Workflow Templates

Workflow Template Reference

Revised August 24, 2021

Table of contents

1 Workflow wiZard Template Overview 4

1.1 Introduction 4

1.2 Running Workflow Builder 5

1.2.1 Sample JCL 5

1.2.2 MASTER TEMPLATE 6

1.2.3 COMMAND LINE SYNTAX 6

2 Template Syntax 7

2.1 Sample Template 7

2.2 Workflow Tags 8

2.2.1 NAME 8

2.2.2 TITLE 8

2.2.3 DESCRIPTION 8

2.2.4 --DYNAMIC-- 9

2.2.5 SKILLS 9

2.2.6 TYPE 9

2.2.7 MAXRC 10

2.2.8 AUTO 10

2.2.9 SAVE-AS 10

2.2.10 JOBNAME 10

2.2.11 INCLUDE\_IF 10

2.2.12 PREREQ 11

2.2.13 CONTENT 11

2.2.13.1 #INCLUDE support 11

2.2.14 INSTRUCTIONS 12

2.2.15 GROUP 12

2.2.16 COMMENT 12

2.2.17 OPTIONAL 13

2.2.18 SUCCESS 13

2.2.19 SUSPEND 13

2.3 Variables 13

2.3.1 Property Variables 13

2.3.2 Overriding Property Values 14

2.3.3 Workflow Builder Properties 14

2.3.4 Workflow-Defined Variables 14

2.3.5 Workflow Builder Supplied Variables 14

2.4 Conditional Expressions 15

2.5 Internal Macros 15

2.5.1 #APPEND() 15

2.5.2 #COMPUTE(expression) 16

2.5.3 #ENDSWITH(value,str1,…) 16

2.5.4 #FMT\_VERSION(VerStr,Fmt) 17

2.5.5 #IF() / #ELSE() 17

2.5.6 #IF\_EMPTY 17

2.5.7 #ITERATE(*varName=*StartValue,EndValue) 18

2.5.8 #FOR\_EACH(set,condition) 18

2.5.9 #GEN\_ALLOC\_JCL(*tag*) 20

2.5.10 #GEN\_DELETE\_JCL(*tag*) 20

2.5.11 #GEN\_COMPRESS\_STEP(*tag*) 21

2.5.12 #GEN\_COPY\_STEPS(*tag*) 21

2.5.13 #GEN\_CLIST\_DD and #GEN\_DD(libname,Opts,LLQ1,LLQ2,LLQ3…) 21

2.5.14 #MIN/MAX(P1, P2,…,Px) 22

2.5.15 #PAD(STRING, LENGTH) 22

2.5.16 #REPLACE(src,find,{repl}) 23

2.5.17 #REPLACE\_TOK(src,delimiter,tokenNumber,repl) 23

2.5.18 #ROWCNT(varName:*alias*,*conditionExpression*) 23

2.5.19 #RSYM(x) 23

2.5.20 #TAB(x,{y}) 23

2.5.21 #TARGET\_DSN(OrigLLQ) 24

2.5.22 #TARGET\_FIELD(OrigLLQ, fieldName) 24

2.5.23 #TOKEN(String,Delimiter,Number) 24

2.6 Template Types 25

2.6.1 TYPE=INSTRUCTIONS 25

2.6.2 TYPE=JCL 25

2.6.3 TYPE=TSO-REXX-JCL 25

2.6.4 TYPE=SAVE 25

2.6.5 TYPE=DATASET\_LIST 26

2.6.6 TYPE=DATASET\_POPULATE 27

2.6.7 TYPE=DATASET\_ALLOC, TYPE=DATASET\_COPY, and TYPE=DATASET\_DELETE 28

2.6.8 TYPE=WFMACRO 28

2.7 Velocity Scripting 29

2.8 Shared Properties 29

2.9 Standards 30

2.9.1 NAME 30

2.9.2 SAVE-AS 30

2.9.3 SHARED PROPERTY MEMERS 30

# Workflow wiZard Template Overview

## Introduction

z/OSMF provides the capability of workflows. Workflows guide a user through a set of ordered steps to complete a task. Workflows are very useful but can be difficult to manage a workflow that may contain a large number of steps with input from multiple people as it is ultimately a single XML document. There are additional complications when a workflow needs to be built from a library of steps, such as a software vendor that might need to select steps to deliver based upon a combination of products being delivered. For these reasons, a tool – referred to as the Workflow Builder - was created to provide a flexible way of building workflows.

The workflow builder is the heart of the workflow wiZard project. It reads a library of templates along with a set of properties, determines which steps are necessary based upon rules that use property values, determines a suitable order to satisfy the workflow engine requirements, inserts variable definitions when required, and outputs workflow XML.

This document describes the basic operation of the workflow builder and documents template syntax. Unless otherwise noted, all input is expected to be in EBCDIC. There is no current support for ASCII or other codepages.

## Running Workflow Builder

The workflow builder is currently designed to be submitted as a batch job. The batch job takes parameters describing datasets that contain a list of property files and template PDS/PDSE libraries to read as well as a location to write the generated workflow XML.

### Sample JCL

Here is sample JCL to execute the workflow builder functionality.

//BLDWF JOB (5226),'XXXX',MSGLEVEL=(1,1),NOTIFY=&SYSUID,

// MSGCLASS=X,CLASS=A

//\*

//\* WORKFLOW BUILDER

//\*

//BLDWF EXEC PGM=BPXBATCH

//STDOUT DD SYSOUT=\*

//STDERR DD SYSOUT=\*

//STDPARM DD \*

PGM

/usr/local/bin/wfwizbld

TEMPLATE=MY.TEMPLATE.LIB1

TEMPLATE=MY.TEMPLATE.LIB2

PROPERTY=MY.PROPERTY(EZDB2)

WORKFLOW=MY.WORKFLOW(CFGDB2)

/\*

### MASTER TEMPLATE

A special member, named MASTER, is required. This contains a workflow XML “skeleton” and contains a tag to indicate where the workflow steps should be inserted. A sample is provided that looks like this:

<?xml version="1.0" encoding="IBM-1047" standalone="yes"?>

<workflow>

<autoTakeOwnership>true</autoTakeOwnership>

<workflowInfo>

<workflowID scope="none">${WF\_ID}</workflowID>

<workflowDescription>Configure Environment ${ENV\_NAME}

</workflowDescription>

<workflowVersion>1.00</workflowVersion>

<vendor>BMC Software, Inc.</vendor>

<General/>

</workflowInfo>

<variable name="izud-datasets">

<label>Zones</label>

<abstract>System</abstract>

<description>System</description>

<category>SWI Variables</category>

<array/>

</variable>

<variable name="BMCENVS" scope="global">

<label>Environments</label>

<abstract>System</abstract>

<description>System</description>

<category>BMC Variables</category>

<array/>

</variable>

--STEPS--

</workflow>

### COMMAND LINE SYNTAX

TEMPLATE *dsn*

Specifics a DSN name where templates are located. You may specific multiple TEMPLATE DSNs. If a template member name matches a member name read by a previous TEMPLATE dataset then that template member is skipped to match behavior of dataset concatenation. Ideally, the dataset should be fixed format with an LRECL of sufficient size to hold the longest line of input. An LRECL of 132 suffices for most purposes.

TRACE

Enables trace. Trace goes to STDERR at this time.

PROPERTY or PROPxxxx  
Determines a property member that should be read. If the PROPxxxx name is other than PROPERTY then the variables will have xxxx prefixed on the variable names as a “scope” of the format “xxxx-VariableName”. Most people will just use PROPERTY and will not use this “scope” functionality. One use of the scope feature is to distinguish sets of properties that might have variable names in common. For example, a vendor generating a workflow to configure products may have some values that are related to some set of common infrastructure along with product-level variables. This functionality will be documented better in the future.

WORKFLOW *dsn(member)*  
Where the primary workflow XML should be written. This is used when only a single workflow is being built. This is the most common use of the workflow builder.

WORKFLOW\_DSN *dsn*

In situations where multiple workflows may be written, this is a dataset location where the individual workflow XML members should be written. This use is an advanced feature that currently applies to CONFIGURATION and DEPLOYMENT workflows as specified by an optional TARGET keyword in the templates. *This feature is subject to change* and is made with the idea that software vendors may want to generate workflows to be presented during deployment versus workflows that might be executed at the vendor to performs some level of pre-configuration operations when generating software instances to deliver to a customer.

# Template Syntax

Most templates define a workflow step. However, there are other templates that are intended to be used as “include” files, to define Velocity macros, to define an abstraction layer for datasets that could have a different name, define rules to allocate and populate datasets by copying content from other datasets (such as copying members from SMP/E target datasets to libraries intended to be used for runtime), etc.

The syntax is intended to be simple and, other than Velocity macros or other intentional use of Velocity scripts, requires little to no understanding of Velocity.

## Sample Template

A sample template looks like this:

NAME=MYSTEP  Step name

TITLE=First step  Title of step, as shown in the workflow user interface

DESCRIPTION=This is my first step  Description of the step, as shown in the workflow user interface

TYPE=JCL  This step submits JCL

JOBNAME=MYSTEP  The jobname to use if the user chooses to save JCL

SAVE-AS=${DVOLIB}($JOBNAME)  Where to save the JCL, should the user want to save the JCL ${varname} values come from the PROPERTY content.

--INSTRUCTIONS--  Marks the start of instructions to be shown to the user. This can be multiple lines.

This is a sample step. Submit this job to do something.

--CONTENT--  Step content – such as the JCL for a JCL step  
//MYEXEC EXEC PGM=MYUTIL

//MYOUT DD SYSOUT=\*

A template can contain variables in the format ${variableName}. The workflow building tool will replace values passed in the PROPERTY members immediately rather than having the Velocity engine perform insertion. Variable references not passed in a PROPERTY member are left as-is.

## Workflow Tags

The following tags are available in a template. Unless otherwise noted, these tags are in the form of NAME=Value, with any characters to the right of the = being considered as part of the value. In other words, no quotation marks are required for values containing spaces.

Variables and macros are allowed in most fields. These will be discussed in another section.

### NAME

This optional tag sets the step name. This name must comply with z/OSMF standards. This step name is not shown to the user but may be referenced by other tags such as DEPENDS to define dependent steps. By default, the NAME is assumed to be the member name of the template. However, it is good practice to give the step a name other than the member name (again, complying with z/OSMF workflow specifics for a step name).

### TITLE

This tag sets the title of the step. This is shown to the user and is how the user identifies steps. The title should not exceed 100 characters.

### DESCRIPTION

A short description of the step. This is shown to the user to give them an idea of what the step does. Description may be multiple lines be using the syntax –DESCRIPTION—(That is two “minus” signs together) and ends with “—END DESCRIPTION—” (again, -- is two minus signs together).

### --DYNAMIC--

The --DYNAMIC-- indicates the step should generate 0 or more steps using #GENERATE\_STEP. Since #GENERATE\_STEP is only used in the context of a DYNAMIC block it is not documented under the regular macro call section.

A typical use case would be to iterate a “table”-style variable and generate steps using the “columns” of each “row”. For example:

--DYNAMIC--   
#FOR\_EACH(DSN\_LIST:D,#ENDSWITH(${D}[DSNID],"SAMP"))   
 #GENERATE\_STEP(MY\_STEP\_${DSNID}[LPAR])   
#END\_FOR\_EACH   
--INSTRUCTIONS--  
Instructions for ${D}[DSNID]   
--CONTENT--  
Step content for ${D}[DSNID].

This iterates through a DSN\_LIST “table” provided in a PROPERTY member and locates all DSN IDs ending in SAMP. It generates a new step called MY\_STEP\_lpar, where lpar is the value of the LPAR “column” for the rows of the table. #GENERATE\_STEP only has a single parameter: the step name for the new step. The step name must be unique within the workflow. See #GENERATE\_STEP for more information on generating steps dynamically..

### SKILLS

This defines the skillset needed. This helps the user know who should perform the step, such as DB2 DBA, Systems Programmer, etc.

### TYPE

This tag defines the type of step. Supported types are:

* INSTRUCTIONS: The step only contains instructions.
* JCL: The step contains JCL that can be submitted.
* SAVE: z/OSMF allows content to be saved as-is. This can be used for creating JCL include files or any other regular text file. Content is not interpreted as JCL.
* TSO-REXX-JCL: z/OSMF will submit the content as REXX script.
* TSO-UNIX-Shell: UNIX shell command. Must include SUCCESS tag as well for the workflow to be valid.
* GROUP: This step represents a GROUP. A group is a step that contains multiple steps underneath it. A group can be referenced with the GROUP tag of other templates to indicate they should be placed within the group. A GROUP step doesn’t perform any task other than being a container for sub-steps.

NOTE: Single-member GROUP steps may have the “GROUP” level removed by the workflow builder whereas GROUP steps with multiple steps will be emitted by the workflow builder.

* INCLUDE: This represents content that can be included from other templates. Only the –CONTENT—section of an INCLUDE template is meaningful. See the #INCLUDE support section for information on how to use an INCLUDE template.
* MACRO: This step represents a Velocity macro. The content is Velocity macro code. Any macros found will be added to the steps that appear to reference the macros. Velocity macro usage is likely going to be minimal, if used at all, due to the difficulty in diagnosing Velocity script errors in z/OSMF.
* WFMACRO: This step represents a workflow builder macro. Unlike a Velocity macro, a workflow macro is “executed” by the workflow builder.

### MAXRC

This is the maximum return code from a job that will allow the step to be considered successful. The default is 0.

### AUTO

Indicates if the step can be automated. An automated step can be submitted by z/OSMF without user intervention. A user can tell z/OSMF to submit all consecutive AUTO steps. z/OSMF will perform the steps with no user intervention until it hits a non-automated step or a step fails. Values can be TRUE or FALSE. If AUTO is not specified then TRUE is assumed for all steps.

### SAVE-AS

Specifies where to save the content of this step or where to save the JCL produced by this step. This allows the customer to save a file or save a job.

### JOBNAME

This tag is used by the workflow builder to tell z/OSMF what the jobname should be for a given step. The JOBNAME can also be used as a variable, such as for a member name for the SAVE-AS tag or anywhere else within the template it might be needed.

### INCLUDE\_IF

This contains a comma-separated list of conditions that must be met for the step to be considered as part of the workflow. Each INCLUDE\_IF line may have multiple conditions specified. All conditions on a line must evaluate to TRUE for the line to be TRUE. Multiple INCLUDE\_IF lines are allowed – only one line must evaluate to TRUE for the member to be included.

Example:

INCLUDE\_IF=${LGC\_NEEDED} = “Y” AND ${DBC\_SSID} != “”  
 INCLUDE\_IF=${DBC\_NEEDED} = “Y” AND ${DBC\_SSID} !=””

See the section on [conditional expressions](#_Conditional_Expressions) for more information on expression syntax.

### PREREQ

This lists one or more step that are a perquisite for the step to be in a “Ready” state in z/OSMF. By z/OSMF rules, this means that the step will appear after its prerequisite steps. If a prerequisite step is to be excluded based upon INCLUDE\_IF rules, or if the prerequisite step doesn’t exist in the template library, then it is removed as a prerequisite step. For example, if a step has a PREREQ with X, Y, and Z but step Y is excluded by INCLUDE\_IF rules then the step is adjusted to only PREREQ upon X, Z, and any steps that step Z had included as a prerequisite but it will not include step Z itself since step Z is not a part of the generated workflow. This allows for steps that need to be performed after some other steps, but only if those steps are required or exist. The remaining prerequisite steps are set in the workflow XML for z/OSMF to enforce.

Once the workflow builder has found an appropriate place in the order of the workflow for a step, it then attempts to sort steps that are together in alphabetical order by NAME. However, PREREQ conditions have top priority when determining step order.

### CONTENT

This tag is defined as --CONTENT--. Any content between –CONTENT--and the end of the template OR until another line starting with “--“is located is considered part of the step content. This is the content for JCL, SAVE-AS, and REXX scripts.

### #INCLUDE support

Content may have INCLUDE tags to include other members from the template concatenation.

--CONTENT--

#INCLUDE=DB2STEP1

#INCLUDE=DB2STEP2:COND=${REORG\_NEEDED}=”Y”

#INCLUDE=DB2STEP2:COND=${COPYP\_NEEDED}=”Y”

#INCLUDE=DB2STEP3:COND=${REORG\_NEEDED}=”Y”,${DB2GRP}!=””

#INCLUDE=DB2STEP4:COND=${COPYP\_NEEDED}=”Y”

#INCLUDE=DB2STEP4:INCAGAIN:COND=${STATS\_NEEDED}=”Y”

In the above example, DB2STEP1’s content will always be included. It will include DB2STEP2 if REORG\_NEEDED is TRUE or if COPYP\_NEEDED is TRUE. Note that an include member is only included once *unless* INCAGAIN is specified. This means DB2STEP2 will not be included twice even if REORG\_NEEDED and COPYP\_NEEDED are both Y. DB2STEP3 is included if REORG\_NEEDED is TRUE AND DB2GRP is not an empty string. DB2STEP4 is included only if COPY\_NEEDED is TRUE and will be included a second time if $STATS\_NEEDED is TRUE.

### INSTRUCTIONS

This tag is defined as --INSTRUCTIONS--. Any content between –INSTRUCTIONS--and the end of the template OR until another line starting with “--“is located is considered part of the instructions. z/OSMF supports HTML tags for formatting, so feel free to make it look nice. If HTML is not detected in the instructions then the instructions are wrapped with appropriate tags that let the browser see the text is fixed-width data.

The INSTURCTIONS section is treated in a similar manner as the CONTENT section: All macros and such function the same as they do in CONTENT.

Example:

--INSTRUCTIONS—  
Do something useful with this  
workflow step.  
--END INSTRUCTIONS--

### GROUP

Workflows support nested steps. The GROUP tag indicates the step is to be part of a special GROUP step type. The step will appear as a child of this group. The GROUP name must match a template of TYPE=GROUP or the step will be left out of the workflow XML (with a warning produced).

A group may be a subgroup of another group as well by setting a the subgroup’s GROUP= name to the intended parent group’s NAME.

If a group ends up with only a single member then the GROUP node is left out of the workflow to make the workflow steps more compact. Groups with multiple steps will always be emitted.

### COMMENT

This tag is defined as –COMMENT--. Any content inside the –COMMENT—block will be ignored. (Note: Like Content and Instruction blocks, any line starting with “--” ends the block.

Example:

--COMMENT—  
Anything in here will  
be ignored.  
--END COMMENT—

### OPTIONAL

This tag, if TRUE, indicates the step is optional. z/OSMF shows the step as optional to the user and this step will NOT be added as a prerequisite for any other step (implicitly, via group prerequisite, or explicitly). Prerequisites will be used only to properly place the step into the right place within the workflow.

### SUCCESS

This tag contains a string that the z/OSMF workflow engine will check in the output of a UNIX shell script to determine if the script executes successfully. See z/OSMF’s workflow XML reference for information on this pattern. It can include patterns.

### SUSPEND

This tag, if TRUE, indicates to z/OSMF that it is to suspend automation when it reaches this step.

## Variables

There are 3 types of variables: Workflow-defined variables, which are defined in the master workflow template itself and that can have values passed from the properties file, properties-only variables that are not defined to the workflow itself but do have values in the properties file, and workflow builder variables.

All variables are recognized by the workflow builder as ${var\_name}. Example:  
 <rtcsreg xcfgroup="${DBC\_GROUP}#">

### Property Variables

Property variables are replaced with their value from the properties file when the workflow is built. The majority, if not all, of variables will fall into this category. These variables are replaced by the workflow builder during build time if they are in “simple” form, such as ${VARNAME} or $!{VARNAME}. ${VARNAME} is substituted with the variable’s value. $!{VARNAME} is substituted with the variable’s value only if the variable is set, otherwise it is replaced with an empty string. More complex use of these variables will need to be of the form ${instance-VARNAME} so the Velocity engine will be able to do more complex operations.

The workflow builder also supports variable scope. This is most useful for using shared properties (work in progress) where variables from shared environments will have a scope placed up front, such as ${INFR-VARNAME}. Replacement will first look for the scope-specific variable called INFR-VARNAME. If it cannot find INFR-VARNAME then it tries to locate VARNAME. This will allow templates to prioritize grabbing properties from a shared infrastructure property file over one defined in the product property file.

### Overriding Property Values

In order to provide a convenient way to override values set by the config tool, the workflow builder will look in the submitter’s UNIX home directory for a file called cvibldwf.properties. If this file exists, the workflow builder will read these properties and they will override values from the regular property input. This is useful for development to enable workflow tracing or perhaps other internal purposes.

### Workflow Builder Properties

The following are property values that, if set, impact the workflow builder.

* CVIBLDWF\_GEN\_JOBNAME – If set to anything other than STANDARD, the workflow builder will not write the line that tells z/OSMF to use the JOBNAME for JCL steps. This means the name will be left up to z/OSMF’s default naming conventions, which the user can adjust but will be the same for all workflow steps. This feature can be useful if multiple people are testing installs on the same system so that we don’t end up with the same jobs trying to execute at the same time, causing delays in development as they compete on the queue.
* CVIBLDWF\_TRACE – If set to ON, the workflow builder will enable trace and write it to NGITRACE (which will be automatically allocated).

### Workflow-Defined Variables

Workflow-defined variables are left in the XML as variables, replaced with “instance-“ prepended to make sure the scope is set appropriately for the workflow. The z/OSMF Velocity engine replaces the values in the workflow steps at runtime. Because the workflow engine automatically adds all property file variables into the workflow as variables, the use of workflow-defined variables is not necessary.

The difference between a workflow variable and a property variable is that a workflow variable may be setup to prompt the user for its value. Because of this, the value is not substituted by the workflow engine but leaves it up to the Velocity engine during workflow step execution. You may use workflow variables as ${VARNAME} or ${instance-VARNAME}. The workflow builder will prepend “instance-“ to any workflow variable names that do not already have the prefix attached.

### Workflow Builder Supplied Variables

Workflow builder variables are defined by the workflow builder tool itself. The following variables are set at startup:

${BASE36\_TIME} – Current timestamp, using year/month/day of month/hour/minutes/seconds, converted to a Base 36 string (0-9, A-Z for digits). Ends up as a 7-character string.  
${CUR\_DATE} – The current date in MM/DD/YYYY format  
${CUR\_TIME} – The current time in HH:MM:SS format  
${CUR\_TIMESTAMP} - ${CUR\_DATE} at ${CUR\_TIME}  
${CUR\_MONTH} – Current month  
${CUR\_MDAY} – Current day of month  
${CUR\_YEAR} – Current year  
${CUR\_HOUR} – Current hour of time  
${CUR\_MINUTE} – Current minute of time  
${CUR\_SECOND} – Current seconds of time  
${UNIQUE\_NUMER} – A unique 64-bit number. A sample use is to define a temporary location in /tmp.

## Conditional Expressions

The workflow builder has an expression evaluation engine to handle simple and complex expressions.

Variable / Constant Designations

* ${VARIABLE\_NAME} - VARIABLE\_NAME value will be used or an empty string if not set
* “CONST\_VALUE” - CONST\_VALUE will be used
* VAR\_OR\_CONST - If VAR\_OR\_CONST matches a variable name then the variable’s value used. If not, then VAR\_OR\_CONST is assumed to be a constant.

Operators

The following operators are supported:

* AND, &&
* OR, ||
* EQ, == (NOTE: A single = is used by the expression engine for assignment, not equality checking)
* NEQ, !=, ^=
* GT, >
* LT, <
* GE, >=
* LE, <=
* ( and )
* Math operators +, -, \*, ^ and / are supported as well as bitwise operators | and &.

## Internal Macros

Velocity supports macros. However, writing macros can be complex so the workflow builder supplies some built-in macros to perform some common operations.

### #APPEND()

#APPEND() at the start of a line will place the data at the end of the preceding line. This may be used in combination with other macros, such as #TAB, to align the data being appended.

### #COMPUTE(expression)

Evaluates an expression and writes out the result. Example:

BOB IS #COMPUTE( 200 \* ${SOMEPCT} / 100 )% NORMAL

Assume ${SOMEPCT} is a variable in the property file with a value of 35, this results in:

BOB IS 70% NORMAL

### #ENDSWITH(value,str1,…)

This is likely to be useful only during condition evaluation. Returns *true* if the value passed ends with any of the strings passed. For example, #ENDSWITH(${DSN\_LIST}[DSNID],LINK,LIB) as a #FOR\_EACH condition returns true if the DSNID ends with LINK or LIB.

### #FMT\_VERSION(VerStr,Fmt)

This macro will format VerStr according to Fmt. VerStr is a string or variable with value VV.RR.MM. Fmt indicates how to format the output. Fmt values can be:

VV – 2-digit version, padded on the left with zeroes if needed.  
 RR – 2-digit release, padded on the left with zeroes if needed.  
 MM – 2-digit mod, padded on the left with zeroes if needed.  
 V – version, unpadded.  
 R – release, unpadded.  
 M – mod, unpadded.

Example: #FMT\_VERSION(9.04.3,VVRMM) results in 09403.

### #IF() / #ELSE()

IF / ELSE logic may be implemented using syntax as follows:

#IF(expressionA)  
Lines to write if expression A is true  
#ELSE(expressionB)  
Lines to write if expression A is false and expression B is true.  
#ELSE()  
Lines to write if expression A and B are both false.  
#END\_IF

If multiple lines are to be written then this method is preferred over #WRITE\_IF. #WRITE\_IF was intended to conditionally handle a single line of data and grew to handle more complexity. #IF/#ELSE is easier to read that #WRITE\_IF when there are many conditions or many lines to be written.

Expressions may span multiple lines. Processing of the expression stops when a closing parenthesis is found. Expressions may contain embedded parentheses: #IF ( (A=B) || (A=C) ) is a valid statement.

### #IF\_EMPTY

This call checks content between #IF\_EMPTY and #EMPTY\_CONTENT. If the checked content is empty then the #EMPTY\_CONTENT block up to #END\_IF\_EMPTY is written. This is useful for places where a template may have an extensive series of conditions that may or may not result in empty output and, if so, another section of output should be performed.

Example:

#IF\_EMPTY  
#FOR\_EACH(MyTable,${MyTable}[SYSPLEX}=PLEXA)  
Data is ${MyTable}[SSID]  
#END\_FOR\_EACH  
#EMPTY\_CONTENT  
No data was found for this PLEXA.  
I can place other conditions within this block or keep it simple like this example.  
#END\_IF\_EMPTY

### #ITERATE(*varName=*StartValue,EndValue)

This allows for basic iteration over a range of numbers from StartValue to EndValue, inclusive. *varName=* is optional but, if present, overrides the default ${ITER} variable name to use when iterating. This variable will contain the current value within the iteration range. The start and ending values may either be descending or ascending.  
  
Example:

#ITERATE(D=5,20)  
This is ascending iteration ${D}  
#END\_ITERATE  
  
#ITERATE(D=20,5)  
This is descending iteration ${D}  
#END\_ITERATE  
  
Parameters may contain other variables and expressions. For example, you might want to use #MIN or MAX to define a range or possibly #ROWCNT.

### #FOR\_EACH(set,condition)

This call iterates a variable that contains a set of variables. Each “row” of variables is iterated between #FOR\_EACH and #END\_FOR\_EACH.

A variable containing a set can be setup in the properties file like this:

MYSET=[{"property1":"tt1",”member”:”MEM1”,"dsName":"TEST.DSNAME.TT1"},\  
      {"property1":"tt2",”member”:”MEM2”,"dsName":"TEST.DSNAME.TT2"},\  
      {"property1":"tt3",”member”:”MEM3”,"dsName":"TEST.DSNAME.TT3"}]

The set may be iterated like this:

#FOR\_EACH(${MYSET})  
${MYSET}[member] has a DSN of ${MYSET}[dsName]  
#END\_FOR\_EACH

‘set’ may contain an alias to use, which allows for nested FOR\_EACH calls on the same list. #FOR\_EACH(MYSET:OUTER, …) would use the variable name ‘OUTER’ to access the content, such as ${CONTENT}[member] instead of ${MYSET}[member].

The content to be iterated may contain calls to other macros. Nested #FOR\_EACH operations are also supported.

You may make a call to a workflow macro to iterate the set. Assume you have the following WFMACRO called MYMAC:

#PARMS(ROWVAR)  
#FOR\_EACH(${ROWVAR})  
MACRO CALL FOR ROW ${ROWVAR}[rowcnt] - ${ROWVAR}[member] ${ROWVAR}[dsName]  
#END\_FOR\_EACH

You could call this macro like this:

#MYMAC(${MYSET})

The macro would iterate the set and produce:

MACRO CALL FOR ROW 1 MEM1 TEST.DSNAME.TT1   
MACRO CALL FOR ROW 2 MEM2 TEST.DSNAME.TT2  
MACRO CALL FOR ROW 3 MEM3 TEST.DSNAME.TT3

Note that ‘rowcnt’ is an automatically-generated variable for each row indicating the row number within the set that is being processed.

Optionally, a condition can be passed that will be evaluated to iterate only upon entries that cause the condition to be true.

#FOR\_EACH(${DSN\_LIST},#ENDSWITH(${DSN\_LIST}[DSNID],LINK,LIB))

This would iterate all datasets whose DSNID ends with LINK or LIB.

A special macro called ‘#UNIQUE’ is available for use in FOR\_EACH conditional expressions that returns true if the iteration is seeing the unique set of parameters for the first time. For example, #UNIQUE(${DB2TABLE}[SYSPLEX],${DB2TABLE}[LPAR]) on a list called DB2TABLE would iterate once per unique SYSPLEX/LPAR pairs.

Here is a complex example using nested FOR\_EACH:

The PROPERTY DD contains:  
DB2TABLE=\                                                                     
[\                                                                             
{"SYSPLEX":"PLEXA", "LPAR":"LPA1", "SSID":"BART", "DSN":"BARTDSN" },\          
{"SYSPLEX":"PLEXB", "LPAR":"LPB3", "SSID":"HOMR", "DSN":"HOMERSN" },\          
{"SYSPLEX":"PLEXB", "LPAR":"LPB3", "SSID":"MOE",  "DSN":"MOEDSN" },\           
{"SYSPLEX":"PLEXA", "LPAR":"LPA2", "SSID":"TROY", "DSN":"TROYDSN" },\          
{"SYSPLEX":"PLEXA", "LPAR":"LPA1", "SSID":"LISA", "DSN":"LISADSN" },\          
{"SYSPLEX":"PLEXC", "LPAR":"LPC5", "SSID":"NED",  "DSN":"FLANDERS" }\          
]

A template has a #FOR\_EACH like this:

#FOR\_EACH(DB2TABLE:OUTER,#UNIQUE(${OUTER}[SYSPLEX],${OUTER}[LPAR]))      
All DB2 subsystems on ${OUTER}[SYSPLEX] ${OUTER}[LPAR]                   
#FOR\_EACH(DB2TABLE:INNER,${OUTER}[SYSPLEX]==${INNER}[SYSPLEX] &&         
                         ${OUTER}[LPAR]==${INNER}[LPAR])                 
  DB2 ${INNER}[SSID] DSN ${INNER}[DSN]                                   
#END\_FOR\_EACH                                                            
#END\_FOR\_EACH

All DB2 subsystems on PLEXA LPA1        
  DB2 BART DSN BARTDSN                  
  DB2 LISA DSN LISADSN                  
All DB2 subsystems on PLEXB LPB3        
  DB2 HOMR DSN HOMERSN                  
  DB2 MOE DSN MOEDSN                    
All DB2 subsystems on PLEXA LPA2        
  DB2 TROY DSN TROYDSN                  
All DB2 subsystems on PLEXC LPC5        
  DB2 NED DSN FLANDERS

### #GEN\_ALLOC\_JCL(*tag*)

This call, with optional tag, runs the list of datasets that have been defined using DATASET\_LIST templates and places the allocation JCL defined for the dataset into the template. If not specified, the tag defaults to the empty string. Only datasets matching the tag are considered. Only datasets that have been successfully reference by a DATASET\_POPULATE entry will be considered.

This feature might be specific to BMC Software but others might find it useful.

### #GEN\_DELETE\_JCL(*tag*)

This call, with optional tag, runs the list of datasets that have been defined using DATASET\_LIST templates and generates JCL to delete the datasets. If not specified, the tag defaults to the empty string. Only datasets matching the tag are considered. Only datasets that have been successfully reference by a DATASET\_POPULATE entry will be considered.

This feature might be specific to BMC Software but others might find it useful.

### #GEN\_COMPRESS\_STEP(*tag*)

This call, with optional tag, runs the list of datasets that have been defined using DATASET\_LIST templates and creates a compress step. If not specified, the tag defaults to the empty string. Only datasets matching the tag are considered. Only datasets that have been successfully reference by a DATASET\_POPULATE entry will be considered.

This feature might be specific to BMC Software but others might find it useful.

### #GEN\_COPY\_STEPS(*tag*)

This call, with optional tag, runs the list of datasets that have been defined using DATASET\_LIST templates and creates a copy step for each. If not specified, the tag defaults to the empty string. Only datasets matching the tag are considered. Only datasets that have been successfully reference by a DATASET\_POPULATE entry and whose populate entry found a FROM record will be considered. The JCL will use the FROM dataset from the population record(s) and copy them to the destination dataset.

This feature might be specific to BMC Software but others might find it useful.

### #GEN\_CLIST\_DD and #GEN\_DD(libname,Opts,LLQ1,LLQ2,LLQ3…)

This macro will generate a DD concatenation for libname including all datasets from a variable called DSN\_LIST specified in a property member with the IDs listed. An LLQ may also be replaced with a variable, such as ${RTE\_BMCLINK}, which will resolve to a DSN name.

Example:

#GEN\_DD(STEPLIB,!UBMCLINK,BMCLINK,${RTE\_BBLINK})

This would locate all dataset ending in BMCLINK, excluding !UBMCLINK, as well as the dataset identified by ${RTE\_BBLINK}. Any duplicate is only emitted once.

The output is a complete //STEPLIB DD DISP=SHR,DSN=xxxx concatenation.

NOTE: Older templates may use #GEN\_DD=parm1,parm2,parm3. This syntax is still supported but not recommended.

Options available are:

* REPLACE=*TextToFind*:*NewText* or REPLACE=[a:b,c:d,…,y:z]

Replaces strings in the DSN. This can be useful for changing out symbol names. This may contain a single find/replace string or multiple strings by using brackets. In the case of the bracket example, *a* will be replaced with *b*, *c* will be replaced with *d*, and *y* will be replaced with *z*.

NOTE: REPLACE remains active until the next REPLACE parameter is specified. REPLACE parameters are *not* cumulative.

Example:

#GEN\_DD(STEPLIB,REPLACE=&DVRTEHLQ:&PREFIX,LINK)

Would generate a STEPLIB using all \*LINK datasets, replacing &DVRTEHLQ with &PREFIX.

* PREFIX=*String*

Will use *string*\_LIST rather than DSN\_LIST for the dataset list name.

* SCOPE = *Scope*

This sets the default scope to use for DSN ID matching. By default the scope is empty, matching the PROPERTY DSN templates. If SCOPE is specified then only DSN IDs matching the scope will be found. For example, SCOPE=INFR would only match against DSN entries generated from processing PROPINFR properties run against the DSN templates. SCOPE=ALL will match against all DSN entries.

* ENCLOSE=*Char*

This is only used by GEN\_CLIST\_DD. It defines a character used to surround the DSN names

GEN\_CLIST\_DD Example:

#GEN\_CLIST\_DD(ENCLOSE=’,!UBMCLINK,BMCLINK,${RTE\_BBLINK})

This would locate all dataset ending in BMCLINK, excluding UBMCLINK, as well as the dataset identified by ${RTE\_BBLINK}. A single quote will surround the names. Any duplicate is only emitted once.

‘&DVHLQ..BMCLINK’ +  
 ‘INFRA.BMCLINK’

### #MIN/MAX(P1, P2,…,Px)

This macro returns the minimum or maximum value contained in the list of parameters. Values are either checked by number or by string comparison depending on the values passed.

### #PAD(STRING, LENGTH)

This macro takes the first parameter and pads it to the right with spaces to achieve LENGTH or truncate to LENGTH as needed. A variable name may be passed using ${VARNAME} for STRING. Typical use is to allow space in JCL for a variable while still maintaining alignment for a continuation character on the JCL line.

### #REPLACE(src,find,{repl})

This call will locate string “find” in the string specified by “src” and replace it with string “repl”. For example, #REPLACE(“DB2SSID”,”DB2”,”IMS”) would result in IMSSSID. *Repl* may be an empty string and, if not specified, is assumed to be an empty string. For example, #REPLACE(“DB2SSID”,”DB2”,””) and #REPLACE(“DB2SSID”,”DB2”) would both result in SSID.

The content of *src* is not modified. Variables may be used in any of the strings.

### #REPLACE\_TOK(src,delimiter,tokenNumber,repl)

This call will locate token *tokenNumber* in *src* using *delimiter* and replace it with *repl*. For example:

#REPLACE\_TOK(${jobStatement},” “,1,//TESTJOB) would replace the first token in jobStatement, as delimited by space, with //TESTJOB.

The content of *src* is not modified. Variables may be used in *src* and in *repl*.

### #ROWCNT(varName:*alias*,*conditionExpression*)

Returns the count of rows whose conditions match the expression. *Alias* allows the ability to shorthand the variable name for shorter syntax. Please see #FOR\_EACH for more information about expression matching when iterating row variables.

### #RSYM(x)

The value passed will have JCL symbols replaced, when possible. The workflow builder will look at &variable names and try to replace them with Velocity equivalents.

### #TAB(x,{y})

The current line is padded with spaces so that the characters after the #TAB() macro are at column ‘x’. If ‘x’ is less than the location of the #TAB then no padding is added unless ‘y’ is also specified, in which case #TAB will continue to space over ‘y’ spaces at a time until it gets beyond the end of the line.

#TAB can be used to align text.

Example:

#TAB(5)Col1#TAB(15)Col2

Would result in:

COL1 COL2

…where COL1 beings at column 5 and COL2 begins at column 15.

#TAB(5)Col1#TAB(5,10)Col2 would result in the same output as above. This form of #TAB may be most useful with the #APPEND() feature.

NOTE: If your line contains data that the workflow builder does not natively resolve (left up to Velocity to resolve) then the #TAB may not give the expected result. This is only an issue if you are using z/OSMF-defined variables or Velocity script.

### #TARGET\_DSN(OrigLLQ)

Locates a DSN whose bmc-orig-llq matches OrigLLQ from the TARGET set variable. The TARGET set variable should look something like this:

TARGET=[{ "DSN" : "CVI.SMPE.BMCSAMP", \   
 "BMC-ORIG-LLQ" : "BMCSAMP", \  
 “TRACKS” : “16” }, \  
… ]

This macro should be used instead of the original “GETDSN” macro provided during the workflow builder prototype phase. For example, #TARGET\_DSN(ADMCLIB) would return NGI.DB2BNDL.SI.TARG.ADMCLIB.

An alternative form of the macro is #TARGET\_DSNS. #TARGET\_DSNS will return all DSN names that match rather than just one matching DSN. For example, #TARGET\_DSNS(XXLINK) will return ALL target DSNs whose bmc-orig-llq matches XXLINK. It is possible multiple targets can have the same bmc-orig-llq.

### #TARGET\_FIELD(OrigLLQ, fieldName)

Locates an SMP/E target DSN whose bmc-orig-llq matches OrigLLQ and returns a field. The default field is DSN. Fields available depend upon the content of the TARGETS= property sent to the workflow builder. In the example below, fields are DSN, BMC-ORIG-LLQ (which is what OrigLLQ matches against), and tracks.

TARGET=[{ "DSN" : "CVI.SMPE.BMCSAMP", \   
 "BMC-ORIG-LLQ" : "BMCSAMP", \  
 “TRACKS” : “16”, \  
…

This has the same behavior as #TARGET\_DSN by default since FIELD defaults to DSN.

#TARGET\_FIELD(BMCSAMP,TRACKS) would return “16” in the above example.

### #TOKEN(String,Delimiter,Number)

Returns token (Number) from (String) using delimiters in Delimiter. The delimiter must use double quotes if the delimiter contains a comma.

Example:

#TOKEN(THIS.IS.MY.STRING,”.”,3) returns MY.  
#TOKEN(THIS,IS,MY,STRING,”,”,4) returns STRING

## Template Types

### TYPE=INSTRUCTIONS

Instruction templates are intended to give the user instructions but not have any job to submit nor file to create. These are useful for giving a step that provides documentation to the user for a series of steps that follow. For example, the first step of a group of DB2 configuration jobs may tell the user the general flow of the steps that will happen, perhaps suggesting that the customer submit the steps in an automated manner.

The CONTENT section is ignored on INSTUCTIONS templates.

### TYPE=JCL

This type of template indicates that the CONTENT section contains JCL that will be submitted. z/OSMF puts the job card information at the top of the job so the JCL should *not* contain job card information.

It is anticipated that customers would like to submit AND save their jobs. If this is the case, the template should contain a SAVE-AS statement to indicate the job should be saved. z/OSMF will perform this submit and save automatically if the step is submitted via automation. However, z/OSMF will NOT automatically save if the step is performed manually. This behavior is something we are contesting with IBM as we believe that manual and automated execution of steps should have identical results.

The INSTRUCTIONS section is display to the user when they perform the step.

### TYPE=TSO-REXX-JCL

This type of template indicates that the CONTENT section contains REXX script to be submitted as JCL. Aside from the content being REXX script, these templates otherwise behave the same as TYPE=JCL.

### TYPE=SAVE

This type of template indicates that the CONTENT section contains data to be stored into a file or member. It is likely you will want to add a SAVE-AS line for this type of member so the customer’s save location will be defaulted as desired and for automation to automatically save to an appropriate location.

### TYPE=DATASET\_LIST

DATASET\_LIST templates are special templates that provide a list of datasets that may need to be allocated or used by other workflow steps. These templates are processed before any other template and ignore PREREQ conditions as they are not presented to the customer.

Such a template will typically have only a TYPE=DATASET\_LIST entry and a CONTENT section as no other tags are necessary.

The CONTENT section contains a DSN statement followed by JCL lines that can be used to allocate the JCL. A TYPE=DATASET\_ALLOC template will pull in the JCL content to allocate the datasets.

The DSN statement syntax is:

DSN={DSNID},{DSN}  
 TAG=MyTag (optional)  
 //jcl lines  
 //used to allocation

The DSNID is an identifier that will be used by DATASET\_POPULATE templates. It is basically a variable name used to identify the dataset. The {DSN} is the {DSN} in the form that your jobs need to access the dataset. The workflow builder makes no attempt to interpret the DSN value to determine validity. The DSNID name is setup as a variable that may be used as any other property variable, with the value being set to the DSN value.

The TAG entry tags the DSN entry with a name that can then be used for internal macros #GEN\_ALLOC\_JCL, #GEN\_COPY\_STEPS and #GEN\_COMPRESS\_STEP. If passed to these “macros”, they will operate only upon the DSNs using the tag.

${DSNID} and ${DSN} may be used in the JCL section for the dataset rather than having to repeat them. See below for examples.

--CONTENT--  
DSN=BMCXML,&DVRTEHLQ..BMCXML  
//${DSNID} EXEC ALLOC,ALLODSN=${DSN},  
// DSNTYPE='PDS',  
// SPACE='CYL,(18,15,449)',  
// DCB='RECFM=VB,BLKSIZE=32760,LRECL=260'

DSN=DBXML,&DVRTEHLQ..DBXML  
TAG=DBLIBS  
//${DSNID} EXEC ALLOC,ALLODSN=${DSN},  
// DSNTYPE='PDS',  
// SPACE='CYL,(18,15,449)',  
// DCB='RECFM=VB,BLKSIZE=32760,LRECL=260'

An array called DSN\_LIST is created for all DSNs that are indicated as being used by DATASET\_POPULATE entries. The list contains the following subvariables that can be used by #FOR\_EACH.

DSN – DSN name  
 RDSN – DSN name with JCL-style symbols resolved (if available)  
 DSNID – The DSN ID for the entry (if DSN=XYZ,… then DSNID is XYZ)  
 JCL – The allocation JCL for the DSN entry

### TYPE=DATASET\_POPULATE

This template type declares usage of a given list of datasets. You may use #INCLUDE\_IF statements to limit these templates to only process under certain conditions. These steps are NOT presented to the customer but are used to indicate which DSNs are needed, a list of datasets to use to populate the dataset, and to create a variable name that indicates which DSN is being used for a given purpose. For example, the customer may have selected to have all load modules go into BMCLINK or they may want merged datasets such as DBLINK, IMLINK, etc. The POPULATE template will declare which dataset should be used and setup appropriate variables.

TYPE=DATASET\_POPULATE  
INCLUDE\_IF=SOME\_CONDITION!=FALSE  
--CONTENT--  
POPULATE=RTE\_BRDXML,FROM=#TARGET\_DSN(BRDXML)  
TO=BMCXML,COND=RUNTIME="BMCLINK"  
TO=#TARGET\_DSN(DBXML),HLQPREFIX=SMPHLQ,COND=RUNTIME="TARGETS"  
TO=DBXML,COND=RUNTIME!="BMCLINK"

This indicates that a variable called RTE\_BRDXML will be established. It will either contain the value of the DSN for DSN ID BMCXML, the SMP/E target DBXML, or DSN ID DBXML depending on whether the RUNTIME variable is set to BMCLINK or not.

If TO represents a DSN ID then the content of TARGET DSN BRDXML will be copied into the destination dataset.

If TO contains a DSN name rather than ID then a new dataset entry is generated automatically using the HLQPREFIX (which defaults to SMPHLQ if not specified) as a &HLQ. prefix in the name. For example, the SMP/E target DSN above would look something like &SMPHLQ1..DBXML. Such automated dataset entries will be bypassed for #GEN\_ALLOC\_JCL, #GEN\_DELETE\_JCL, #GEN\_COMPRESS\_STEPS, and #GEN\_COPY\_STEPS. The implication is they will not be part of allocation/copy jobs. By default these dataset entries have a DSNID of DYNDD\_xx\_popid. So the above example would generate a DSN with DSNID DYNDD\_1\_RTE\_BRDXML. #GEN\_DD statements matching DSN IDs ending in XML would locate this dataset. You may set your own suffix by specifying DYNSUFFIX= on the line. For example, DYNSUFFIX=DBXML would result in a name of DYNDD\_1\_DBXML. Note that it would be tempting to have the workflow builder use the first parameter to #TARGET\_DSN for this but remember that this is just a macro and TO could be formed through other means so #TARGET\_DSN would not always exist.

FROM= is optional and, if missing, means that the dataset will be used (and the TO variable created) but no data needs to be copied.

Other steps may reference the DSN by using the variable ${RTE\_BRDXML}, which will either have the DSN for BMCXML or DBXML.

### TYPE=DATASET\_ALLOC, TYPE=DATASET\_COPY, and TYPE=DATASET\_DELETE

These steps are special and will be provided by the NGI team. This template is treated like a TYPE=JCL step but will leverage the DATASET\_LIST and DATASET\_POPULATE information to allocate and copy data to these datasets.

### TYPE=WFMACRO

A workflow macro is like an #INCLUDE except it can take parameters. Here is sample CONTENT from a workflow macro:

NAME=SAMPMAC  
TYPE=WFMACRO  
--CONTENT--  
#PARMS(LIBNAME, VOL,STCLASS,MGTCLASS,DACLASS)  
DSN=${LIBNAME}  
#WRITE\_IF=(${VOL} != ""):VOL=SER=${VOL}  
#WRITE\_IF=ELSE(${STCLASS} != ""):STORCLAS=${STCLASS}  
#WRITE\_IF=ELSE(${MGTCLASS} != ""):MGMTCLAS=${MGTCLASS}  
#WRITE\_IF=ELSE(${DACLASS} != ""):DATACLAS=${DACLASS}  
UNIT=${DVUSUNT}  
DISP=(,CATLG)  
--END CONTENT—

To call this macro, include the following line anywhere in the –CONTENT—, --INSTRUCTIONS—section of a template (or INCLUDE):  
  
#SAMPMAC(MY.DSN,${DVUSVOL},${DVUSTCLS},${DVUSGCLS},${DVUSACLS})

The macro will be expanded by replacing the #PARMS entries with the passed-in names. The names are NOT resolved at the time of replacement – that is performed later during regular processing of the template. This means, for example, that in the example above that #WRITE\_IF=ELSE(${DACLASS}!=””):DATACLAS=${DACLASS} would be expanded to be:

#WRITE\_IF=ELSE(${DVUSACLS}!=””):DATACLAS=${DVUSACLS}

This is mentioned only to let you know that if ${DVUSACLS} was an empty string that it will still work – you wouldn’t end up with #WRITE\_IF=ELSE(!=””):DATACLAS=.

If DVUSACLS was set to ABC then the final result would be:

DATACLAS=ABC

A variable named ${*macname*\_REF} is available within the macro and its included content. This will represent the number of times the macro has been referenced so far within a given step. The first call to a macro will have ${*macname*­\_REF} set to a value of 1. A value of 0 is set for an unreferenced macro, so if a template attempts to reference a ${*macname*\_REF} macro outside of the macro’s instructions then a value of 0 will be the result.

## Velocity Scripting

If you need to do something advanced, please see <http://velocity.apache.org/engine/1.7/vtl-reference.html>

All instructions and content are marked to be interpreted by Velocity. z/OSMF does not expose all of the Velocity “tools” (such as date/time operations) but the reference explains the Velocity syntax for other things, such as string manipulation, defining macros, etc.

The intention is for the workflow builder to mask the majority of Velocity syntax so development teams do not have to learn it. This reference may be more useful to someone coding additional functionality into the workflow builder or providing a common macro shipped with NGI.

Be aware that since Velocity interprets every line of the content and instructions that there may be times something unexpected happens. .Velocity uses things such as the # character as part of its syntax. To prevent Velocity from interpreting a portion of your template, you may escape it using:

#[[ whatever you want in here ]]#

This tells Velocity not to interpret anything within #[[ ]]#

Example:

#[[ This line is not interpreted. I can use # all day long. ##. ]]#

## Shared Properties

(This is currently an experimental feature)

Shared property DDs contain properties that come from a shared component but are NOT being actively configured.. For example, a shared DD may contain all properties used when a common infrastructure environment was configured. The current configuration *uses* that environment but the values (with exceptions below) are NOT to be considered for workflow steps.

You may pass multiple PROPxxxx DDs to the workflow builder. They will be processed in the order in which they appear (well, at least the order in which they appear in the tiot…)

All variables contained in such properties are added to the main PROPERTY list with a scope (prefix) of xxxx-. For example, if PROPINFR is read then INFR- is prefixed to the variable names and they are added to the main property list. A pass is made for all DSN/POP templates for each PROPxxxx DD so that runtime library names can be made available to the workflow templates (with xxxx- scope added).

The DSN/POP templates for these are NOT considered when generating allocation/population steps – they are ONLY used to get the DSN/POP variables created.

A template may request a variable from a shared property by added the scope prefix to the variable. If the variable is found with the given scope then the value from the requested scope is substituted. If it is not found, but there is a variable by the same name *without* a scope specified that that value is substituted.

Let’s say you have a variable called ${DVOLIB}. If DVOLIB is specified in both PROPINFR DD and PROPERTY DD then you will have 2 variables: ${INFR-DVOLIB} and ${DVOLIB}. If a template requests ${INFR-DVOLIB} then it will get the one from the PROPINFR property list. If PROPINFR DD was NOT passed to the workflow then ${INFR-DVOLIB} would not exist but ${DVOLIB} would exist, so a request for ${INFR-DVOLIB} would return ${DVOLIB} since ${INFR-DVOLIB} was not found. Likewise, a template can access a POP/DSN identifier with a given scope, such as ${INFR-BMCLINK}. This will attempt to be resolved from the INFR-scoped variables and, if not found, will use the non-scoped variable list to resolve.

TARGET, which is a special variable representing TARGET dataset information from software instances, is merged from each property AFTER DSN/POP processing has taken place for the main PROPERTY file. This allows templates to access “shared” target datasets so long as they are not in a DSN/POP entry. This prevents DSN/POP templates from attempting to pull common infrastructure targets into product runtimes but still allows access to them for other uses. (QUESTION: Are there places where this is needed? If not, we can skip this merge…)

## Standards

The following are suggested standards when developing templates.

### NAME

The NAME should start with a product or component code to prevent name conflicts. Example: NAME=NGI\_ALLOC

### SAVE-AS

SAVE-AS should be “SAVE-AS=${DVOLIB}($JOBNAME)” for all job workflow steps. This will save the jobs to the customer’s JCL dataset.

### SHARED PROPERTY MEMERS

Shared property tags to SYSIN should be:

PROPINFR - Common Infrastructure  
 PROPDB2C - DB2 Common (do we need this?)