) {
  …
  case COMMAND_BREAK_CONTROL: // handle the 'Tool break' command
    if (!toolChecked) { // avoid duplicate COMMAND_BREAK_CONTROL
      onCommand(COMMAND_STOP_SPINDLE);
      onCommand(COMMAND_COOLANT_OFF);
      writeBlock(
        gFormat.format(65),
        "P" + 9853,
        "T" + toolFormat.format(tool.number),
        "B" + xyzFormat.format(0),
        "H" + xyzFormat.format(properties.toolBreakageTolerance)
      );
      toolChecked = true;
    }
```

```
  return;
case COMMAND_TOOL_MEASURE: // ignore tool measurements
  return;
}

// handle commands that output a single M-code
var stringId = getCommandStringId(command);
var mcode = mapCommand[stringId];
if (mcode != undefined) {
  writeBlock(mFormat.format(mcode));
} else {
  onUnsupportedCommand(command);
}
                                                    }
```

Handling Manual NC Commands in the onCommand Function

5.3 onParameter

```
function onParameter(name, value) {
```

Arguments	Description
name	Parameter name.
value	Value stored in the parameter.

The *onParameter* function handles the *Action*, *Call program*, *Display message*, and *Print message* Manual NC commands. It is passed both the name of the parameter being defined and the text string associated with that parameter. This section will describe how the *Action* command can be used, since this is the most commonly used of these commands.

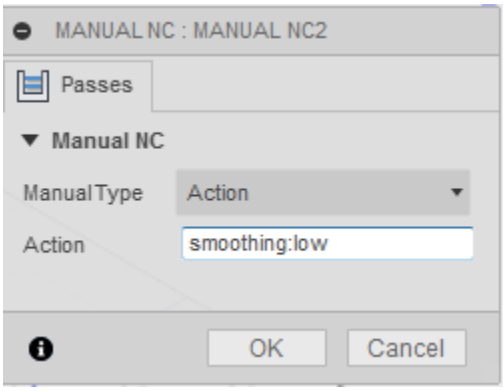
The *Action* command is typically used to define post processor settings, similar to the post properties defined at the top of the post processor, except that the settings defined using this command typically only apply to a single operation. Since the HSM operations are executed in the order that they are defined in the CAM tree, the Manual NC command will always be processed prior to the operation that they precede. You can also use the *Action* command to define a setting so that the command can be executed within another section of the post, by referencing this setting. You can even define settings that are typically set in the post properties into your program, so you are not reliant on making sure that the property is set for a specific program. In this case the *Action* command would set the value of the post property based on the input value associated with the command.

It is the *onParameter* function's responsibility to parse the text string passed as part of the *Action* command. The text string could be a value, list of values, command and value, etc. The following table lists the *Action* commands that are supported by the sample post processor code used in this section. These *Action* commands set variables that will be used elsewhere in the program.

Action Command	Values	Description
Smoothing	Off, Low, Medium, High	Sets the smoothing type
Tolerance	.001-.999	Smoothing tolerance
fastToolChange	Yes, No	Overrides the *fastToolChange* post property

Sample Action Type Manual NC Commands

In this example, the format for entering the *Action* Manual NC command is to specify the command followed by the ':' separator which in turn is followed by the value, in the *Action* text field.



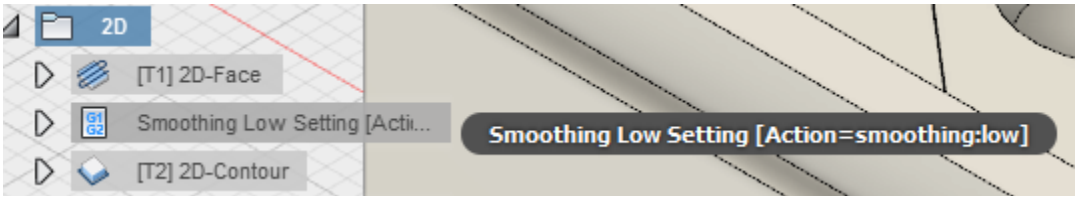
***Action* Command Format**

```
var smoothingType = 0;
var smoothingTolerance = .001;
function onParameter(name, value) {
  var invalid = false;
  switch (name) {
  case "action":
    var sText1 = String(value).toUpperCase();
    var sText2 = new Array();
    sText2 = sText1.split(":");
    if (sText2.length != 2) {
      error(localize("Invalid action command: ") + value);
      return;
    }
    switch (sText2[0]) {
    case "SMOOTHING":
      smoothingType = parseChoice(sText2[1], "OFF", "LOW", "MEDIUM", "HIGH");
      if (smoothingType == undefined) {
        error(localize("Smoothing type must be Off, Low, Medium, or High"));
        return;
      }
      break;
    case "TOLERANCE":
      smoothingTolerance = parseFloat(sText2[1]);
      if (isNaN(smoothingTolerance) || ((smoothingTolerance < .001) || (smoothingTolerance > .999))) {
```

```
        error(localize("Smoothing tolerance must be a value between .001 and .999"));
        return;
      }
      break;
    case "FASTTOOLCHANGE":
      var fast = parseChoice(sText2[1], "YES", "NO");
      if (fast == undefined) {
        error(localize("fastToolChange must be Yes or No"));
        return;
      }
      properties.fastToolChange = fast;
      break;
    default:
      error(localize("Invalid action parameter: ") + sText2[0] + ":" + sText2[1]);
      return;
    }
  }
}

/* returns the choice specified in a text string compared to a list of choices */
function parseChoice() {
  var stat = undefined;
  for (i = 1; i < arguments.length; i++) {
    if (String(arguments[0]).toUpperCase() == String(arguments[i]).toUpperCase()) {
      if (String(arguments[i]).toUpperCase() == "YES") {
        stat = true;
      } else if (String(arguments[i]).toUpperCase() == "NO") {
        stat = false;
      } else {
        stat = i - 1;
        break;
      }
    }
  }
  return stat;
}
```

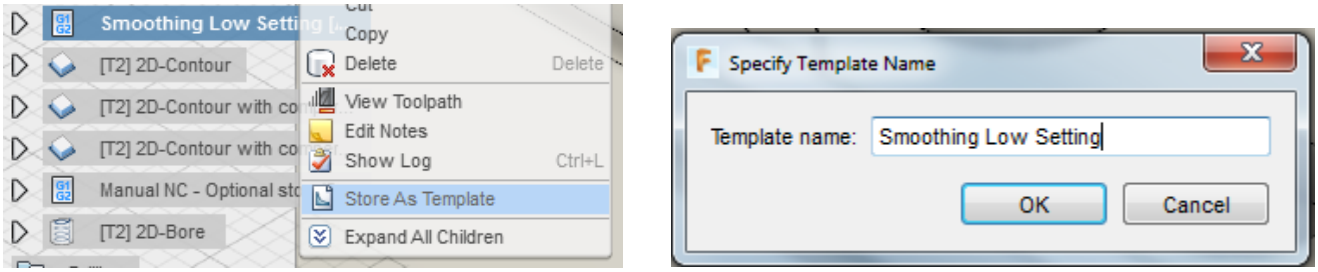
Handling the *Action* Manual NC Command

To make it easier to use custom *Action* Manual NC commands you can use the Template capabilities of HSM. First you will create the Manual NC command that you will turn into a template using the example in the *Action Command Format* picture shown above. Once the Manual NC command is created you will want to give it a meaningful name by renaming it in the Operation Tree.



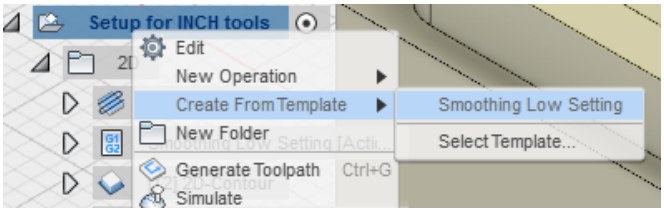
Rename the *Action* Manual NC Command Before Creating Template

Now you will create a template from this Manual NC command by right clicking on the Manual NC command and selecting *Store As Template*. You will want to give the template the same name as you did in the rename operation.



Creating the Manual NC Command Template

The template is now ready to be used in other operations and parts. You do this by right clicking a Setup or a Folder in the Operations Tree, position the mouse over the *Create From Template* menu and select the template you created.



Using the Manual NC Command Template You Created

5.4 onPassThrough

Function onPassThrough (value)

Arguments	Description
value	Text to be output to the NC file.

The *Pass through* Manual NC command is used to pass a text string directly to the NC file without any processing by the post processor. It is similar to editing the NC file and adding a line of text by hand. The text string could be standard codes (G, M, etc.) or a simple message. Since the post has no control or knowledge of the codes being output, it is recommended that you use the *Pass through* command sparingly and only with codes that cannot be output using another method.

The *onPassThrough* function handles the *Pass through* Manual NC command and is passed the text entered with the command. The following sample code will accept a text string with comma delimiters that will separate the text into individual lines.

```
function onPassThrough(text) {
  var commands = String(text).split(",");
  for (text in commands) {
    writeBlock(commands[text]);
  }
}
```

Output Lines of Codes/Text Separated by Commands Using the *Pass through* Manual NC Command

Like the *Action* Manual NC command, you can setup a Template to use with the *Pass through* command if you find yourself needing to output the same codes in multiple instances.

6 Debugging

6.1 Overview

The first thing to note when debugging is that there is not an interactive debugger associated with the Autodesk CAM post processors. This means that all debugging information must be output using settings within the post and with explicit writes. This section describes different methods you can use when debugging your post.

You can also use the HSM Post Processor Editor to aid in debugging your program as described in the *Running/Debugging the Post* section of this manual

6.2 The dump.cps Post Processor

The dump.cps post processor will process an intermediate CNC file and output a file that contains all of the information passed from HSM to the post processor. The output file has a file extension of *.dmp*. The contents of the dump file will show the settings of all parameter values and will list the entry functions called along the arguments passed to the function and any settings that apply to that function. The dump.cps output can be of tremendous value when developing and debugging a post processor.

```
342: onParameter('dwell', 0)
344: onParameter('incrementalDepth', 0.03937007874015748)
346: onParameter('incrementalDepthReduction', 0.003937007932681737)
348: onParameter('minimumIncrementalDepth', 0.01968503937007874)
350: onParameter('accumulatedDepth', 5)
352: onParameter('chipBreakDistance', 0.004023600105694899)
354: onMovement(MOVEMENT_CUTTING /*cutting*/)
354: onCycle()
  cycleType='chip-breaking'
  cycle.clearance=123456
  cycle.retract=0.19685039370078738
```

```
  cycle.stock=0
  cycle.depth=0.810440544068344
  cycle.feedrate=15.748000257597194
  cycle.retractFeedrate=39.370100366787646
  cycle.plungeFeedrate=15.748000257597194
  cycle.dwell=0
  cycle.incrementalDepth=0.03937007874015748
  cycle.incrementalDepthReduction=0.003937007932681737
  cycle.minimumIncrementalDepth=0.01968503937007874
  cycle.accumulatedDepth=5
  cycle.chipBreakDistance=0.004023600105694899
354: onCyclePoint(-1.25, 0.4999999924907534, -0.810440544068344)
355: onCyclePoint(1.25, 0.4999999924907534, -0.810440544068344)
356: onCycleEnd()
```

Sample dump.cps Output

6.3 Debugging using Post Processor Settings

There are variables available to the developer that control the output of debugging information. This section contains a description of these variables.

6.3.1 debugMode

```
debugMode = true;
```

Setting the *debugMode* variable to true enables the output of debug information from the *debug* command and is typically defined at the start of the post processor.

6.3.2 setWriteInvocations

```
setWriteInvocations (value);
```

Arguments	Description
value	*true* outputs debug information for the entry functions.

Enabling the *setWriteInvocations* setting will create debug output in the NC file similar to what is output using the *dump* post processor. The debug information contains the entry functions (*onParameter*, *onSection*, etc.) called during post processing and the parameters that they are called with. This information will be output prior to actually calling the entry function and is labeled using the *!DEBUG:* text.

```
!DEBUG: onRapid(-0.433735, 1.44892, 0.23622)
N190 Z0.2362
!DEBUG: onLinear(-0.433735, 1.44892, 0.0787402, 39.3701)
N195 G1 Z0.0787 F39.37
```

```
!DEBUG: onLinear(-0.433735, 1.44892, -0.5, 19.685)
N200 Z-0.5 F19.68
```

setWriteInvocations Output

6.3.3 setWriteStack

```
setWriteStack (value);
```

Arguments	Description
value	*true* outputs the call stack that outputs the line to the NC file.

Enabling the *setWriteStack* setting displays the call stack whenever text is output to the NC file. The call stack will consist of the *!DEBUG:* label, the call level, the name of the post processor, and the line number of the function call (the function name is not included in the output).

```
!DEBUG: 1 rs274.cps:108
!DEBUG: 2 rs274.cps:919
!DEBUG: 3 rs274.cps:357
N125 M5
```

setWriteStack Output

```
…
108: writeWords2("N" + sequenceNumber, arguments);
…
357: onCommand(COMMAND_STOP_SPINDLE);
…
919: writeBlock(mFormat.format(mcode));
```

Post Processor Contents

6.4 Functions used with Debugging

Functions that can be used to output debug information to the log and NC files include *debug*, *writeln*, and *log*. Additionally, the *writeComment* function present in almost all post processors can be used.

The text provided to the debug functions can contain operations and follow the same rules as defining a string variable in JavaScript. You can also specify vectors or matrixes and these will be properly formatted for output. For example,

```
var x = 3;
debug("The value of x is " + x);
```

For floating point values you may want to create a format that limits the number of digits to right of the decimal point, as some numbers can be quite long when output.

```
var numberFormat = createFormat({decimals:4});
var x = 3;
```

```
debug("The value of x is " + numberFormat.format(x));
```

When writing output debug information to the log and/or NC files it is recommended that you precede the debug text with a fixed string, such as "DEBUG – ", so that it is easily discernable from other output.

6.4.1 debug

```
debug (text);
```

Arguments	Description
text	Outputs *text* to the log file when *debugMode* is set to *true*.

The *debug* function outputs the provided text message to the log file only when the *debugMode* variable is set to true. The text is output exactly as provided, without any designation that the output was generated by the *debug* function.

6.4.2 log

```
log(text);
```

Arguments	Description
text	Outputs *text* to the log file.

The *log* function outputs the text to the log file. It is similar to the *debug* function, but does not rely on the *debugMode* setting.

6.4.3 writeln

```
writeln(text);
```

Arguments	Description
text	Outputs *text* to the NC file.

The *writeln* function outputs the text to the NC file. It is used extensively in post processors to output valid data to the NC file and not just debug text.

6.4.4 writeComment

```
writeComment(text);
```

Arguments	Description
text	Outputs *text* to the NC file as a comment.

The *writeComment* function is defined in the post processor and is used to output comments to the output NC file. It is described in the onComment section of this manual.

6.4.5 writeDebug

Function writeDebug(text);

Arguments	Description
text	Outputs *text* to the NC and log files.

The *writeDebug* function is not typically present in the generic post processors. You can create one to handle the output of debug information to both the log file and NC file so that if the post processor either fails or runs successfully you would still see the debug output.

```
function writeDebug(text) {
  if (true) { // can use the global setting 'debugMode' instead
    writeln("DEBUG - " + text); // can use 'writeComment' instead
    log("DEBUG - " + text); // can use 'debug' instead
  }
}
```

Sample writeDebug Function

7 Multi-Axis Post Processors

7.1 Adding Basic Multi-Axis Capabilities

Adding multi-axis capabilities to a post processor can be rather straight forward or difficult depending on the situation. This chapter will cover the basics and the more complex aspects of multi-axis support, such as adjusting points for a head, inverse time feedrates, etc.

Please note that support for 3+2 operations is not handled here, except for the setup of the machine. Refer to the Work Plane section in the *onSection* chapter for a description on how to handle 3+2 operations.

7.1.1 Create the Rotary Axes Formats

The output formats for the rotary axes must first be defined. In existing multi-axis posts and posts that contain the skeleton structure of multi-axis support these codes should already be defined. You should add (or verify that they already exist) the following definitions at the top of the post processor in the same area that all other formats are defined.

```
var abcFormat = createFormat({decimals:3, forceDecimal:true, scale:DEG});
…
var aOutput = createVariable({prefix:"A"}, abcFormat);
var bOutput = createVariable({prefix:"B"}, abcFormat);
var cOutput = createVariable({prefix:"C"}, abcFormat);
```

Define the Rotary Axes Formats

The *scale:DEG* parameter specifies that the rotary axes angles will be output in degrees. If you require the output to be in radians, then omit the *scale* setting.

7.1.2 Create a Multi-Axis Machine Configuration

The *onOpen* function should contain the logic to create the machine configuration. The code should be present in most generic post processors, but it can be easily added to posts that are only setup for 3-axis capabilities.

```
if (true) { // note: setup your machine here
  var aAxis = createAxis({coordinate:X, table:false, axis:[1, 0, 0], range:[-360, 360], preference:1});
  var cAxis = createAxis({coordinate:Z, table:false, axis:[0, 0, 1], range:[-360, 360], preference:1});
  machineConfiguration = new MachineConfiguration(aAxis, cAxis);

  setMachineConfiguration(machineConfiguration);
  optimizeMachineAngles2(0); // TCP mode
 }

 if (!machineConfiguration.isMachineCoordinate(0)) {
  aOutput.disable();
 }
 if (!machineConfiguration.isMachineCoordinate(1)) {
  bOutput.disable();
 }
 if (!machineConfiguration.isMachineCoordinate(2)) {
  cOutput.disable();
 }
```

Define the Machine Configuration

The conditional at the start of the logic must be set to *true* for the multi-axis configuration to be defined. You can now customize the rotary axes to match the machine configuration as described below.

First, you will need to use the *createAxis* function to define all available axes. The *createAxis* function accepts the following parameters.

Parameter	Description
table	Set to *true* when the rotary axis is a table, or *false* if it is a head. The default if not specified is *true*.
axis	Specifies the rotational axis of the rotary axis in the format of a vector, i.e. [0, 0, 1]. This vector does not have to be orthogonal to a major plane, for example it could be [0, .7017, .7017]. The direction of the rotary axes are based on the righthand rule for tables and the lefthand rule for heads. You can change direction of the axis by supplying a vector pointing in the opposite direction, i.e. [0, 0, -1]. This parameter is required.
offset	Defines the rotational position of the axis in the format of a coordinate, i.e. [0, 0, 0], but it is not currently supported by the post processor. Adding support for this offset position when the rotary configurataion consists of at least one head is discussed in the *Adjusting the Points for Rotary Heads* section of this chapter. The default is [0, 0, 0].

Parameter	Description
coordinate	Defines the coordinate of the axis, either X, Y, or Z. You will notice a number used in most of the generic posts, in this case 0=X, 1=Y, and 2=Z. Either specification is acceptable input. This parameter is required.
cyclic	Defines whether the axis is cyclic (continuous) in nature, in that the output will always be within the range specified by the *range* parameter. Cyclic axes will never cause the *onRewindFunction* to be called, since they are continuous in nature and do not have limits. The range applies specifically to output values for this axis. The default is *false*.
range	Defines the upper and lower limits of the rotary axis using the format [lower, upper]. If the rotary axis is cyclic, then the range sets the limits of the output values for this axis, if it is not cyclic the range is the actual physical limits of the machine.
preference	Specifies the preferred angle direction at the beginning of an operation. -1 = choose the negative angle, 0 = no preference, and 1 = choose the positive angle. The default is 0.
reset	Defines the starting position of the axis for a new operation and when the rotary axes need to be rewound and reconfigured due to exceeding the limits. 0 = remember the position from previous section, 1 = reset to 0 at start of operation, 2 = reset to 0 at automatic rewind, 3 = reset to 0 at start of operation and at automatic rewind. This parameter is implemented since R42225 of the post engine.
resolution	Specifies the resolution in degrees of the rotational actuator. Typically, this will be set to the number of digits to the right of the decimal as specified in the *createFormat* call for the rotary axes. The default is 0.

createAxis Parameters

```
// 4 axis setup, A rotates around X, direction is positive:
var aAxis = createAxis({coordinate:X, table:true, axis:[1, 0, 0], range:[-360,360], preference:1});
machineConfiguration = new MachineConfiguration(aAxis);

// 4 axis setup, A rotates around X, direction is negative:
var aAxis = createAxis({coordinate:X, table:true, axis:[-1, 0, 0], range:[-360,360], preference:1});
machineConfiguration = new MachineConfiguration(aAxis);


// 5 axis setup, B rotates around Y, C rotates around Z, directions both positive:
var bAxis = createAxis({coordinate:Y, table:true, axis:[0, 1, 0], range:[-360,360], preference:1});
var cAxis = createAxis({coordinate:Z, table:true, axis:[0, 0, 1], range:[-360,360], preference:1});
machineConfiguration = new MachineConfiguration(bAxis, cAxis);
```

Sample Rotary Configurations

Now you will have to define the machine configuration using the following command.

```
machineConfiguration = new MachineConfiguration(aAxis, cAxis);
```

Define Machine Configuration

The order in which the axes are defined in the *MachineConfiguration* definition is important and must use the following order.

Order	Rotary Axis
1	Rotary head slave/rider
2	Rotary head master/carrier
3	Rotary table master/carrier
4	Rotary table slave/rider

machineConfiguration Rotary Axis Order

The *setMachineConfiguration(machineConfiguration);* statement is required to activate the machine configuration with the defined rotary axes.

The *optimizeMachineAngles2* function determines if the tool endpoint coordinates should be adjusted for the rotary axes and can have the following values.

Value	Description
0	Don't adjust the coordinates for the rotary axes. Used for TCP mode.
1	Adjust the coordinates for the rotary axes. If either of the rotary axes is a head, then this setting should not be used as there is no internal support for adjusting the points for a head. Adding this capability to the post is discussed further in the *Adjusting the Points for Rotary Heads* section of this chapter.
2	Adjust the coordinates for rotary tables. No adjustment will be made for heads.

optimizeMachineAngles2 Settings

The code to disable the output variables for each axis is generic in nature and should be exactly as shown.

```
if (!machineConfiguration.isMachineCoordinate(0)) {
  aOutput.disable();
}
if (!machineConfiguration.isMachineCoordinate(1)) {
  bOutput.disable();
}
if (!machineConfiguration.isMachineCoordinate(2)) {
  cOutput.disable();
}
```

Disable the Output of Unused Rotary Axes

7.1.3 Output Initial Rotary Position

The initial rotary axes positions are usually output in the *onSection* function in the same section of code that handles the Work Plane. The function *getInitialToolAxisABC*() is used to obtain the initial rotary axes positions.

```
  // set working plane after datum shift

  if (currentSection.isMultiAxis()) {
    forceWorkPlane();
    cancelTransformation();
    onCommand(COMMAND_UNLOCK_MULTI_AXIS);
    var abc = currentSection.getInitialToolAxisABC();
    gMotionModal.reset();
    writeBlock(
      gMotionModal.format(0),
      conditional(machineConfiguration.isMachineCoordinate(0), aOutput.format(abc.x)),
      conditional(machineConfiguration.isMachineCoordinate(1), bOutput.format(abc.y)),
      conditional(machineConfiguration.isMachineCoordinate(2), cOutput.format(abc.z))
    );
  } else {
```

Output Initial Rotary Axes Positions

7.1.4 Create the onRapid5D and onLinear5D Functions

Now that you have the machine defined you will need to verify that the onRapid5D and onLinear5D functions are present. These are the functions that will process the tool path generated by multi-axis operations. If your post already has these functions defined, then great you should be (almost) ready to go, if not then add the following functions to your post.

```
function onRapid5D(_x, _y, _z, _a, _b, _c) {
  if (!currentSection.isOptimizedForMachine()) {
    error(localize("This post configuration has not been customized for 5-axis simultaneous
toolpath."));
    return;
  }
  if (pendingRadiusCompensation >= 0) {
    error(localize("Radius compensation mode cannot be changed at rapid traversal."));
    return;
  }
  var x = xOutput.format(_x);
  var y = yOutput.format(_y);
  var z = zOutput.format(_z);
  var a = aOutput.format(_a);
  var b = bOutput.format(_b);
  var c = cOutput.format(_c);
  if (x || y || z || a || b || c) {
    writeBlock(gMotionModal.format(0), x, y, z, a, b, c);
    feedOutput.reset();
  }
}
```

onRapid Function

```
function onLinear5D(_x, _y, _z, _a, _b, _c, feed) {
  if (!currentSection.isOptimizedForMachine()) {
    error(localize("This post configuration has not been customized for 5-axis simultaneous
toolpath."));
    return;
  }
  if (pendingRadiusCompensation >= 0) {
    error(localize("Radius compensation cannot be activated/deactivated for 5-axis move."));
    return;
  }
  var x = xOutput.format(_x);
  var y = yOutput.format(_y);
  var z = zOutput.format(_z);
  var a = aOutput.format(_a);
  var b = bOutput.format(_b);
  var c = cOutput.format(_c);
  var f = feedOutput.format(_feed);

  if (x || y || z || a || b || c) {
    writeBlock(gMotionModal.format(1), x, y, z, a, b, c, f);
  } else if (f) {
    if (getNextRecord().isMotion()) { // try not to output feed without motion
      feedOutput.reset(); // force feed on next line
    } else {
      writeBlock(gMotionModal.format(1), f);
    }
  }
}
```

onLinear5D Function

Both of these functions as presented are basic in nature and the requirements for your machine may require some modification. For example, the tool endpoint may have to be adjusted for rotary heads or inverse time feedrates may need to be supported.

7.1.5 Multi-Axis Common Functions

There are functions that are useful when developing a post processor for a multi-axis machine. These functions are used to determine if the rotary axes are configured, the beginning and ending tool axis or rotary axes positions for an operation, and control the flow of the multi-axis logic.

Function	Description
machineConfiguration.isMultiAxisConfiguration()	Returns *true* if a machine configuration containing rotary axes has been defined. It is still possible to create output

Function	Description
	for some multi-axis machines if the rotary axes have not been defined, by outputting the tool axis vector instead of the rotary axes positions or by using Euler angles for 3+2 operations.
machineConfiguration.getABC(matrix)	Returns the rotary axes angles for the provided matrix. This matrix is usually the Work Plane matrix (*currentSection.workPlane*).
machineConfiguration.remapToABC(abc, current)	Returns the closest rotary axes angles to the *current* axes positions as a Vector. *abc* is the rotary angles to be remapped.
machineConfiguration.remapABC(abc)	Returns the rotary axes angles within the valid range for each angle as a Vector..
machineConfiguration.getPreferred(abc)	Returns the preferred rotary axes angles given the input *abc* angles as a Vector. The preferred angles will be in the valid range for each angle.
machineConfiguration.isABCSupported(abc)	Returns *true* if the abc angles are within the valid ranges for the defined rotary axes. Returns *false* if any of the angles are outside of their defined range.
section.isOptimizedForMachine()	Returns *true* if an active machine configuration containing rotary axes is defined for the provided section..
section.isMultiAxis()	Returns *true* if the operation specified by *section* is a multi-axis operation.
section.getGlobalInitialToolAxis()	Returns the initial tool axis for the provided section as a Vector. Usually used at the start of an operation using the *currentSection* variable.
section.getInitialToolAxisABC()	Returns the initial rotary axes angles for the provided section as a Vector. Usually used at the start of an operation using the *currentSection* variable. An error will be generated if a machine configuration containing rotary axes has not been defined.
section.getGlobalFinalToolAxis()	Returns the final tool axis for the provided section as a Vector. Usually used at the start of an operation using *getPreviousSection()*.
section.getFinalToolAxisABC()	Returns the final rotary axes angles for the provided section as a Vector. Usually used at the start of an operation using *getPreviousSection()*. An error will be generated if a machine configuration containing rotary axes has not been defined.
getCurrentDirection()	Returns the current rotary axes angles as a Vector in a multi-axis operation. It will return the Work Plane forward vector when in a 3-axis or 3+2 operation.
is3D()	Returns *true* if the entire program is a 3-axis operation with no multi-axis operations. Returns *false* if even one operation is a 3+2 or multi-axis operation.

Multi-Axis Common Functions

7.2 Output of Continuous Rotary Axis on a Rotary Scale

There are two different styles that are commonly used for rotary axes output, using a linear scale or a rotary scale. A linear scale is the more standard case in today's machines and will move on a progressive scale similar to a linear axis output. For example, a value of 720 degrees will move the axis two revolutions from 0 degrees. A linear scale is almost always used with a non-continuous axis and can be used with a continuous rotary axis.

A rotary scale on the other had typically outputs the rotary angle positions between 0 and 360 degrees, usually with the sign ± specifying the direction. If a sign is not required and the control will always take the shortest route, then it is pretty straight forward to output the rotary axis on a rotary scale, simply define it as a cyclic axis with a range of 0 to 360 degrees.

```
var aAxis = createAxis({coordinate:0, table:true, axis:[1, 0, 0], cyclic:true, range:[0, 360]});
```

Create Rotary Axis on a Rotary Scale

You may also have to create a blank *onRewindMachine* function so that the post processor does not produce an error when the specified range of the axis is exceeded. The new version of the post kernel will treat a cyclic axis as a continuous axis and will not require rewinds when the specified range is exceeded. If you are creating a 5-axis post-processor and the range of the non-continuous rotary axis can be exceeded, then refer to the *onRewindMachine* section in this chapter on how to handle this situation.

```
function onRewindMachine(_a, _b, _c) {
}
```

Add a Blank onRewindMachine Function

For controls that require a sign to designate the direction the rotary axis will move, you will need to keep track of the current axis position and include a function to calculate the rotary axis output with the sign.

```
// collected state
…
var previousABC = new Vector(0, 0, 0)
```

Define the previousABC Variable

```
function setWorkPlane (abc) {
…
  previousABC = abc;
}
…
function onSection() {
…
// set working plane after datum shift
if (currentSection.isMultiAxis()) {
…
  previousABC = abc;
```

```
}
…
function onRapid5D (_x, _y, _z, _a, _b, _c) {
…
  previousABC = new Vector(_a, _b, _c);
}
…
function onLinear5D (_x, _y, _z, _a, _b, _c, feed) {
…
  previousABC = new Vector(_a, _b, _c);
}
```

Save the Current Rotary Axes Positions

The *previousABC* variable should only be set in the *setWorkPlane* function when 3+2 operations are output as the actual rotary axes positions. If the 3+2 orientation is defined using Euler angles, then the rotary axis positions need to be calculated and stored in the *previousABC* variable separately. You can typically use the *machineConfiguration.getABC* function to calculate the rotary axes positions from the work plane.

```
previousABC = machineConfiguration.getABC(currentSection.workPlane);
```

Calculate the Rotary Axes from the 3+2 Work Plane

You will now need to add the function that calculates the rotary axes using a directional (signed) value.

```
/** Calculate angles on rotary scale with signed direction */
function getDirectionalABC(_startAngle, _endAngle, _output) {
  var signedAngle = _endAngle;
  // angles are the same, set the previous output angle to the current angle so it is not output
  if (!abcFormat.areDifferent(_startAngle, _endAngle)) {
    _output.format(_startAngle);
  }
  // calculate the correct direction (sign) based on CLW/CCW direction
  var delta = abcFormat.getResultingValue(_endAngle - _startAngle);
  if (((delta < 0) && (delta > -180.0)) || (delta > 180.0)) {
    if (_endAngle == 0) {
      signedAngle = -Math.PI*2;
    } else {
      signedAngle = -_endAngle;
    }
  }
  return signedAngle;
}
```

getDirectionalABC Function Calculates Rotary Axis on a Rotary Scale

The final required step is to call the *getDirectionalABC* function wherever the rotary axis is output. In the following code, the C-axis is output on a rotary scale.

```
function setWorkPlane(abc) {
…
  writeBlock(
    gMotionModal.format(0),
    conditional(machineConfiguration.isMachineCoordinate(0), aOutput.format(abc.x)),
    conditional(machineConfiguration.isMachineCoordinate(1), bOutput.format(abc.y)),
    conditional(machineConfiguration.isMachineCoordinate(2),
      cOutput.format(getDirectionalABC(previousABC.z, abc.z, cOutput)))
  );
…
function onSection() {
….
// set working plane after datum shift

    if (currentSection.isMultiAxis()) {
      forceWorkPlane();
      cancelTransformation();
      onCommand(COMMAND_UNLOCK_MULTI_AXIS);
      var abc = currentSection.getInitialToolAxisABC();
      gMotionModal.reset();
      writeBlock(
        gMotionModal.format(0),
        conditional(machineConfiguration.isMachineCoordinate(0), aOutput.format(abc.x)),
        conditional(machineConfiguration.isMachineCoordinate(1), bOutput.format(abc.y)),
        conditional(machineConfiguration.isMachineCoordinate(2),
          cOutput.format(getDirectionalABC(previousABC.z, abc.z, cOutput))
      );
…
function onRapid5D(_x, _y, _z, _a, _b, _c) {
…
  var a = aOutput.format(_a);
  var b = bOutput.format(_b);
  var c = cOutput.format(getDirectionalABC(previousABC.z, _c, cOutput));
…
function onLinear5D(_x, _y, _z, _a, _b, _c, feed) {
…
  var a = aOutput.format(_a);
  var b = bOutput.format(_b);
  var c = cOutput.format(getDirectionalABC(previousABC.z, _c, cOutput));
…
```

Adding Calls to getDirectionalABC

Similar to setting the *previousABC* variable in the *setWorkPlane* function, the call to *getDirectionalABC* should only be made when the actual rotary axes positions are output and should not be used when Euler angles are output.

7.3 Adjusting the Points for Rotary Heads

While the post kernel can handle adjusting the tool path for rotary tables, it does not currently have support for adjusting the tool path for rotary heads. This is a feature that you will need to add yourself and this section will guide you through it.

First, you will need to make sure that you set the proper tool path adjustments so that the tool tip data is provided as input to the post processor using the *optimizeMachineAngles2* function as described earlier in this chapter.

```
optimizeMachineAngles2(0);  // Set to 2 if the machine is a head/table configuration without TCP
```

Setting Tool Tip Input

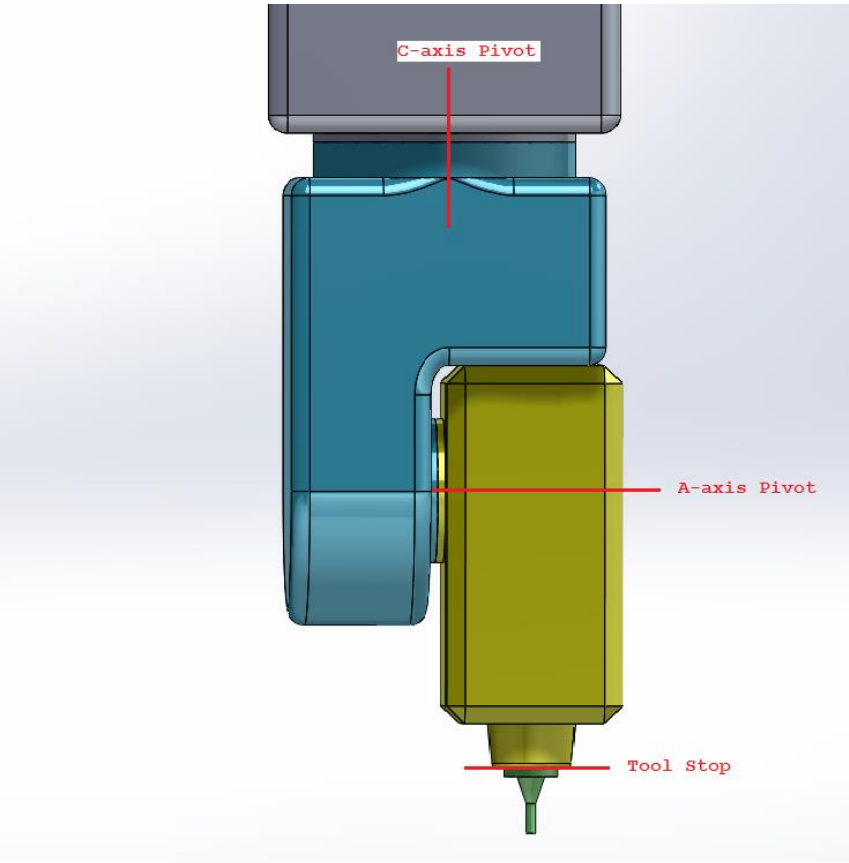
Now if your machine supports TCP programming, you can skip the rest of this section, as this is all that is needed (except for outputting the code to enable TCP programming if required by the machine control).

If your machine consists of a rotary table and a rotary head, then the value passed to *optimizeMachineAngles2* should be 2, so that the tool path is adjusted for the table. The distance to the pivot point(s) of the head is defined in the *offset* parameter of the *createAxis* command.

```
var bAxis = createAxis({
  coordinate:1,
  table:false,
  axis:[1, 0, 0],
  offset[0, 0, toPreciseUnit(27.5, MM)], // distance from tool stop to B-axis pivot
  range:[-180.00, 180.00]
});

var cAxis = createAxis({
  coordinate:2,
  table:false,
  axis:[0, 0, 1],
  offset:[toPreciseUnit(63.7), 0, toPreciseUnit(15.5)], // distance from B-axis pivot to C-axis pivot
  range:[-360.00, 360.00], cyclic:false
});
machineConfiguration = new MachineConfiguration(bAxis, cAxis);
```

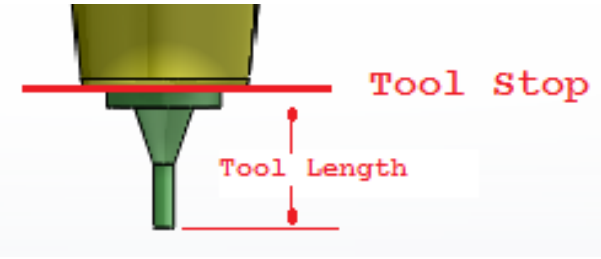
Define Distances to Pivot Point(s) of Rotary Heads

Remember the head slave/rider axis is defined first and then the head master/carrier axis. When the master and slave heads share a common pivot point, then only the offset for the slave axis needs to be defined. This offset is defined from the tool stop position to the pivot point. When the pivot points are different, the master axis offset is defined as the offset from the slave pivot point. Most machines will use a common pivot point for both rotary axes.



Rotary Head Pivot References

One other offset that needs to be addressed is the distance from the tool tip to the tool stop. This value is either a fixed number if tool length compensation is supported, or is defined by the tool body length.



Tool Length Definition

You will need to add a function that adjusts the points for the rotary head. This is typically accomplished by translating the tool tip point up the tool axis to the pivot point of the rotary head and then back down along the spindle axis to the virtual tool tip when the head is at 0 degrees and the control supports tool length compensation along the spindle axis. If tool length compensation is not supported, you will need to translate the tool tip to the pivot point of the rotary head and output this position. The following code is used to calculate the output coordinates for a rotary head and can be included in your post processor.

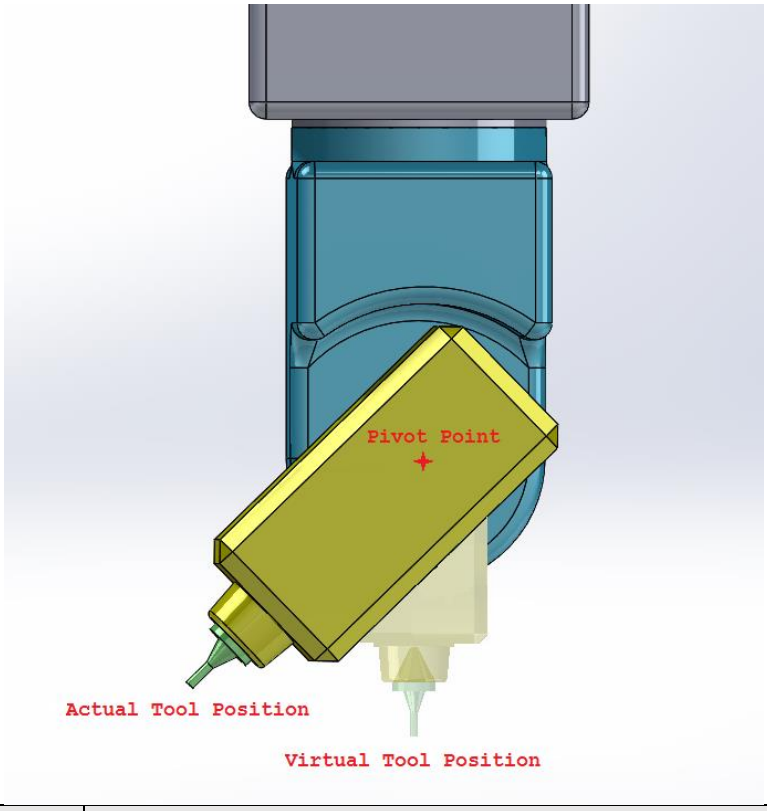
```
var TCP_TOOL = 0; // returns tool tip, assumes optimized coordinates as input
var TCP_OPTIMIZED = 1; // returns optimized coordinates, assumes tool tip as input
var OPTIMIZE_NONE = 0; // TCP mode
var OPTIMIZE_HEADS = 1; // optimize for heads only

var useOptimized = OPTIMIZE_HEADS; // set for for non-tcp controls
var usePivotPoint = false; // true outputs head pivot point, use false with tool length compensation
var toolLength = 0; // set to length of tool to be added to pivot distance

function getOptimizedPosition(_xyz, _abc, _which) {
  if (useOptimized == OPTIMIZE_NONE) {
    return _xyz;
  }

  var xyz = new Vector(_xyz.x, _xyz.y, _xyz.z);
  var abc = new Vector(_abc.x, _abc.y, _abc.z);
  var rotaryAxis =  new Array(
    machineConfiguration.getAxisU(),
    machineConfiguration.getAxisV(),
    machineConfiguration.getAxisW()
  );
  var reverse = _which == TCP_TOOL;

  if (!reverse) {
    xyz = getOptimizedHeads(xyz, abc, rotaryAxis, reverse);
  } else {
    xyz = getOptimizedHeads(xyz, abc, rotaryAxis, reverse);
  }
  return xyz;
}

/** adjust points for heads */
function getOptimizedHeads(_xyz, _abc, _rotaryAxis, _reverse) {
  var xyz = new Vector(_xyz.x, _xyz.y, _xyz.z);
  var first = true;
  var spindleVector = machineConfiguration.getSpindleAxis();
  displacement = new Vector(0, 0, 0);
  var tl = 0;
  if (typeof toolLength == "number") {
    tl = toolLength;
  }
  for (var i = 0; i < 3; ++i) {
    if (_rotaryAxis[i].isEnabled() && _rotaryAxis[i].isHead()) {
      var offset = _rotaryAxis[i].getOffset();
      if (_reverse) {
```

```
      offset = offset.negated;
    }
    var toolDisplacement = Vector.product(spindleVector, tl);
    if (_reverse) {
     toolDisplacement = toolDisplacement.negated;
    }
    displacement = Vector.sum(displacement, offset);
    displacement = Vector.sum(displacement, toolDisplacement);
    displacement =
_rotaryAxis[i].getAxisRotation(_abc.getCoordinate(_rotaryAxis[i].getCoordinate())).multiply(displac
ement);
    if (!usePivotPoint) {
     displacement = Vector.diff(displacement, offset);
     displacement = Vector.diff(displacement, toolDisplacement);
    }
    tl = 0;
   }
  }
  xyz = Vector.sum(xyz, displacement);
  return xyz;
}
```

getOptimizedPosition Function Adjusts Points for Rotray Heads

The code is generic in nature and should not have to be modified when inserting it into a post processor, but there are settings at the top of the code that may have to be changed to match your requirements.

Setting	Description
useOptimized	Should be set to OPTIMIZE_HEADS to adjust the tool end points for the head rotations.
usePivotPoint	*true* = the tool locations will be adjusted to be at the pivot point of the rotary head. *false* = the tool locations will be adjusted to be at the virtual tool endpoint (as if the rotary head angles are at 0 degrees). *false* is used for machines that support tool length compensation with multi-axis moves, where the virtual tool position will be output.
toolLength	The length of the tool. This can be set to a fixed value when tool length compensation is used. When the pivot point is output, then this value should reflect the distance of the tool tip to the tool stop position.



Tool Position	Output when …
Actual Tool Position	TCP is supported
Virtual Tool Position	Tool length compensation is supported
Pivot Point	Neither TCP nor tool length compensation support for multi-axis moves

Output Location Depends on Machine Requirements

Calls to the *getOptimizedPosition* need to be inserted wherever the tool position is output, for example in the *onRapid5D* and *onLinear5D* functions. The same calls may have to be added to the 3-axis functions, output of initial point, *onLinear*, *onRapid*, *onCyclePoint*, and *onCircular*, depending on if tool length compensation is supported in the control.

```
function onRapid5D(_x, _y, _z, _a, _b, _c) {
  if (pendingRadiusCompensation >= 0) {
    error(localize("Radius compensation mode cannot be changed at rapid traversal."));
    return;
  }

  // adjust points for heads
  var xyz = getOptimizedPosition(new Vector(_x, _y, _z), new Vector(_a, _b, _c),
    TCP_OPTIMIZED);

  var x = xOutput.format(xyz.x);
  var y = yOutput.format(xyz.y);
  var z = zOutput.format(xyz.z);
```

Add the Call to getOptimizedPosition to the 5-axis Motion Functions

If the control requires the pivot point locations, then you will need to define the tool length each time a tool is loaded and add the call to *getOptimizedPosition* to all functions that output the tool position, including the *onSection* (output of initial position), *onRapid*, *onLinear*, *onCyclePoint*, and *onCircular* functions.

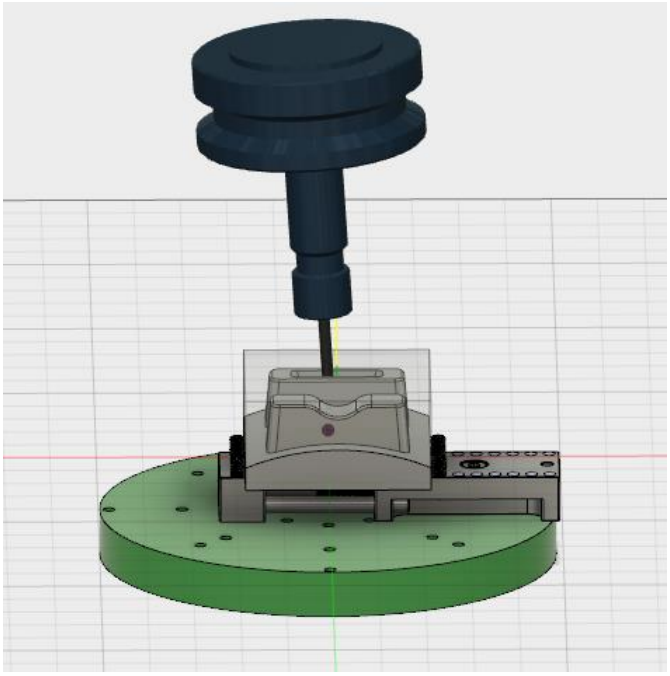
```
toolLength = tool.bodyLength;
var xyz = getFramePosition(currentSection.getInitialPosition());
var initialPosition = getOptimizedPosition(xyz, abc, TCP_OPTIMIZED);
```

Set the Tool Length for Pivot Point Output

7.4 Handling the Singularity Issue in the Post Processor

The post processor kernel handles the problem when the tool axis direction approaches the singularity of the machine. The singularity is defined as the tool axis orientation that is perpendicular to a rotary axis, either a table or head. When the tool direction approaches the singularity, you may notice that the rotary axis can start to swing violently even if there is only a small deviation in the tool axis. If you can imagine a machine with an A-axis trunnion carrying a C-axis table and the tool axis is 0, sin(.001), cos(.001). This causes the output rotary positions to be A.001 C0. Now if the rotary axis changes to 0, sin(.001), cos(.001), a change of less than .002 degrees you will notice that the rotary positions to be A.001 C90. You can see where a very small directional change in the tool axis (<.002) will cause a 90-degree change in the C-axis.

The singularity logic in the kernel will massage the tool axis direction to keep the tool within tolerance and minimize the rotary axis movement in these cases. A safeguard that linearizes the moves around the singularity has also been implemented. This linearization will add tool locations as necessary to keep the tool endpoint within tolerance of the part.



Tool Direction Approaching the Singularity

There are settings in the post processor that manage how the singularity issue is handled. These settings are defined using the following command.

```
machineConfiguration.setSingularity(adjust, method, cone, angle, tolerance, linearizationTolerance)
```

Variable	Description
adjust	Set to *true* to enable singularity optimization within the post processor. Singularity optimization includes the ability to adjust the tool axis to minimize singularity issues (large rotary axis movement when the tool axis approaches perpendicularity to a rotary axis) and the linearization of the moves around the singularity to keep the tool endpoint within tolerance. The default is *true*.
method	When set to SINGULARITY_LINEARIZE_OFF it disables the linearization of the moves to keep the tool endpoint within tolerance of the programmed tool path around the singularity. SINGULARITY_LINEARIZE_ROTARY will linearize the moves around the singularity. Additional points are added to keep the tool within the specified tolerance and is optimized for revolved movement as if the tool were moving around a cylinder or other revolved feature. SINGULARITY_LINEARIZE_LINEAR will also add additional points to keep the tool within tolerance, but will keep the tool endpoint moving in a straight line. The default is SINGULARITY_LINEARIZE_ROTARY.
cone	Specifies the angular distance that the tool axis vector must be within in reference to the singularity point before the singularity logic is activated. This is usually a small value (less than 5 degrees), since the further away the tool

Variable	Description
	axis is from the singularity, the less noticeable the fluctuations in the rotary axes will be and the less benefit this feature will provide. This parameter is specified in radians and the default value is .052 (3 degrees).
angle	The minimum angular delta movement that the rotary axes must move prior to considering adjusting the tool axis vector for singularity optimization. This limit is used to keep from adjusting the tool axis vector when the rotary axes do not fluctuate greatly. This is typically set to a value of 10 degrees or more. This parameter is in radians and the default value is .175 (10 degrees).
tolerance	The tolerance value used to keep the tool within tolerance when the tool axis is adjusted to minimize rotary axis movement around the singularity. The default value is .04mm (.0015in).
linearizationTolerance	The tolerance value to use when additional points are added to keep the tool endpoint within tolerance of the programmed move when the tool axis is near the singularity. The default value is .04mm (.0015in).

The default settings are valid for most tool paths, but this command allows for some tweaking in special cases where you want to fine tune the output.

7.5 Rewinding of the Rotary Axes when Limits are Reached

The post processor kernel will select the starting angles of the rotary axes based on the best possible solution to avoid rewind situations when one of the rotary axes crosses its limits. This is accomplished by scanning the entire operation to see if a rewind of the rotary axes is required due to limit violations and if so adjusting the starting angles of the rotary axes to see if the rewind can be avoided. If a solution to avoid the rewind cannot be found, then the solution that produces the most rotary movement prior to requiring a rewind will be used.

The best possible solution for the rotary axes is always selected at the start of an operation and when a rewind is required due to a rotary axis crossing the limits, the tool will always stop on the exact limit of the machine, eliminating previous scenarios where a valid solution for the rewinding of the rotary axes could not always be found.

When a rewind is required there is a group of functions that can be added to the custom post processor to handle the actual rewinding of the affected rotary axis. This code can be easily copied into your custom post processor and modified to suit your needs with just a little bit of effort.

One setting that is very important when defining a rotary axis is the *cyclic* parameter in the call to *createAxis*. Where in older versions of the post kernel you would set the *cyclic* parameter to enable the ability to rewind the rotary axes when the limits were reached, *cyclic* is now considered synonymous with continuous, meaning that this axis has no limits and will not be considered when determining if the rotary axes have to be repositioned to stay within limits. The *range* specifier used in conjunction with a cyclic axis defines the output limits of a rotary axis, for example specifying a range of [0,360] will cause all output angles for this axis to be output between 0 and 360 degrees. The range for a non-cyclic axis defines the actual physical limits of that axis on the machine and are used to determine when a rewind is

required. Please note that the physical limits of the machine may be a numeric limit of the control instead of a physical limit, such as 9999.9999.

Another important setting is the *reset* parameter, which allows you to define the starting angle at the start of an operation and after a rewind of the axes has occurred. By default, the post engine will use the ending angle of the previous multi-axis operation. Some controls allow for the rotary axis encoder to be reset so that the stored angle is reset to be within the 0-360 degrees without unwinding the axis. In this case you will want to issue the proper codes to reset the axis encoder, for example G28 C0, and specify *reset:3* when you create the axis.

Now on to how you can implement the *onRewindMachine* capabilities in your post. First, copy the code from a post processor that contains these functions, such as the *haas umc-750.cps* post processor. All the lines between and including the following lines should be copied.

```
// Start of onRewindMachine logic
…
// End of onRewindMachine logic
```

Copy this Code to your Custom Post Processor

This code is generic in nature and will work with all machine configurations; table/table, head/head, and head/table. Because of this most of the functions included in this code will not have to be modified by you. The rest of this section describes the changes that you may have to make to customize the rewind logic for your machine.

The *safeRetractDistance* value is added to the distance that the tool will be retracted out of the part prior to rewinding the rotary axis. The tool will be retracted past the stock of the part plus this value.

```
properties = {
…
  safeRetractDistance: 0.0 // distance to add to retract distance when rewinding rotary axes
}
```

Add *safeRetractDistance* to Properties Table

The variables at the top of the rewind code determine if rewinds are supported, the feedrates, and stock expansion.

```
var performRewinds = false; // enables the onRewindMachine logic
var safeRetractFeed = (unit == IN) ? 20 : 500;
var safePlungeFeed = (unit == IN) ? 10 : 250;
var stockAllowance = (unit == IN) ? 0.1 : 2.5;
```

Variable	Description
performRewinds	When set to *false* an error will be generated when a rewind of a rotary axis is required. Setting it to true will enable the rewind logic to be executed.
safeRetractFeed	Specifies the feedrate to retract the tool prior to rewinding the rotary axis.

safePlungeFeed	Specifies the feedrate to plunge the tool back into the part after rewinding the rotary axis.
stockAllowance	The tool will retract past the defined stock by default. You can expand the defined stock on all sides by the *stockAllowance* value.

Variables that Control Tool Retraction

The first function that is unique for different machines is the *onRewindMachineEntry* function, which is used to either override or supplement the standard rewind logic. It will simply return *false* when the standard rewind logic of retracting the tool, repositioning the rotary axes, and repositioning the tool is desired. Code can be added to this function for controls that just require the encoder to be reset or to output the new rotary axis position when the control will automatically track the tool with the rotary axis movement. The following example resets the C-axis encoder on a Hass machine when it is currently at a multiple of 360 degrees and the B-axis does not change.

```
/** Allow user to override the onRewind logic. */
function onRewindMachineEntry(_a, _b, _c) {
  // reset the rotary encoder if supported to avoid large rewind
  if (properties.rewindCAxisEncoder) {
    if ((abcFormat.getResultingValue(_c) == 0) && !abcFormat.areDifferent(getCurrentDirection().y,
_b)) {
      writeBlock(gAbsIncModal.format(91), gFormat.format(28), "C" + abcFormat.format(0));
      writeBlock(gAbsIncModal.format(90));
      return true;
    }
  }
  return false;
}
```

Sample Code to Reset Encoder Instead of Rewinding C-axis

Returning a value of *true* designates that the *onRewindMachineEntry* function performed all necessary actions to reposition the rotary axes and the retract/reposition/plunge sequence will not be performed. Returning *false* will process the retract/reposition/plunge sequence normally.

The *moveToSafeRetractPosition* function controls the move to a safe position after the tool is retracted from the part and before the rotary axes are repositioned. It will typically move to the home position in Z and optionally in X and Y using a G28 or G53 style block. You should find similar code to retract the tool when positioning the rotary axes for a 3+2 operation and in the *onClose* function, which positions the tool at the end of the program. You should use the same logic found in these areas for the *moveToSafeRetractPosition* function.

```
/** Retract to safe position before indexing rotaries. */
function moveToSafeRetractPosition(retracted)
  if (!retracted) {
    writeBlock(gFormat.format(28), gAbsIncModal.format(91), "Z" +
      xyzFormat.format(machineConfiguration.getRetractPlane())); // retract
    writeBlock(gAbsIncModal.format(90));
```

```
    zOutput.reset();
  }
}
```

Move to a Safe Position Prior to Repositioning Rotary Axes

The *returnFromSafeRetractPosition* function controls the move back to the position of the tool at the original retract location past the stock. This function is called after the rotary axes are repositioned.

```
/** Return from safe position after indexing rotaries. */
function returnFromSafeRetractPosition(position) {
  forceXYZ();
  xOutput.reset();
  yOutput.reset();
  zOutput.disable();
  onRapid(position.x, position.y, position.z);
  zOutput.enable();
  onRapid(position.x, position.y, position.z);
}
```

Return from Safe Position after Repositioning Rotary Axes

These should be all the areas of the rewind code that should be modified, the rest is generic for all machine configurations.

7.6 Multi-Axis Feedrates

During multi-axis contouring moves, the machine control will typically expect the feedrate numbers to be either in Inverse Time or some form of Degrees Per Minute. Inverse Time feedrates are simply the inverse of the time that the move takes, i.e. *1 / movetime*. If your control supports both Inverse Time and Degrees Per Minute feedrates, it is recommended that you use Inverse Time as this is the most accurate. Please note that if your machine supports TCP (Tool Control Point) programming, then it probably supports direct Feed Per Minute (FPM) feedrates during multi-axis contouring moves and does not require either Inverse Time or DPM feedrates.

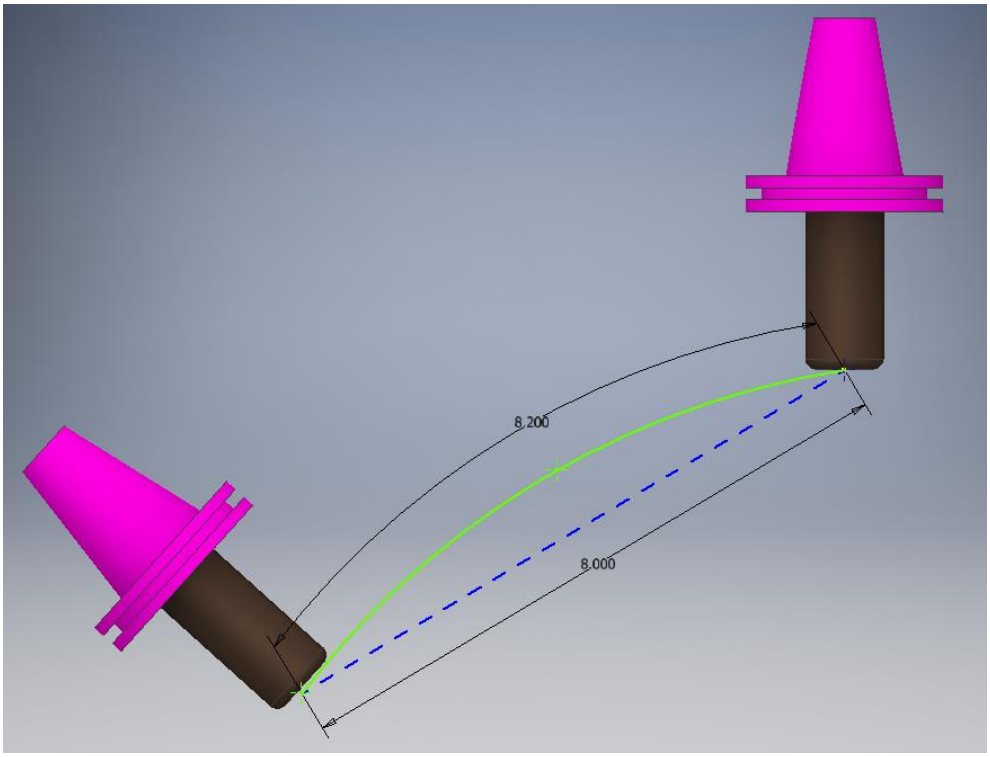
To implement multi-axis feedrate support into your post, you will first need to copy the code from a post processor that already supports this feature, such as the *haas trunnion.cps* post processor. All the lines between and including the following lines should be copied.

```
// Start of multi-axis feedrate logic
…
// End of multi-axis feedrate logic
```

Copy this Code to your Custom Post Processor

This code is generic in nature and will work with all machine configurations; table/table, head/head, and head/table. Because of this most of the functions included in this code will not have to be modified by you, though you will have to modify other sections of the post processor to fully implement this feedrate logic.

One capability of the multi-axis feedrate calculation is that it considers the actual tool tip movement in reference to the rotary axes movement and not just the straight-line movement along the programmed tool tip, creating more accurate multi-axis feedrates. In the following picture the move along the arc caused by the movement of the rotary axis (green arc) is used in the calculation instead of the straight-line move generated by HSM (blue line).



Actual Tool Path on Machine is Used in Feedrate Calculations

The rest of this section describes the changes that you may have to make to customize the multi-axis feedrate logic for your machine.

If Inverse Time feedrates are supported you will need to create the *inverseTimeOutput* variable at the top of the post processor code and if the accuracy of the Inverse Time feedrates is different than the standard FPM feedrate you will also need to create a new format to associate with it.

```
var inverseFormat = createFormat({decimals:4, forceDecimal:true});
…
var inverseTimeOutput = createVariable({prefix:"F", force:true}, feedFormat);
```

Create *inverseTimeOutput* Variable

The variables at the top of the multi-axis feedrate code define variables used in the calculation of Inverse Time and DPM feedrates.

```
var dpmBPW = (unit == IN) ? 0.1 : 1.0; // ratio of rotary accuracy to linear for DPM calculations
var inverseTimeUnits = 1.0; // 1.0 = minutes, 60.0 = seconds
```

```
var maxInverseTime = 45000; // maximum value to output for Inverse Time feeds
var maxDPM = 9999.99; // maximum value to output for DPM feeds
var useInverseTimeFeed = true; // use 1/T feeds
var previousDPMFeed = 0; // previously output DPM feed
var dpmFeedToler = 0.5; // tolerance to determine when the DPM feed has changed
// var previousABC = new Vector(0, 0, 0); // previous ABC if used in post, don't define if not used
var forceOptimized = undefined; // used to override optimized-for-angles points (XZC-mode)
```

Variable	Description
dpmBPW	Defines the pulse weight ratio for the rotary axes when DPM feedrates are output as a combination of linear and rotary movements. The pulse weight is a scale factor based on the rotary axes accuracy compared to the linear axes accuracy. For example, it should be set to .1 when the linear axes are output on .0001 increments and the rotary axes on .001 increments.
inverseTimeUnits	Defines the unit of time for Inverse Time feedrates. Specify 1.0 for minutes or 60.0 for seconds.
maxInverseTime	Specifies the maximum value that can be output for Inverse Time feedrates.
maxDPM	The maximum value that can be output for DPM feedrates.
useInverseTimeFeed	Can be set to *true* for Inverse Time feedrates or *false* for DPM feedrates when only one of the formats is supported.
previousDPMFeed	Used to determine when the DPM feedrate should be output. Should not be changed.
dpmFeedToler	Determines when the DPM feedrate should be output. The calculated feedrate number will not be output unless it changes by more than this value.
previousABC	Some post processors require that the previous ABC output positions be maintained inside the post processor, for example post processors that output the rotary axes on a rotary scale with the sign of the value specifying the direction of rotation. In this case, *previousABC* should be defined. If the post processor does not maintain this variable, then it should be commented out, otherwise it will adversely affect the calculations of the multi-axis feedrates.
forceOptimized	This variable is used for Mill/Turn machines where multi-axis programming uses the points adjusted for the C-axis and XZC programming uses the input tool endpoint positions. It is set to *false* when calculating the XZC mode positions and to *undefined* for all other positions. It must be set to *undefined* when created.
headOffset	For machines that have a rotary head, the *headOffset* variable can be defined. It contains the fixed pivot length combined with the tool length and is used to calculate the length of the move. Basically, it is the distance from the tool tip to the pivot point of the head. This variable is typically defined in post processors that support rotary heads.

Variables that Control Multi-Axis Feedrate Calculations

getMultiaxisFeed is the controlling function used for multi-axis feedrate calculations. It retrieves the total move length and determines whether to use Inverse Time or DPM feedrates.

```
/** Calculate the multi-axis feedrate number. */
function getMultiaxisFeed(_x, _y, _z, _a, _b, _c, feed) {
  var f = {frn:0, fmode:0};
  if (feed <= 0) {
    error(localize("Feedrate is less than or equal to 0."));
    return f;
  }

  var length = getMoveLength(_x, _y, _z, _a, _b, _c);

  if (useInverseTimeFeed) { // inverse time
    f.frn = inverseTimeOutput.format(getInverseTime(length.tool, feed));
    f.fmode = 93;
    feedOutput.reset();
  } else { // degrees per minute
    f.frn = feedOutput.format(getFeedDPM(length, feed));
    f.fmode = 94;
  }
  return f;
}
```

getMultiAxisFeed Function

The object returned from the *getMoveLength* function returns the values that may be needed by different post processors to calculate the Inverse Time and DPM feedrates. The following table lists the variables calculated and returned by the *getMoveLength* function.

Variable	Description
abc	Delta movement for each rotary axis, returned as a Vector.
abcLength	Combined rotary delta movement.
radius	Calculated radius for each rotary axis, returned as a Vector.
tool	Calculated tool endpoint movement along the actual tool path.
xyz	Delta movement for each linear axis, returned as a Vector.
xyzLength	Combined linear delta movement.

Move LengthVariables

If your machine supports both Inverse Time and DPM feedrates, you can add the *useInverseTime* variable to the property table at the top of the post processor and allow the user to choose the multi-axis feedrate format.

```
properties = {
…
  useInverseTime: true // true = inverse time feedrates, false = degrees per minute feedrates
}
```

```
…
propertyDefinitions = {
…
  useInverseTime: {title:"Use inverse time feedrates",
   description:"'Yes' enables inverse time feedrates, 'No' outputs DPM feedrates.", type:"boolean"}
```

Optionally Add *useInverseTime* to Properties Table

The only other function that you may need to modify in the included code is the *getFeedDPM* function that calculates the Degrees Per Minute feedrates. It contains calculations for standard DPM feedrates and combination FPM/DPM feedrates based on the combined movement of the linear and rotary axes, sometimes referred to as Pulses Per Minute. There are some controls that use a proprietary calculation for DPM feedrates, such as the Fadal control. In this case there is a defined block where you can add the control specific calculation.

```
/** Calculate the DPM feedrate number. */
function getFeedDPM(_moveLength, _feed) {
  if ((_feed == 0) || (_moveLength.tool < 0.0001) || (toDeg(_moveLength.abcLength) < 0.0005)) {
    previousDPMFeed = 0;
    return _feed;
  }
  var moveTime = _moveLength.tool / _feed;
  if (moveTime == 0) {
    previousDPMFeed = 0;
    return _feed;
  }
  var dpmFeed;
  var tcp = !getOptimizedMode() && (forceOptimized == undefined);   // false for rotary heads
  if (tcp) { // TCP mode is supported, output feed as FPM
    dpmFeed = _feed;
  } else if (false) { // set to 'true' for standard DPM
    dpmFeed = Math.min(toDeg(_moveLength.abcLength) / moveTime, maxDPM);
    if (Math.abs(dpmFeed - previousDPMFeed) < dpmFeedToler) {
      dpmFeed = previousDPMFeed;
    }
  } else if (false) { // set to 'true' for combination FPM/DPM
    var length = Math.sqrt(Math.pow(_moveLength.xyzLength, 2.0) +
      Math.pow((toDeg(_moveLength.abcLength) * dpmBPW), 2.0));
    dpmFeed = Math.min((length / moveTime), maxDPM);
    if (Math.abs(dpmFeed - previousDPMFeed) < dpmFeedToler) {
      dpmFeed = previousDPMFeed;
    }
  } else { // machine specific calculation
    var length = Math.sqrt(Math.pow(_moveLength.tool, 2.0) +
      Math.pow(_moveLength.xyzLength, 2.0));
    dpmFeed = toDeg(_moveLength.abcLength) / (length / _feed);
    if (Math.abs(dpmFeed - previousDPMFeed) < dpmFeedToler) {
```

```
      dpmFeed = previousDPMFeed;
    }
  }
  previousDPMFeed = dpmFeed;
  return dpmFeed;
}
```

Standard DPM Calculation in *getFeedDPM* Function

Now there are other areas of the post processor that need to be changed to support these feedrate modes. First, the onLinear5D function must have support added to call the function and output the correct feedrate codes.

```
function onLinear5D(_x, _y, _z, _a, _b, _c, feed) {
  …
  // get feedrate number
  var f = {frn:0, fmode:0};
  if (a || b || c) {
    f = getMultiaxisFeed(_x, _y, _z, _a, _b, _c, feed);
    if (useInverseTimeFeed) {
      f.frn = inverseTimeOutput.format(f.frn);
    } else {
      f.frn = feedOutput.format(f.frn);
    }
  } else {
    f.frn = feedOutput.format(feed);
    f.fmode = 94;
  }
  if (x || y || z || a || b || c) {
    writeBlock(gFeedModeModal.format(f.fmode), gMotionModal.format(1), x, y, z, a, b, c, f.frn);
  } else if (f.frn) {
    if (getNextRecord().isMotion()) { // try not to output feed without motion
      feedOutput.reset(); // force feed on next line
    } else {
      writeBlock(gFeedModeModal.format(f.fmode), gMotionModal.format(1), f.frn);
    }
```

***onLinear5D* Required Changes**

You will need to reset the feedrate mode to FPM either at the end of the multi-axis operation or on a standard 3-axis move. It is much easier to do this at the end of the section, otherwise you would have to modify all instances that output feedrates, such as in onLinear, onCircular, onCycle, etc.

```
function onSectionEnd() {
…
  if (currentSection.isMultiAxis()) {
    writeBlock(gFeedModeModal.format(94)); // inverse time feed off
  }
```

Reset FPM Mode in *onSectionEnd*

```
writeBlock(gFeedModeModal.format(94), gMotionModal.format(1), gFormat.format(40), x, y, z, f);
```

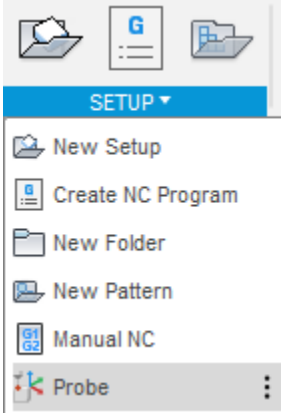
Optionally Reset FPM Mode in All Output Blocks with Feedrates

8 Adding Support for Probing

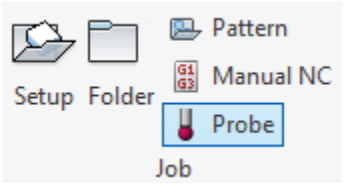
Fusion and HSM have support for multiple styles of probing operations, including WCS Probing, Geometry Probing, and Surface Inspection. While the probing capabilities are supported by many of the library post processors, they are not supported by all of them and custom post processors may not have these capabilities. This chapter discusses the required changes to a post processor to support the probing operations.

8.1 WCS Probing

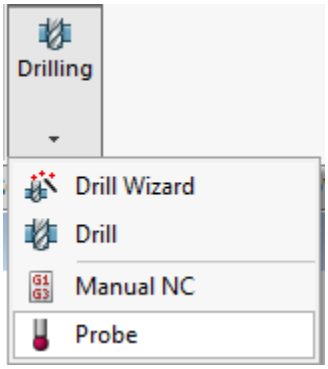
WCS Probing is defined as probing operations that are used to probe the part for the purpose of defining a Work Coordinate System. While all Autodesk CAM products support WCS Probing, you will find these operations in a different area of the interface for each of the products.



Fusion 360



Inventor HSM



Inventor HSM

You can check the post processor you are working with to see if it supports WCS Probing. The easiest method is to try to run a probing operation against the post, the post will fail if probing is not supported. You may see an error message complaining about the spindle speed being out of range (probe operations do not turn on the spindle) or a message that states that the probing cycle must be handled in the post processor.

```
################################################################################
Error: Spindle speed out of range.
Error at line: 735
Error in operation: 'WCS Probe1'
Failed while processing onSection() for record 261.
################################################################################
```

Spindle Speed Error Message

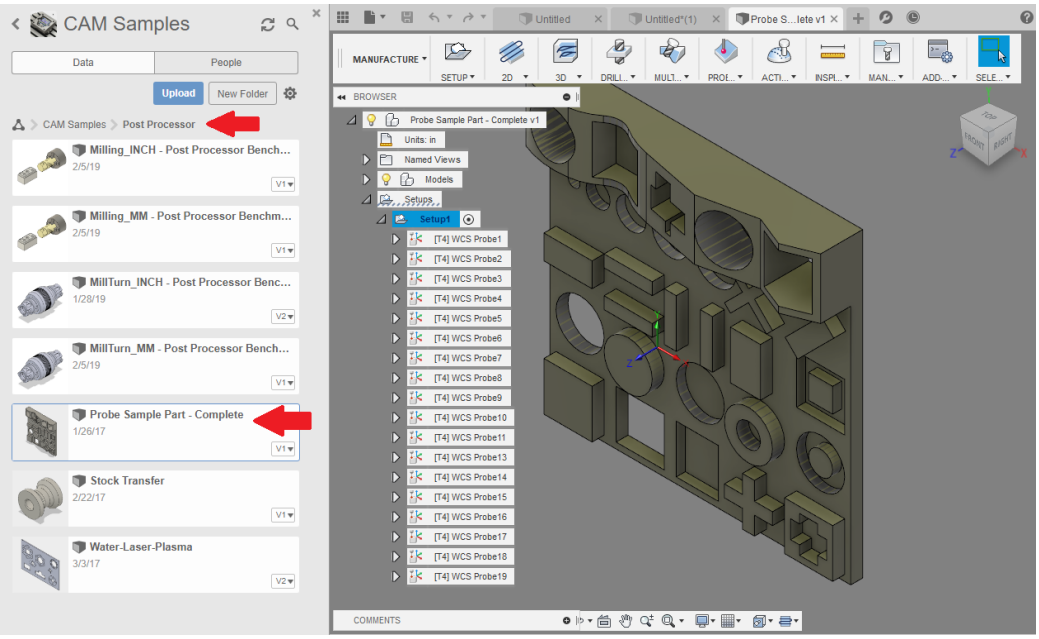
```
################################################################################
Error: The probe cycle 'probing-xy-outer-corner' is machine specific and must always be handled in
the post configuration.
Error in operation: 'WCS Probe1'
Failed while processing onCycle() for record 280.
################################################################################
```

Machine Specific Error Message

If you receive either of these messages, then probing is not supported in your post processor and you will need to add it.

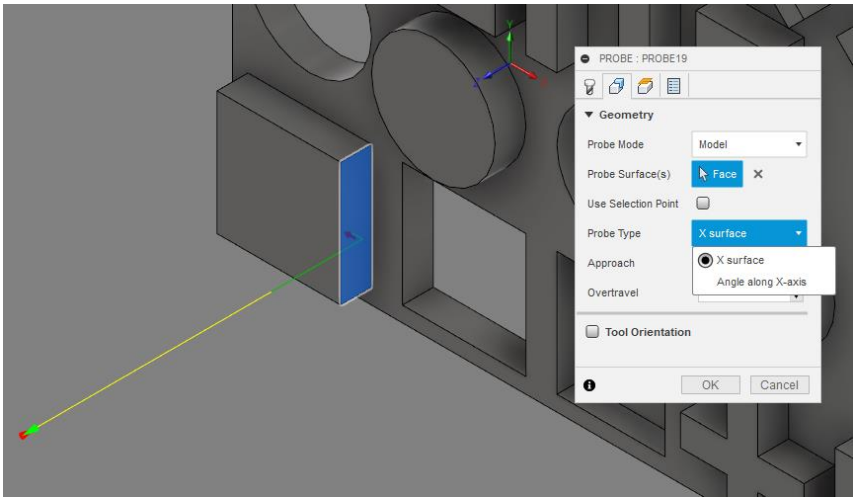
8.1.1 Probing Operations

There is a sample model available for testing the probing logic in a post processor. In Fusion it is contained int the CAM Samples/Post Processor folder. This model contains a part designed for testing probing cycles using the available WCS Probing operations.



Sample Probing Part

One thing you will notice when creating a probing operation is that interface is intelligent enough to only give you the probing operation types that apply to the type of geometry selected. For example, if you select a planar face perpendicular to the X-axis, then the only operations available to you are the *X surface* and *Angle along X-axis* operations.



Intelligent Probe Selection

The WCS Probing operations are considered a canned cycle in the post processor and therefore are output in the *onCyclePoint* function, with the probe type being stored in the *cycleType* variable. The following table lists the available probing operations. You should note that probing cycles cannot be expanded and must be handled in the post processor, either by performing the cycle or by giving an error.

cycleType	Description
probing-X	Probes a wall perpendicular to the X-axis.
probing-Y	Probes a wall perpendicular to the Y-axis.
probing-Z	Probes a wall perpendicular to the Z-axis.
probing-x-wall	Probes a wall thickness in the X-axis
probing-y-wall	Probes a wall thickness in the Y-axis
probing-x-channel	Probes the open distance between two walls in the X-axis
probing-y-channel	Probes the open distance between two walls in the Y-axis
probing-x-channel-with-island	Probes the open distance between two walls with an island between the walls in the X-axis
probing-y-channel-with-island	Probes the open distance between two walls with an island between the walls in the Y-axis
probing-xy-circular-boss	Probes the outer wall of a circular boss
probing-xy-circular-hole	Probes the inner wall of a circular hole
probing-xy-circular-hole-with-island	Probes the inner wall of a circular hole with an island in the hole
probing-xy-rectangular-boss	Probes the outer walls of a rectangular protrusion
probing-xy-rectangular-hole	Probes the inner walls of a rectangular hole
probing-xy-rectangular-hole-with-island	Probes the inner walls of a rectangular hole with an island in the hole
probing-xy-inner-corner	Probes an inner corner. Modifies the origin and rotation of the part.

cycleType	Description
probing-xy-outer-corner	Probes an outer corner. Modifies the origin and rotation of the part.
probing-x-plane-angle	Probes a wall at an angle to the X-axis. Modifies the rotation of the part.
probing-y-plane-angle	Probes a wall at an angle to the Y-axis. Modifies the rotation of the part.

Probing Cycles

The parameters defined in the WCS Probing operation are passed to the cycle functions using the *cycle* object. The following variables are available and are referenced as 'cycle.*parameter'*.

Parameter	Description
approach1	The distance from the contact point at which the probe starts to approach the part.
approach2	The distance from the contact point at which the probe starts to approach the second face of a multi-face operation.
depth	The position along the probe axis to touch the part.
probeClearance	The height the probe rapids to on its way to the start of the probing and the position it returns to after the probing operation is finished.
probeOvertravel	The maximum distance the probe can move beyond the expected contact point and still record a measurement.
probeSpacing	The probe spacing between points on the selected face for Angle style probing.
retract	The height to retract the probe to at the programmed feedrate.
width1	The width of the boss or hole being probed.
width2	The width of the secondary walls (Y-axis) of a rectangular boss or hole being probed.

Probing Parameters

8.1.2 Adding the Core Probing Logic

Adding WCS Probing support requires the main logic to output the probing cycle, supporting functions, and some logic added to the main sections of the post processor. You should first open a post processor that contains support for probing before starting to add probing to your post processor, since the logic and most of the code will remain the same. Most of the generic post processors use Renishaw style probing Macros (Fanuc, Haas, etc.), but there are also controls that support probing without the use of these Macros, such as the Datron, Heidenhain, and Siemens controls. Be sure to start with closest match to the machine you are creating a post processor for. The examples used in this chapter use the code for the Renishaw style probing Macros.

The following functions support angular probing and may have to be modified to match the requirements of your control. The code shown is for a Fanuc style control. They should be added prior to the *onCyclePoint* function.

```
/**
  Determine if angular probing is supported.
*/
function getAngularProbingMode() {
  if (machineConfiguration.isMultiAxisConfiguration()) {
    if (machineConfiguration.isMachineCoordinate(2)) {
      return ANGLE_PROBE_USE_CAXIS;
    } else {
      return ANGLE_PROBE_NOT_SUPPORTED;
    }
  } else {
    return ANGLE_PROBE_USE_ROTATION;
  }
}

/**
  Output rotation offset based on angular probing cycle.
*/
function setProbingAngle() {
  if ((g68RotationMode == 1) || (g68RotationMode == 2)) { // Rotate coordinate system for Angle
Probing
    if (!properties.useG54x4) {
      gRotationModal.reset();
      gAbsIncModal.reset();
      writeBlock(
        gRotationModal.format(68), gAbsIncModal.format(90),
        (g68RotationMode == 1) ? "X0" : "X[#135]",
        (g68RotationMode == 1) ? "Y0" : "Y[#136]",
        "Z0", "I0.0", "J0.0", "K1.0", "R[#139]"
      );
      g68RotationMode = 3;
    } else if (angularProbingMode != ANGLE_PROBE_NOT_SUPPORTED) {
      writeBlock("#26010=#135");
      writeBlock("#26011=#136");
      writeBlock("#26012=#137");
      writeBlock("#26015=#139");
      writeBlock(gFormat.format(54.4), "P1");
      g68RotationMode = 0;
    } else {
      error(localize("Angular probing is not supported for this machine configuration."));
      return;
    }
  }
}
```

Probing Parameters

The core logic for probing is in the *onCycle* function. The first part of the code to copy into your post is at the top of the *onCyclePoint* function and defines the WCS code to adjust for the probing operation.

```
  var probeWorkOffsetCode;
  if (isProbeOperation()) {
    if (!useMultiAxisFeatures && !isSameDirection(currentSection.workPlane.forward, new Vector(0,
0, 1)) && (!cycle.probeMode || (cycle.probeMode == 0))) {
      error(localize("Updating WCS / work offset using probing is only supported by the CNC in the
WCS frame."));
      return;
    }

    var workOffset = probeOutputWorkOffset ? probeOutputWorkOffset : currentWorkOffset;
    if (workOffset > 99) {
      error(localize("Work offset is out of range."));
      return;
    } else if (workOffset > 6) {
      probeWorkOffsetCode = probe100Format.format(workOffset - 6 + 100);
    } else {
      probeWorkOffsetCode = workOffset + "."; // G54->G59
    }
  }
```

Setting the WCS code

The highlighted code is controller specific and may have to be modified to match your control. It will be similar to the WCS logic in the *onSection* function.

The code that outputs the probing calls is usually located after the drilling cycle logic in the main switch block. Copy all code that contains the case statements for probing operations.

```
  switch (cycleType) {
  case “drilling”:
  …
  case “probing-x”: // copy from this line to before the “default” case
  …
  default:
```

Calling the Probe Macro

Now add the conditional to ignore subsequent cycle locations. Probing cycles only contain a single location.

```
  // 2nd through nth cycle point
  } else {
    if (isProbeOperation()) {
      // do nothing
```

```
  } else if (cycleExpanded) {
```

Ignore Subsequent Cycle Locations

Add the following code to the *onCycleEnd* function to end the probing operation.

```
function onCycleEnd() {
  if (isProbeOperation()) {
    writeBlock(probeCode.code.format(probeCode.value), "P" + 9810,
zOutput.format(cycle.clearance)); // protected retract move
    writeBlock(probeCode.code.format(probeCode.value), "P" + 9833); // spin the probe off
    setProbingAngle(); // define rotation of part
    // we can move in rapid from retract optionally
  } else {
    …
}
```

8.1.3 Adding the Supporting Probing Logic

There various locations that contain support logic for probing operations in the post processor. Some of this code may already be in your post processor. If the post uses a special format for the output of the Probe WCS code, then you will need to add this format at the top of the post.

```
var probe100Format = createFormat({decimals:3, zeropad:true, width:3, forceDecimal:true});
```

May be Required for Formatting the Probe WCS Code

Add the following definitions to the *fixed settings* section at the top of the post processor.

```
var ANGLE_PROBE_NOT_SUPPORTED = 0;
var ANGLE_PROBE_USE_ROTATION = 1;
var ANGLE_PROBE_USE_CAXIS = 2;
```

Add to Fixed Settings Section

Add the following variables to the collected state section at the top of the post processor.

```
var g68RotationMode = 0;
var angularProbingMode;
```

Add to Collected State Section

The following function and variable definition should be added prior to the *onParameter* function. The *onParameter* function should also have the shown conditional added if it is not there.

```
function isProbeOperation() {
  return hasParameter("operation-strategy") && (getParameter("operation-strategy") ==
"probe");
}
```

```
var probeOutputWorkOffset = 1;

function onParameter(name, value) {
  if (name == "probe-output-work-offset") {
    probeOutputWorkOffset = (value > 0) ? value : 1;
  }
```

Add Prior to and to onParameter Function

The following code needs to be added to the *onSection* function.

```
    if (!isProbeOperation() &&
      (insertToolCall ||
       forceSpindleSpeed ||
       isFirstSection() ||
       (rpmFormat.areDifferent(spindleSpeed, sOutput.getCurrent())) ||
       (tool.clockwise != getPreviousSection().getTool().clockwise))) {
    forceSpindleSpeed = false;
```

Don't Output Spindle Speed with a Probe Tool

```
  if (isProbeOperation()) {
    if (g68RotationMode != 0) {
      error(localize("You cannot probe while G68 Rotation is in effect."));
      return;
    }
    angularProbingMode = getAngularProbingMode();
    writeBlock(probeCode.code.format(probeCode.value), "P" + 9832); // spin the probe on
  }

  // define subprogram
  subprogramDefine(initialPosition, abc, retracted, zIsOutput);

  retracted = false;
}
```

Add at the end of the onSection Function

Coolant should be disabled during probing operations, so make sure that the following conditional is in the *getCoolantCodes* function.

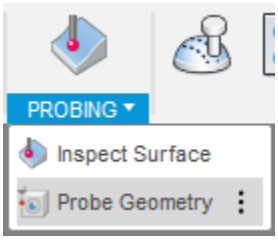
```
function getCoolantCodes(coolant) {
  var multipleCoolantBlocks = new Array(); // create a formatted array to be passed into the outputted
line
  if (!coolants) {
    error(localize("Coolants have not been defined."));
  }
  if (isProbeOperation()) { // avoid coolant output for probing
    coolant = COOLANT_OFF;
```

```
}
```

Disable Coolant for Probing Operations

8.2 Geometry Probing

Geometry Probing behaves similarly to WCS Probing. It is used to measure geometric features on the part during machining. The measured geometric features are checked against specified tolerances for size and position. Based on the result, you can update the tool wear, or instruct the machine to stop machining if the feature is out of tolerance. Geometry Probing is initiated using the *Probe Geometry* operation listed in the PROBING menu.



Geometry Probing Operation

Like in WCS Probing, the parameters defined in the Geometry Probing operation are passed to the cycle functions using the *cycle* object. These are in addition to the parameters defined for WCS Probing, which are also available in Geometry Probing. The following variables are available and are referenced as 'cycle.*parameter'*.

Parameter	Description
angleAskewAction	Set to "stop-message" when *Askew Action* is enabled.
incrementComponent	Increments the output component number when printing the results.
outOfPositionAction	Set to "stop-message" when *Out of Position Action* is enabled.
printResults	Measurements will be printed to a file on the controller when enabled.
toleranceAngle	Used to determine if angular measurement is within tolerance.
tolerancePosition	Used to determine if the positional measurement is within tolerance.
toleranceSize	Used to determine if the size of the feature (hole, boss) is within tolerance.
toolDiameterOffset	Defines the tool diameter offset register used to machine the feature.
toolLengthOffset	Defines the tool length offset register used to machine the feature.
toolWearErrorCorrection	The percentage of the deviation to update the tool wear by.
toolWearUpdateThreshold	The minimum deviation that will trigger a tool wear update.
updateToolWear	Enabled when tool wear compensation should be activated on the controller.
wrongSizeAction	Set to "stop-message" when *Wrong Size Action* is enabled.

Geometry Probing Parameters

To add Geometry Probing to your post you will first need to implement WCS Probing. After this there are only minor changes required to support Geometry Probing. First make sure that the *isProbeOperation* looks like the following.

```
function isProbeOperation() {
  return hasParameter("operation-strategy") && ((getParameter("operation-strategy") == "probe" ||
getParameter("operation-strategy") == "probe_geometry"));
}
```

isProbeOperation Function with Geometry Probing Support

In the *onCyclePoint* function you will need to modify the probing cycles so that they call the *getProbingArguments* function, which formats the parameter output for both WCS and Geometry Probing.

```
case "probing-x":
      forceXYZ();
      // move slowly always from clearance not retract
      writeBlock(gFormat.format(65), "P" + 9810, zOutput.format(z - cycle.depth), getFeed(F)); //
protected positioning move
      writeBlock(
        gFormat.format(65), "P" + 9811,
        "X" + xyzFormat.format(x + approach(cycle.approach1) * (cycle.probeClearance +
tool.diameter/2)),
        "Q" + xyzFormat.format(cycle.probeOvertravel),
        getProbingArguments(cycle, probeWorkOffsetCode) // "S" + probeWorkOffsetCode
      );
      break;
```

Add Call to getProbingArguments to All Probing Operations

Now you will need to add the *getProbingArguments* function prior to the *onCycleEnd* function.

```
function getProbingArguments(cycle, probeWorkOffsetCode) {
  var probeWCS = hasParameter("operation-strategy") && (getParameter("operation-strategy") ==
"probe");
  return [
    (cycle.angleAskewAction == "stop-message" ? "B" + xyzFormat.format(cycle.toleranceAngle ?
cycle.toleranceAngle : 0) : undefined),
    ((cycle.updateToolWear && cycle.toolWearErrorCorrection < 100) ? "F" +
xyzFormat.format(cycle.toolWearErrorCorrection ? cycle.toolWearErrorCorrection / 100 : 100) :
undefined),
    (cycle.wrongSizeAction == "stop-message" ? "H" + xyzFormat.format(cycle.toleranceSize ?
cycle.toleranceSize : 0) : undefined),
    (cycle.outOfPositionAction == "stop-message" ? "M" + xyzFormat.format(cycle.tolerancePosition
? cycle.tolerancePosition : 0) : undefined),
    ((cycle.updateToolWear && cycleType == "probing-z") ? "T" +
xyzFormat.format(cycle.toolLengthOffset) : undefined),
    ((cycle.updateToolWear && cycleType !== "probing-z") ? "T" +
xyzFormat.format(cycle.toolDiameterOffset) : undefined),
```

```
    (cycle.updateToolWear ? "V" + xyzFormat.format(cycle.toolWearUpdateThreshold ?
cycle.toolWearUpdateThreshold : 0) : undefined),
    (cycle.printResults ? "W" + xyzFormat.format(1 + cycle.incrementComponent) : undefined), // 1
for advance feature, 2 for reset feature count and advance component number. first reported result in a
program should use W2.
    conditional(probeWorkOffsetCode && probeWCS, "S" + probeWorkOffsetCode)
  ];
}
```

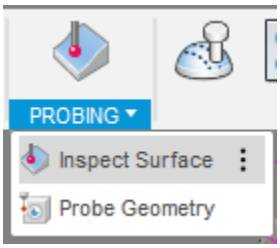
getProbingArguments Function Formats Probing Parameters for Output

8.3 Inspect Surface

The Inspect Surface operation creates a probing strategy that specifies contact points across the surfaces of the model to be measured by a probe while the part is still on the machine tool. The results can then be imported and compared against the model to identify if the manufactured part is in or out of tolerance.

Inspection streamlines the manufacturing process by letting you identify problem areas and decide on any rework needed earlier in the process. It also helps to reduce the need to move parts between the machine tool and a measuring device.

Surface Inspection is initiated using the *Inspect Surface* operation listed in the PROBING menu.



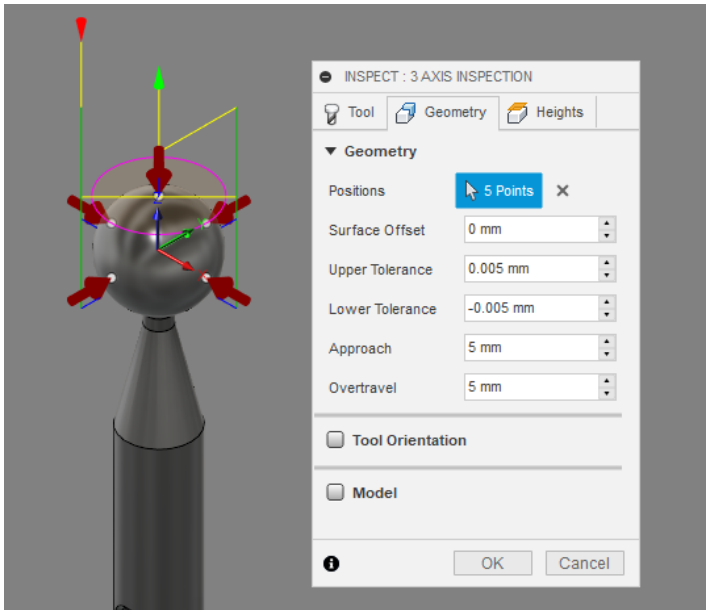
Inspect Surface Operation

If you wish to use the Inspect Surface operations, you will need a post processor that will allow you to output and run these inspection paths on your machine. You can either use one of the generic Inspection post processors available on the Fusion/HSM Post Library, or modify your current milling post which is already set up for your machine to add in the inspection functionality.

The Inspection post processors will have the *inspection* or *inspect surface* suffix appended to the name of the post processor. These are the only post processors that support Inspect Surface operations. You will need to use one of these generic posts as a source for adding the inspection code to your post processor.

8.3.1 Inspect Surface Operations

Inspect Surface operations differ from the other probing operations, in that you will select points on the face of the part to inspect instead of individual features of the part.



Surface Inspect Interface

The Surface Inspect operations are considered a cycle in the post processor and therefore call the *onCyclePoint* function, though they are expanded in the *inspectionCycleInspect* function. The standard *cycleType* variable to define the cycle type is not set for Surface Inspect operations, but rather the *isInspectionOperation* function is used to determine if it is a Surface Inspection cycle. This is further explained in the *Adding the Supporting Surface Inspect Logic* section. Unlike other cycles that pass a single point to the *onCyclePoint* function, the Surface Inspect cycle will contain the following 3 points per cycle location, with each location generating a separate and subsequent call to *onCyclePoint*.

Location	How to determine	Description
First	isFirstCyclePoint()	Safe move to approach inspection location
Second	(default)	Inspection move
Third	isLastCyclePoint()	Retract move

Three Points per Inspection Location

The parameters defined in the Inspect Surface operation are passed to the cycle functions using either the *cycle* object or through section parameters (*getParameter*). These parameters are not described here, since they are handled in the core Surface Inspect functions that are copied from an existing inspection post processor.

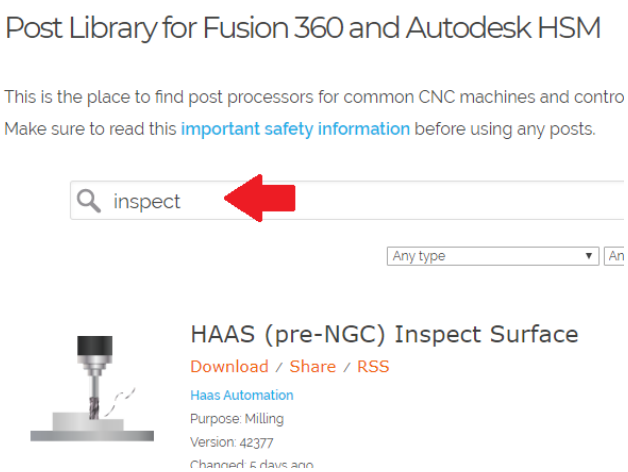
8.3.2 Adding the Core Inspect Surface Logic

Adding Surface Inspect support requires the main logic to be copied directly from a post processor that already supports inspection, and logic added to the main sections of the post processor. You should first open a post processor that contains support for inspection before starting to add Inspect Surface support to your post processor, since the logic and most of the code will remain the same. As of this writing, the following post processors have support for inspection, notice that all of them are named with the *inspect surface* or *inspection* suffix.

Post Library Name	Filename
Fanuc Inspection	fanuc inspection.cps
HAAS (pre-NGC) Inspect Surface	haas inspect surface.cps
HAAS – Next Generation Control Inspect Surface	haas next generation inspect surface.cps
Heidenhain Inspection	heidenhain inspection.cps
Siemens SINUMERIK 840D Inspection	siemens 840D inspection.cps

Post Processors that Support Surface Inspect Operations

You can also search the online [Post Library for Fusion 360 and Autodesk HSM] to see if any other post processors have been added with inspection capabilities.



Search for Posts that Support Surface Inspect Operations

The main code for Inspect Surface logic is located at the end of the post processor. You will need to copy from the definition of description located after the *onClose* or *onTerminate* function to the end of the file and add this code to the end of your post processor.

```
description = "HAAS - Next Generation Control Inspect Surface";
minimumRevision = 42251;
longDescription = "Generic post for the HAAS Next Generation control with inspect surface capabilities.";
```

Copy From this Code to the End of the File for Core Surface Inspect Logic

8.3.3 Adding the Supporting Inspect Surface Logic

There are a number of locations that contain support logic for Inspect Surface operations in the post processor. You can refer to any of the generic post processors that support Inspect Surface operations for an example on where this code is implemented.

Add the following code at the end of the *onOpen* function.

```
  // Probing Surface Inspection
  if (typeof inspectionWriteVariables == "function") {
```

```
  inspectionWriteVariables();
}
```

Add to the End of the onOpen Function

For multi-axis machines it is important that an actual machine configuration is defined and is not reliant on 3+2 plane codes and/or IJK output. Please refer to the *Multi-Axis Post Processors* section for a description on implementing multi-axis support to your post processor.

If your post processor does not have the *isInspectionOperation* function defined, then add it after the *isProbeOperation* function.

```
function isInspectionOperation(section) {
  return section.hasParameter("operation-strategy") && (section.getParameter("operation-strategy")
== "inspectSurface");
}
```

Add isInspectionOperation

The following code needs to be added to the *onSection* function.

```
  if (!isInspectionOperation(currentSection) && !isProbeOperation() &&
    (insertToolCall ||
     forceSpindleSpeed ||
     isFirstSection() ||
     (rpmFormat.areDifferent(spindleSpeed, sOutput.getCurrent())) ||
     (tool.clockwise != getPreviousSection().getTool().clockwise))) {
  forceSpindleSpeed = false;
```

Don't Output Spindle Speed with a Probe Tool

At the end of the *onSection* function, but before any subprograms are defined, add the following code.

```
  if (isInspectionOperation(currentSection) && (typeof inspectionProcessSectionStart == "function"))
{
    inspectionProcessSectionStart();
  }
```

Initialize the Surface Inspect Operation

At the top of the *onCyclePoint* function add in the following code.

```
  if (isInspectionOperation(currentSection) && (typeof inspectionCycleInspect == "function")) {
    inspectionCycleInspect(cycle, x, y, z);
    return;
  }
```

Call the Controlling Surface Inspect Function

At the start of the *onSectionEnd* function add the following code.

```
  if (typeof inspectionProcessSectionEnd == "function") {
    inspectionProcessSectionEnd();
```

```
}
```

Finalize the Surface Inspect Operation

At the end of the *onClose* function, but before any subprogram statements, add the following code.

```
if (typeof inspectionProgramEnd == "function") {
  inspectionProgramEnd();
}
```

Finalize the Surface Inspect Program

Index

Index

Index

Index

Index

W